FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Efficient multiplication of sparse matrices
Student:	Ladislav Bartůněk
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

1) Study various sparse matrix storage formats. [1, 2]
2) Study algorithms for matrix multiplication and discuss their usage in sparse matrix multiplication. [3, 4]
3) Implement selected algorithms in C/C++ after an agreement with the supervisor.
4) Optimize these algorithms and parallelize them using Open MP technology
5) Test these algorithms on your own generated sparse matrices and on sparse matrices available from publicly available sources (e.g. MatrixMarket, ...).
6) Measure the performance of these algorithms on faculty server STAR.
7) Compare the performance with already existing libraries solving sparse matrix multiplication (e.g. Eigen [5] library).

References

[1] DONGARRA, Jack. Survey of Sparse Matrix Storage Formats [online]. 1995 [cit. 2017-02-22]. Dostupné z: http://www.netlib.org/linalg/html_templates/node90.html
[2] LICHÝ, Lukáš. Efektivní násobení řídkých matic [online]. ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE, 2015. Dostupné také z: https://dip.felk.cvut.cz/browse/pdfcache/lichyluk_2014bach.pdf. Bakalářská práce.
[3] ZVONÍK. Rychlé násobení matic a Strassenův algoritmus ve Winogradově úpravě [online]. 2010. Dostupné z: http://people.fjfi.cvut.cz/pelanedi/tema/skriptatemakonecsemestru/strassenzvonik.pdf
[4] YUSTER, Raphael a Uri ZWICK. Fast Sparse Matrix Multiplication. ACM Transactions on Algorithms (TALG) [online]. 2005. Dostupné z: http://dl.acm.org/citation.cfm?id=1077466
[5] Eigen: Dostupné z: https://eigen.tuxfamily.org/dox-devel/group__TutorialSparse.html

doc. Ing. Jan Janoušek, Ph.D.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague October 4, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Efficient multiplication of sparse matrices

Ladislav Bartůněk

Department . . . Department of Theoretical Computer Science
Supervisor: Ing. Ivan Šimeček, Ph.D.

May 15, 2019

Acknowledgements

I would like to thank my supervisor Ing. Ivan Šimeček, Ph.D. for having godly patience with me.

Declaration

In Prague on May 15, 2019 . . . .

Citation of this thesis

Bartůněk, Ladislav. *Efficient multiplication of sparse matrices.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce popisuje formáty řídkých matic XY, YX a CRS, algoritmy násobení řídké matice s řídkou maticí a zároveň popisuje algoritmy používané pro paralelní algoritmy. Součástí práce je implementace v C++, testy nad reálnými matice a porovnání výsledků s vypočítanými předpoklady

Klíčová slova řídké matice, násobení řídkých matic, paralelní algoritmy

Abstract

This work describes formats of sparse matrices XY, YX and CRS, algorithms of multiplication of sparse matrix with sparse matrix and describes algorithms used in parallel computation. Thesis includes implementation in C++, tests with real matrices and comparison of results with calculated expectations

Keywords sparse matrices, sparse matrix multiplication, parallel algorithms

Contents

List of Figures

List of Tables

Introduction

Storing data is a very common operation in computer science, how to approach it and how to handle the data afterwards can be a big task and it has been improving ever since there were first computers. Using the correct storage formats can mean difference between a very fast program and an unusable binary blob.
Matrices are one of the most important storage formats in computer science. They are used in most of the modern programs. Sometimes matrices contain a large number of zero elements. These matrices are called sparse matrices and we can and should use this fact to improve performance of various operations on those.
The multiplication of matrix and matrix are one of the very basic operations, as such these should get proper attention while dealing with matrix storage formats.
This study looks into multiple ways to store matrices, such as XY, YX and CRS and handle these multiplications with these matrices, and how they are handled during parallel computation.

Chapter 1

Theoretical background

This chapter will explain most of the terms used in this thesis.

