UKRAINIAN CATHOLIC UNIVERSITY

FACULTY OF APPLIED SCIENCES

COMPUTER SCIENCE BACHELOR PROGRAMME

The parallel C++ matrix operations library

Linear Algebra final project report

Authors:
Taras Hrytsiv
Yuriy Shtokhman
Anna-Mariya Hayda
(contributed equally)

15 May 2019

Abstract

Nowadays Linear Algebra helps to solve a great number of problems, not only math-related, but those, which can be met in real life as well. Matrices are used in reality more often than people usually think, unless these individuals are related to some sort of science.

In fact, it is hard to imagine where we don't meet matrices. They and operations with them are used in Photoshop for inverting, video games to build images and make linear transformations, in CS for encrypting or programming basic robot's movements. Students study basic operations in universities, colleges, etc. and often they have to spend plenty of time for doing pretty common things like matrices addition, subtraction, multiplication or finding the inverse. While it is possible for 4x4 matrices – on-hand calculations with 1 000 x 1 000 already sounds like a torture. Existing C++ libraries for LA are either too complex and hard to use effectively or work too long (no paralleling included). This is our main focus – easy to deploy and, at the same time, fast library, which can be installed and held in stride by people from different spheres without great effort.

As a result, we created a C++ library with such operations, as:

Matrices:

1. *Addition;*
2. *Subtraction;*
3. *Multiplication;*
4. *Inverse (LU factorization);*

The code is available on https://github.com/Shtokhman/parallel_LA_library

Keywords: matrix addition • matrix subtraction • matrix multiplication • matrix inverse • matrix parallel library • LU factorization

1 Introduction

In today's epoch of digitalization, matrices are in front of us particularly everywhere - more than trillion pics are taken annually, approximately 2 000 - 3 000 video games are released every year since 2015. It means humans need to be able to work with all that digital stuff efficiently and, at the same time, feel comfortable while using it, no matter what app is running: photo editor with image perspective function, a game with breathtaking graphics or a program for scientific researches. That is why developers or just amateurs all over the world are trying to develop own solutions for one or the other sphere. Matrices are used to render images, build physical systems, encrypting messages and even program accurate robot movements. These operations must be calculated as fast as possible and one of the best solutions for multicore systems (which are extremely popular nowadays!) would be the implementation of parallel working code.

The question is which method to use for implementing Linear algebra algorithms into code and which of them will work the best being paralleled. The classical approaches used for hand-solving problems are not always practical.

Pretty standard algorithms are used when speaking about matrix addition, subtraction or multiplication, meanwhile inverse can be found using cofactors method or Gauss-Jordan elimination. They work correctly, of course, but are far from being excellent in terms of execution time, due to their implementation. However, LU factorization turns out to be a prospective alternative to previously described ones. In this report, we describe methods used in our library too.

2 Approaches

2.1 *Addition and subtraction*

Only equal matrices can be added or subtracted. Matrices are equal if they have the same number of rows and columns respectively.

Let A and B be $m \times n$ matrices. Therefore C = A + B will be matrix of size $m \times n$ as well, since summation forms from corresponding entries in matrices, so that $c_{ij} = a_{ij} + b_{ij}$.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

Example 1

$$\begin{bmatrix} 9 & 1 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 6 & 0 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} 15 & 1 \\ 8 & 6 \end{bmatrix}$$

The algorithm remains the same for *subtraction* procedure. Having equal (of the same size) A and B matrices it is possible to find matrix $C = A - B$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$A - B = \begin{bmatrix} a_{11} - b_{11} & a_{12} - b_{12} \\ a_{21} - b_{21} & a_{22} - b_{22} \end{bmatrix}$$

Example 2

$$\begin{bmatrix}3 & 2\\1 & 18\end{bmatrix} - \begin{bmatrix}1 & 7\\0 & 11\end{bmatrix} = \begin{bmatrix}2 & -5\\1 & 7\end{bmatrix}$$

2.2 *Multiplication*

One of the most important things to keep in mind before multiplying matrices is a validation rule. Only $m \times n$ and $n \times p$ matrices can be multiplied by each other and the product would be $m \times p$ matrix. If the product *AB* is defined, then the entry in row *i* of *A* and column *j* of *B*. If $(AB)_{ij}$ denotes the (*i, j*)-entry in *AB*, and if *A* is $m \times n$ matrix, then

$$(AB)_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$ [1]

$$A = \begin{bmatrix}a_{11} & a_{12}\\a_{21} & a_{22}\end{bmatrix}, \qquad B = \begin{bmatrix}b_{11} & b_{12} & b_{13}\\b_{21} & b_{22} & b_{23}\end{bmatrix}$$

$$AB = \begin{bmatrix}a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23}\\a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23}\end{bmatrix}$$

Example 3

$$\begin{bmatrix}4 & 5\\7 & 21\end{bmatrix} \times \begin{bmatrix}9 & 1 & 5\\4 & 14 & 34\end{bmatrix} = \begin{bmatrix}56 & 74 & 190\\147 & 301 & 749\end{bmatrix}$$

2.3 *Inverse via LU factorization*

Let matrix A be of size $m \times n$. *A* can be written as *A = LU*, where *L* is called *lower triangular* (unit lower triangular matrix because it is invertible) and *U – upper triangular.*

Figure 1. LU factorization [...]

$$A = \underbrace{\begin{bmatrix}1 & 0 & 0 & 0\\ * & 1 & 0 & 0\\ * & * & 1 & 0\\ * & * & * & 1\end{bmatrix}}_{L}\underbrace{\begin{bmatrix}\blacksquare & * & * & * & *\\ 0 & \blacksquare & * & * & *\\ 0 & 0 & 0 & \blacksquare & *\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}}_{U}$$

Example 4

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 3 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -3 \\ 0 & 0 & -3 \end{bmatrix} = LU$$

As soon as we get these two matrices we are up to solving two equations for the number of times. The number is the dimension of our input matrix. We generate an Identity matrix and for its every row we solve following equations:

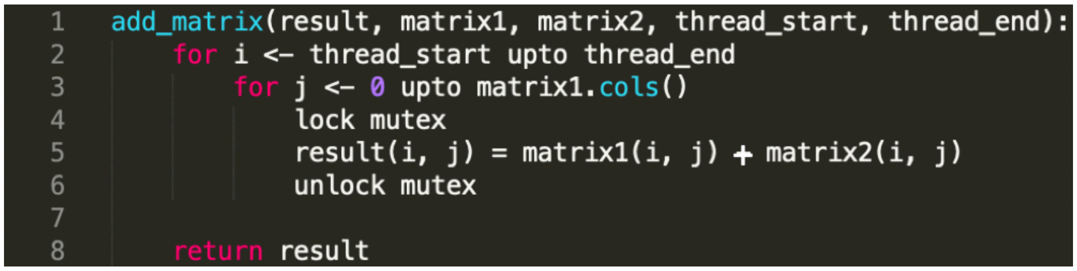
[L] $\times$ *x* = I[i], where *x* is unknown vector of size dim*1 and I[i] is the ith row of an Identity matrix

[U] $\times$ *z* = *x*, where *z* is the column of our inverse matrix

As we mentioned above, such procedure we execute in a loop for a specific number of times and in such a way we obtain our inverse matrix.