1.1 Matrices

A matrix is a rectangular arrangement of numbers or other mathematical objects. In computer science it is used as a two-dimensional array. Matrices have a large use in computer graphics. A matrix A of a type (m, n) is a set of $m \cdot n$ values. Element $a_{r,c}, r \in 1, 2, \ldots, m, c \in 1, 2, \ldots, n$ is placed on row r and column c.
Symbolical example of matrix A of a type (m, n)

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

1.2 Sparse matrices

Sparse matrices are typically big objects containing a large number of zero values. In contrast, if most of the elements are non-zero, then the matrix is considered dense. Large sparse matrices often appear in scientific or engineering applications when solving partial differential equations.

1.2.1 Density

Density is the measurement of how much a row, column or matrix is filled with elements. The easy way to find out density is the $number\_of\_elements/size\_of\_object$. For example a matrix with 5 rows and 6 columns with 15 elements will have density equal to:

$$\rho = 15/(5 \cdot 6)$$

$$\rho = 15/30$$

$$\rho = 50\%$$

1.2.2 Order

Square matrix is matrix with equal rows and columns. Matrix with n rows and columns has it's order equal to n.

1.3 Storage formats

Dense matrices are most commonly stored as a two-dimensional array, as opposed to large sparse matrices which require a different storage format. Those formats ignore zero values and concentrate on non-zero elements and expect every non-defined element to be a zero value.

Few examples:

- Coordinate (XY or YX)
- Compressed Row Storage (CRS) [3]
- Compressed Column Storage (CCS) [4]
- Block Compressed Row Storage (BCRS) [5]
- Compressed Diagonal Storage (CDS) [6]
- Jagged Diagonal Storage (JDS) [7]
- Skyline Storage (SKS) [8]
- Quad Tree (Q-tree) [9]

This work uses dense, coordinate and CRS. Coordinate format is used, as it is, used by MatrixMarket. For use in the multiplication operations I chose Coordinate format and CRS format, as they are effective and simple storage formats, that do not rely on specific spread of values to be efficient. Dense matrices are used as result containers due to their low time complexity for storing values.

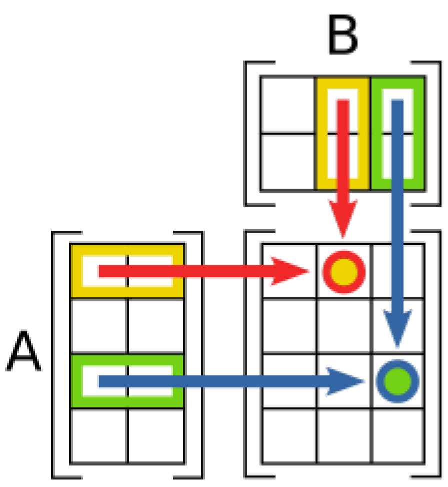

Figure 1.1: Graphical example of elements of matrix being processed with operation of matrix multiplication [1]

1.4 Multiplication of matrix with matrix

Multiplication of matrix and matrix is commonly used in computer science. A good example of sparse matrix multiplication is the method of finite elements [10].
Multiplication is such transformation, that takes matrix $A_{m,n}$ and matrix $B_{n,p}$ and transforms it into matrix product $AB_{m,p}$. 1.1 Where each entry is defined as:

$$(ab)_{ij} = \sum_{k=1}^{m} a_{ij} B_{kj}$$

1.5 Time complexity

Time complexity, or in other words algorithmic complexity, is being used in computer science to evaluate the performance of an algorithm. The goal of time complexity is to classify algorithms according to their performances independent on details of implementation. For this purpose $O(n)$ notation is mostly used where n describes size of the input to be processed. [11]

1.6 Parallel computation

1.6.1 Motivation

”For many decades, Moore’s law has bestowed a wealth of transistors that hardware designers and compiler writers have converted to usable performance, without changing the sequential programming interface. The main techniques for these performance benefits—increased clock frequency and smarter but increasingly complex architectures—are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster. This historic shift to multi-core processors changes the programming interface by exposing parallelism to the programmer, after decades of sequential computing.” [12]

Even though this quote is quite old, it’s still relevant today. Multi-thread execution of code can be used for several purposes. Main ones are for network communication, execution of user interface while executing code in the background, or to increase performance in areas where heavy computation is needed. Multi-threaded execution can be performed even on single core processors, however you cannot obtain any speed-ups on those. Executing a sequential code on a multi-core or multi-threaded processors can be a waste of the potential of a given processor, if it is otherwise unoccupied. Therefore we want to to maximize the potential by using all cores and threads of the processor for our heavy computation.

With multi-core technology progressing further, especially on graphics processing units, the parallel computation is more important. This can be used in multiple ways. One can be everyday enjoyment in the form of games, which can take a huge amount of processing power [13]. This is due to heavy 3-D rendering, which is also used in the film industry. Another usage can be in the form of cryptocurrency mining [14], which can consume near to an infinite amount of power. The most notable usage is probably volunteer computing [15], where vast numbers of computers can be used to parallel science computation that would take an enormous amount of time even on our best super computers.

1.6.2 Description and synchronization

Multi-core processors have two main layers. One layer is the individual cores, which have their own cache memory, while the second layer handles management of the individual cores and serves as a bus between the cores

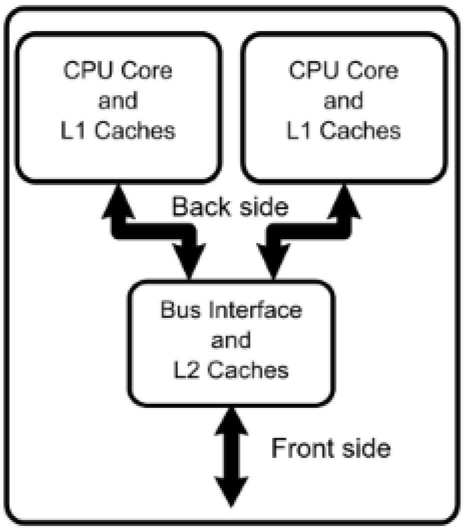

Figure 1.2: Generic example of dual core processor [2]

and the rest of the computer. 1.2 This can create several issues regarding synchronization. These are called race conditions [16], the most common of which is that two threads read a value, while both want to add a value to it. Both add their value in their cache memory and pass it down to the main bus, which sequentially writes both of these values into the memory, resulting in only one value being added while the algorithm goes on.

Another problem that is less common, but still potentially dangerous to the computational power, is false sharing. This may happen if array elements share a cache line and both are updated at the same time, which may lead to sloshing of independent data back and forth, noticeably slowing down computation. [17]

1.6.3 Thread local storage

Thread local storage (TLS) is a dedicated storage area that can only be accessed by one thread. TLS variables can be seen as global variables that are only visible to a particular thread and not the whole program. [18] As such, this area can be used to spread different values to different threads, or to contain results specific for a given thread until a later time, where we will synchronize these values with the main thread.

Chapter 2

Storage formats for matrices

This section describes all matrix formats used in this thesis. Coordinate and CRS formats are used for multiplication, while dense matrix formats are used for saving results.

2.1 Dense matrix

Both dense matrix implementations are implemented as two-dimensional arrays, where the individual rows or columns must be individually allocated to prevent false sharing during parallel computing. The main motivation behind implementing both of these is the time complexity of saving the results from the multiplication. These two implementations are used for two different multiplications. DenseMatrixY is internally ordered first by the Y coordinate and then by the X coordinate, whereas DenseMatrixX is internally ordered first by the X coordinate. As such, if one thread is using only one row or column (depending on which implementation and multiplication is used) for storing values and no other thread will be using it at any point in time, we can safely use them without synchronization and without fear of race conditions and false sharing. 1.6.2

2.2 Coordinate XY

Coordinate format is a basic format for sparse matrices used commonly on MatrixMarket. Each non-zero element is stored as a triplet of coordinate X, coordinate Y, and a value which is supposed to be stored at these coordinates. XY format is ordered first by X coordinate and then by Y coordinate. Format can be defined as three arrays, each storing a given element of the triplet or as a single structure array where one element of the array equals one element of the sparse matrix.