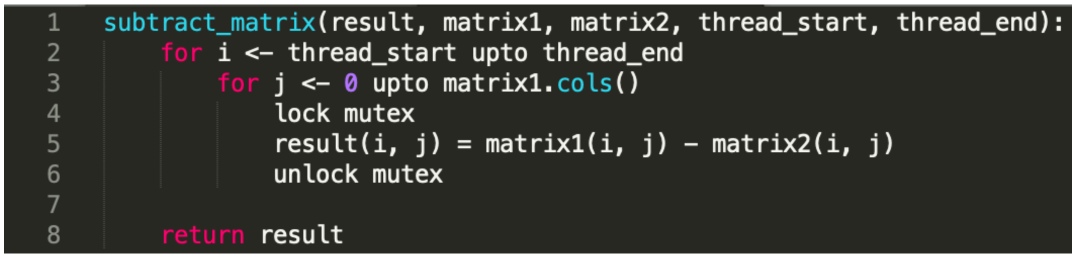
3 Pseudocode

3.1 *Addition*



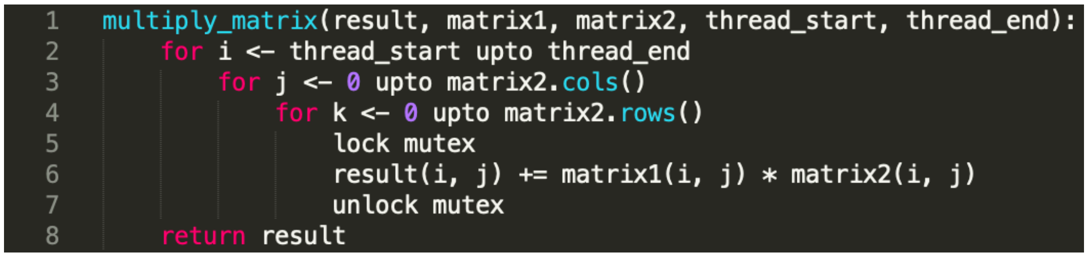
```
add_matrix(result, matrix1, matrix2, thread_start, thread_end):
    for i <- thread_start upto thread_end
        for j <- 0 upto matrix1.cols()
            lock mutex
            result(i, j) = matrix1(i, j) + matrix2(i, j)
            unlock mutex