A matrix stored as:

X	Y	val
1	6	1
2	3	6
3	3	8
3	4	7
4	2	9
4	6	4
5	1	2
5	5	3
6	3	1

Table 2.1: Storage format representation of XY matrix

Will have following graphical representation:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 8 & 7 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 4 \\ 2 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

2.3 Coordinate YX

The only difference between XY and YX is its order of elements. Coordinate YX is ordered first by Y coordinate and then by X coordinate, so the same example from Coordinate XY would be internally represented like:

X	Y	val
5	1	2
4	2	9
2	3	6
3	3	8
6	3	1
3	4	7
5	5	3
1	6	1
4	6	4

Table 2.2: Storage format representation of YX matrix

2.4 Compressed row storage

Compressed Row Storage formats is one of the most general formats. They make absolutely no assumptions about the sparsity of the matrix and they don't store any unnecessary elements. However, they are not very efficient in addressing.

Compressed row format consists of three arrays. First two arrays may be implemented as one array and they consist of x indexes and values of elements. Third array points to the first two arrays to determinate where individual row starts. [19]

A matrix stored as:

val	3	2	5	1	7	8	5	6	2	3	2
colInd	3	5	5	2	6	1	4	6	3	2	5

rowPtr	1	3	4	6	9	10

Table 2.3: Storage format representation of CRS matrix

Will have following graphical representation:

$$\begin{pmatrix} 0 & 0 & 3 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 1 & 0 & 0 & 0 & 7 \\ 8 & 0 & 0 & 5 & 0 & 6 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 2 & 0 \end{pmatrix}$$

Chapter 3

Realization

This chapter will describe whole part of my implementation work.

3.1 Software used

In this section I will mention every software and foreign code I have used to implement this work.

3.1.1 Third party code

3.1.1.1 ANSI C library for Matrix Market I/O

For easy usability of MatrixMarket format I have used ANSI C library for Matrix Market I/O [20], it is a small library with the code included, which helped me to read and write MatrixMarket format to use in my implementation.

3.1.1.2 OpenMP library

For easy parallel computation I used OpenMP library. [21]

3.1.1.3 Eigen library

As a third party library for sparse matrix multiplication I chose Eigen library. [22] From this library I used sparse·sparse matrix multiplication, sparse·dense matrix multiplication in sequence and in parallel.

3.1.2 Wolfram Mathematica

For various calculations, verifying correctness of output and other small tasks I have used Wolfram Mathematica. [23]

3.1.3 Doxygen and Graphviz

For generating UML diagram I used Doxygen [24] with Graphviz [25] installed.

3.1.4 Gnuplot

For generating various graphs in this thesis. [26]

3.1.5 Git

As a versioning software and for back up purposes I used Git [27] with bitbucket.org as a repository storage [28].

3.1.6 Valgrind

For various testing purposes I used Valgrind [29].

3.2 Hierarchy

This section will describe every implemented class used in my project. All storage formats were implemented as described in 2. The UML diagram is available on 3.1

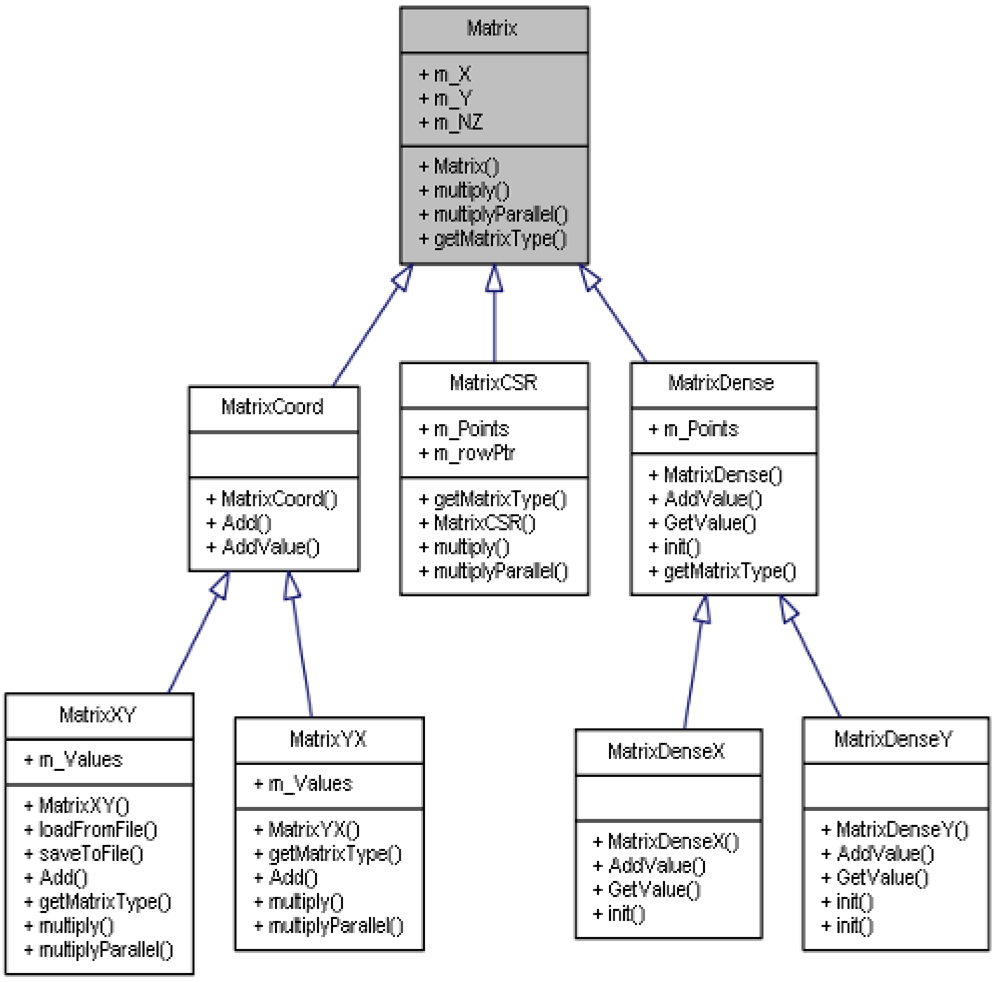

Figure 3.1: UML graph of Matrix Hierarchy

3.2.1 Matrix

The main superclass is called Matrix, every other matrix storage class derives from it. It handles correct multiplications, base variables. It also handles defining second matrix when multiplying with unknown matrix.

3.2.2 MatrixDense

MatrixDense is a superclass to MatrixDenseX and MatrixDenseY. This makes things a little bit more easy to use and ensures, that both of these have the correct functions.

3.2.3 MatrixDenseX

MatrixDenseX is a dense matrix that orders arrays first by X coordinate. It is used in XY·YX multiplication for storing results.

3.2.4 MatrixDenseY

MatrixDenseX is a dense matrix that orders arrays first by Y coordinate. It is used in CRS·CRS and YX·XY multiplication for storing results.

3.2.5 MatrixCoord

MatrixCoord is superclass to MatrixXY and MatrixYX. It commonizes several functions and makes some handling more easy.

3.2.6 MatrixXY

Coordination XY format is used as a base for all input matrices from MatrixMarket and it is implemented to be capable of converting from and to CRS format and from dense matrix. It is the only one implemented to be able to be loaded from a file and saved to a file.

This class allow multiplication with YX format sequentially and in parallel.

3.2.7 MatrixYX

This class allow multiplication with XY format sequentially and in parallel.

3.2.8 MatrixCRS

This class handles operations with CRS format matrix. This class doesn't contain any uncommon functions, only base functions needed by all matrices and multiplication with itself.

3.3 Algorithms

All algorithms were implemented as described below. CRS format has been implemented to be multiplied with itself and YX has been implemented to be multiplied with XY. Both of these multiplications were implemented in both sequence and parallel. Results of all algorithms were tested with each other and one was tested with third party multiplication algorithm, the software itself has a parameter to run these tests.