    return result
```

3.2 *Subtraction*



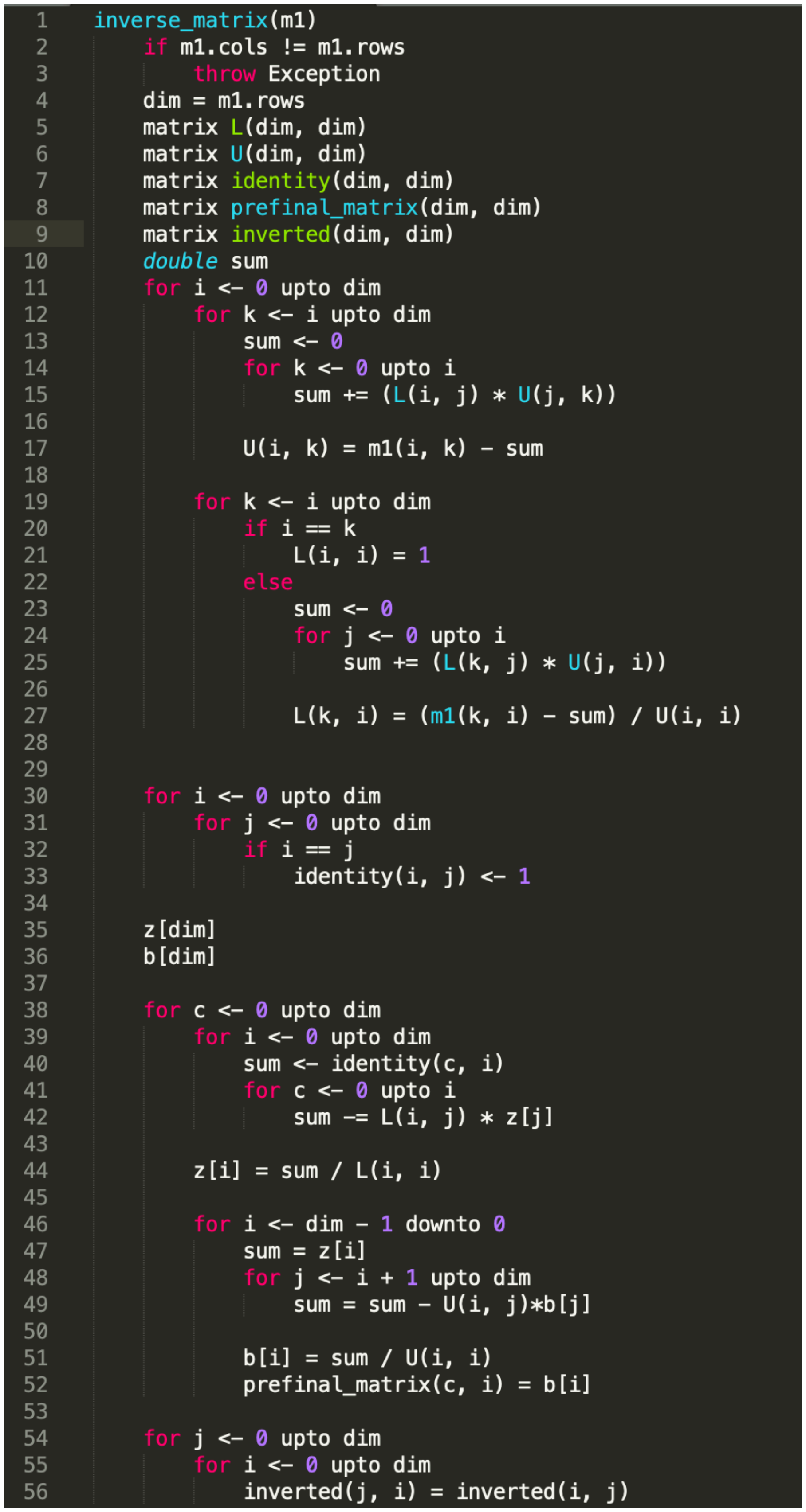
```
subtract_matrix(result, matrix1, matrix2, thread_start, thread_end):
    for i <- thread_start upto thread_end
        for j <- 0 upto matrix1.cols()
            lock mutex
            result(i, j) = matrix1(i, j) - matrix2(i, j)
            unlock mutex

    return result
```