3.3.1 Abbreviations

These are the abbreviations used in this section, especially in the algorithms themselves:

- A - Matrix A
- B - Matrix B
- A_{NZ} - Number of non-zero elements in matrix A
- A_X - Number of columns in matrix A
- A_Y - Number of rows in matrix A
- A_x - Elements of column x from matrix A
- A_y - Elements of row y from matrix A
- A_{ax} - Average number of elements in a row of matrix A
- A_{ay} - Average number of elements in a column of matrix A
- in parallel - Given for cycle will be executed in parallel by multiple threads
- res - Result matrix
- p - Number of processes
- $\rho(A)$ - Density of matrix A
- TLS - Thread local storage
- $Point.X$, $Point.Y$ - X and Y coordinates respectively of given $Point$
- $Point.Value$ - Value associated with given $Point$

3.3.2 Thread architecture

All my parallel algorithms use all threads provided to calculate the results. However in some there's a master thread, that performs some secondary calculations that need to be calculated just once and need to be done sequentially. These tend to be much smaller than the parallel calculation.

3.3.3 Storing of results

There are several common problems. One of them is the problem of storing the results. There are several options, most notable are vector of XY matrix, regular map, hash map and DenseMatrix. Vector has a time complexity of $O(1)$ of storing values that are already existing, however when storing non-existing values it reaches $O(n)$, which is totally unacceptable. Map has time complexity of storing values $O(log(n))$, which is still a little bit too slow. For Hash map although it has best time complexity on paper for storing results, which is equal to $O(1)$, the overhead of those operations can be quite slow. However if we preallocate the entire resut matrix with DenseMatrix, we can get time complexity of $O(1)$ with next to zero overhead. However this comes at a cost, that we need to preallocate all fields upfront. This cost is $O(res_X \cdot res_Y)$, which may be better or worse depending on the density of the result matrix. This also means, that when merging results in parallel algorithms, we need to check the entire result sub-matrices if the values are present.

The density of the result matrix tends to be much bigger than a density of input matrices, as there needs to be just one matching pair of elements from matrices A and B to form an element in result Matrix, however elements from matrices A and B can be reused to form more results. 1.4

3.3.4 Seq. alg. for CRS · CRS

This multiplication works by taking a single element and multiplying it with all elements it needs to be multiplied with while storing the result in a result dense matrix. As every element from matrix A needs to get multiplied with every element from row B_y, this search is pretty easy thanks to having row pointers in second matrix.

```
procedure CRSMultCRSSeq($A, B, res$)
    init(res)
    for $i \leftarrow 1$ to $A_Y$ do
        for $j \leftarrow A.rowPtr[i]$ to $A.rowPtr[i+1]$ do
            $idx \leftarrow A.indexes[j]$
            $val \leftarrow A.values[j]$
            for $l \leftarrow B.rowPtr[idx]$ to $B.rowPtr[idx+1]$ do
                $res[i][B.indexes[l]] \leftarrow res[i][B.indexes[l]] + val \cdot B.values[l]$
            end for
        end for
    end for
end procedure
```

- Initialize and null result matrix in time $O(B_X \cdot A_Y)$
- For each row Y in A_Y in time $O(A_Y)$
    - Find begin and end of given row in time $O(1)$
    - For each non-zero element of given row in time $O(A_y)$
        - Find X coordinate of given element in time $O(1)$
        - Find Value of given element in time $O(1)$
        - Using given X, find beginning and end of the row in time $O(1)$
        - For each member of given row in time $O(B_{ax})$
            - Multiply given element with found element in matrix A and save them in time $O(1)$

$A_Y \cdot A_y$ can be replaced with A_{NZ}. By just adding and multiplying these complexities we can get overall complexity of this algorithm, which is:

$$O(B_X \cdot A_Y + A_{NZ} \cdot B_{ax})$$

3.3.5 Par. alg. for CRS · CRS

For parallel multiplication there are several approaches. One approach is to have each thread store their own results in their own result matrix, and then merge these results in a critical region of code. This can be sped up by each thread remembering, which Y coordinate they accessed, as the parallel split is happening on for, that resolves, y coordinate for the result matrix.

I chose simpler approach, which is to use the appropriate result matrix, which is DenseMatrixY. DenseMatrixY contains individually allocated rows of the dense matrix, as such, we can ensure there are no false sharing or race conditions from multiple processes accessing the same region of memory, and that we can store the output matrix directly and without splitting any results into different matrices. This process saves every result created directly to the result matrix. As such this approach could be slower at higher densities of matrices, but at lower densities this algorithm should outperform any more complicated approach, which will be addressed later in this thesis.

```
procedure CRSMultCRSPar($A, B, res$)
    parallelInit(res)
    for $i \leftarrow 1$ to $A_Y$ do in parallel
        for $j \leftarrow A.rowPtr[i]$ to $A.rowPtr[i+1]$ do
            $idx \leftarrow A.indexes[j]$
            $val \leftarrow A.values[j]$
            for $l \leftarrow B.rowPtr[idx]$ to $B.rowPtr[idx+1]$ do
                $res[i][B.indexes[l]] += val \cdot B.values[l]$
            end for
        end for
    end for
end procedure
```

- Initialize and null result matrix in time $O(B_X \cdot A_Y)$
- For each row Y in A_Y do in parallel in time $O(A_Y/p)$
    - Find begin and end of given row in time $O(1)$
    - For each non-zero element of given row in time $O(A_y)$
        - Find X coordinate of given element in time $O(1)$
        - Find Value of given element in time $O(1)$
        - Using given X, find beginning and end of the row in time $O(1)$
        - For each member of given row in time $O(B_{ax})$
            - Multiply given element with found element in matrix A and save them in time $O(1)$

If matrix A is spread reasonably well, then $A_Y \cdot A_y/p$ can be replaced with A_{NZ}/p. The most-inner cycle can be expressed as $O(B_{ax})$. The overall complexity is:

$$O(B_X \cdot A_Y + (A_{NZ}/p) \cdot B_{ax})$$

3.3.6 Seq. alg. for XY · YX

The XY multiplied with YX needs to do a search before the inner cycle. The search itself would have complexity of $O(log(B_{NZ}))$. With the cycle that it would be wrapped in the complexity would be $O(A_{NZ} \cdot log(B_{NZ}))$. This can be reduced into $O(B_{NZ})$ by preparing the result of the searches by iterating through the second matrix and saving the searched values into an array. From now on this will represented by *PrepareSearches* function. With this the algorithm is pretty similar to the one used for Compressed Row Storage format multiplication.

```
 1: procedure XYMultYXSeq($A, B, res$)
 2:     init(res)
 3:     $secondBegins \leftarrow PrepareSearches(B)$
 4:     for $i \leftarrow 1$ to $A_{NZ}$ do
 5:         $PA \leftarrow A.Points[i]$
 6:         for $j \leftarrow secondBegins[PA.X]$ to $secondBegins[PA.X + 1]$ do
 7:             $PB \leftarrow B.Points[j]$
 8:             $res[PB.X][PA.Y] += PA.Value \cdot PB.Value$
 9:         end for
10:     end for
11: end procedure
```

- Initialize and null result matrix in time $O(B_X \cdot A_Y)$
- Perform preparation of searches in time $O(B_{NZ}))$
- For each element in A in time $O(A_{NZ})$
    - For each element in B_x in time $O(B_{ay})$
        - Multiply given elements and store them in time 1

By just adding and multiplying we can get time complexity of:

$$O(B_X \cdot A_Y + B_{NZ} + A_{NZ} \cdot B_{ay})$$

3.3.7 Par. alg. for XY · YX

This algorithm is just a sequential algorithm with parallel elements added into them. The most obvious is that the main loop now runs in parallel. For synchronization there's a sum of results at the end, where the entire result matrix is split into $(p - 1)$ parts, and the algorithm is run $(p - 1)$ times. In each run every thread takes a different part adding all results matching the part of their private result matrix into the final result matrix, allowing to merge the results in time $O(A_Y \cdot B_X)$.

```
1: **procedure** XYMultYXPar($A, B, res$)
2:     init(res) for all processes
3:     $SecondBegins \leftarrow PrepareSearches(B)$
4:     **for** $i \leftarrow 1$ **to** $A_{NZ}$ **do** in parallel
5:         $PA \leftarrow A.Points[i]$
6:         **for** $j \leftarrow SecondBegins[PA.X]$ **to** $SecondBegins[PA.X + 1]$ **do**
7:             $PB \leftarrow B.Points[j]$
8:             $TLS.res[PB.X][PA.Y] += PA.Value \cdot PB.Value$
9:         **end for**
10:    **end for**
11:    MergeResults
12: **end procedure**
```