3.3 *Multiplication*



```
multiply_matrix(result, matrix1, matrix2, thread_start, thread_end):
    for i <- thread_start upto thread_end
        for j <- 0 upto matrix2.cols()
            for k <- 0 upto matrix2.rows()
                lock mutex
                result(i, j) += matrix1(i, j) * matrix2(i, j)
                unlock mutex
    return result
```

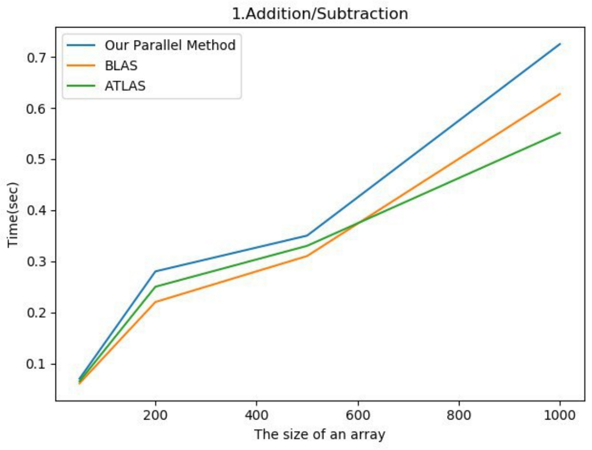
3.4 *Inverse*

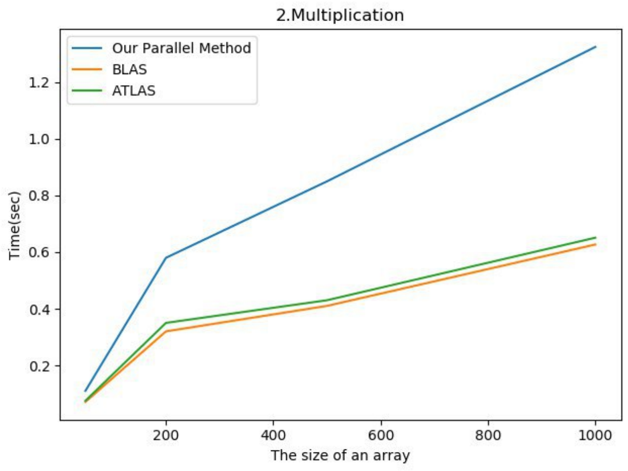


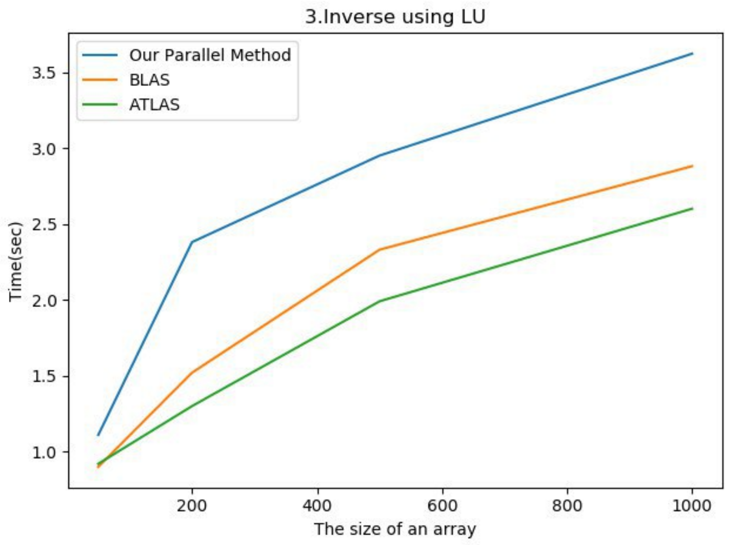
```
inverse_matrix(m1)
    if m1.cols != m1.rows
        throw Exception
    dim = m1.rows
    matrix L(dim, dim)
    matrix U(dim, dim)
    matrix identity(dim, dim)
    matrix prefinal_matrix(dim, dim)
    matrix inverted(dim, dim)
    double sum
    for i <- 0 upto dim
        for k <- i upto dim
            sum <- 0
            for k <- 0 upto i
                sum += (L(i, j) * U(j, k))

            U(i, k) = m1(i, k) - sum

        for k <- i upto dim
            if i == k
                L(i, i) = 1
            else
                sum <- 0
                for j <- 0 upto i
                    sum += (L(k, j) * U(j, i))

                L(k, i) = (m1(k, i) - sum) / U(i, i)


    for i <- 0 upto dim
        for j <- 0 upto dim
            if i == j
                identity(i, j) <- 1

    z[dim]
    b[dim]

    for c <- 0 upto dim
        for i <- 0 upto dim
            sum <- identity(c, i)
            for c <- 0 upto i
                sum -= L(i, j) * z[j]

        z[i] = sum / L(i, i)

        for i <- dim - 1 downto 0
            sum = z[i]
            for j <- i + 1 upto dim
                sum = sum - U(i, j)*b[j]

            b[i] = sum / U(i, i)
            prefinal_matrix(c, i) = b[i]

    for j <- 0 upto dim
        for i <- 0 upto dim
            inverted(j, i) = inverted(i, j)
```

4 Conclusions

In this report we explained how our library works, including theoretical and practical sides. Important notions and explanations necessary for general understanding were provided, which allowed to give an example of efficient and relatively easy in terms of implementation algorithms. Besides that, after some amount of attempts we discovered that number of threads which is close to the most optimal one can't be an absolute one – it is unique for each machine.
We have introduced pseudocode in this report too. It won't require much effort to catch the idea. Moreover, it will be easier for 'strangers' to suggest improvements in future, since one of our key goals is to make our library the open source one. It means other developers will be able to send commit requests and, as a result, library could once become well-known.
We have tested our library performance. Thanks to these tests it is better visible now, that our library better fits for bigger matrices (50 at least), however it still shows pretty nice results working with fairly small matrices, like 10x10.







References:

[1] Lay D.C.: *Linear algebra and its applications*, Fourth edition (2012)

[2] Strang G.: *Linear algebra and its applications*, Fourth edition (2006)

[3] Hudik M., Hodon M.: *Performance optimization of parallel algorithm (2014)*

[4] Hudik M., Hodon M.: *Modelling, optimization and performance prediction of parallel algorithms* (2014)