- Initialize and null result matrices in time $O(B_X \cdot A_Y)$
- Perform preparation of searches in time $O(B_{NZ})$
- For each element in A in time $O(A_{NZ}/p)$
  - For each element in B_x in time $O(B_{ay})$
    - Multiply given elements and store them in time 1
- Merge results in $O(A_Y \cdot B_X)$

By just adding and multiplying we can get time complexity of

$$O(B_X \cdot A_Y + B_{NZ} + A_{NZ} \cdot B_{ay}/p + A_Y \cdot B_X)$$

which can be summarized to:

$$O(2 \cdot A_Y \cdot B_X + B_{NZ} + A_{NZ} \cdot B_{ay}/p)$$

3.3.8 Seq. alg. for YX · XY

The YX multiplied with XY is really slow unless you convert the second matrix into a format that is sorted by Y coordinate. As such there's not much we can do to prevent this without using algorithm to multiply a Coordinate matrix with YX format, which is described at another section of this chapter. This algorithm won't be measured in the results, as it is too slow to compare with other algorithms.

```
procedure YXMultXYSeq($A, B, res$)
    init(res)
    for $i \leftarrow 1$ to $A_{NZ}$ do
        for $j \leftarrow 1$ to $B_{NZ}$ do
            if $A.Points[i].X == B.Points[j].Y$ then
                $PA \leftarrow A.Points[i]$
                $PB \leftarrow B.Points[j]$
                $res[PB.X][PA.Y] += PA.Value \cdot PB.Value$
            end if
        end for
    end for
end procedure
```

- Initialize and null result matrix in time $O(B_X \cdot A_Y)$
- For each element in A in time $O(A_{NZ})$
    - For each element in B in time $O(B_{NZ})$
        - Check if the elements should multiply and save them in time 1

By just adding and multiplying we can get time complexity of:

$$O(B_X \cdot A_Y + A_{NZ} \cdot B_{NZ})$$

3.3.9 Par. alg. for YX · XY

This uses the same main cycles as XY·YX sequential multiplication. However as source matrices are in form of YX and XY, the calculations can be easily segmented into sub-matrices, which can streamline the process of multiplication. The first two *PrepareSearches* calls are used internally to divide matrices into sub-matrices. This algorithm splits the matrix A into p sub-matrices As_{YX} divided by the Y coordinate. As such we need to use DenseMatrixY to store data to prevent false sharing and race conditions. All result matrices are initiated only on coordinates, that are actually going to be used. The matrix B is split by the master process into p Bs_{XY} matrices, that are converted into Bs_{YX} matrices and *PrepareSearches* generates data for faster processing by the inner cycle. After that, every thread iterates their As_{YX} sub-matrix multiplying needed elements from given Bs_{YX}. After all multiplications are done, every process, except the master, saves all the values multiplied into the final result matrix, while master process prepares another batch of Bs_{YX}.

```
procedure YXMultXYPar($A, B, res$)
    $PrepareSearches[A]$
    $PrepareSearches[B]$
    for $i \leftarrow 0$ to $p$ do in Parallel
        $TLS.As \leftarrow A.submatrix(i)$
    end for
    init(res) for all processes
    Parallel begin
    for $repeat \leftarrow 0$ to $p$ do
        if $master$ then
            $Bs \leftarrow B.submatrix(repeat)$
            $Bs \leftarrow ConvertXY - YX(Bs)$
            $subBegins \leftarrow PrepareSearches[Bs]$
        end if
        $barrier$
        for $i \leftarrow 1$ to $TLS.As_{NZ}$ do
            $PA \leftarrow TLS.As.Points[i]$
            for $j \leftarrow subBegins[PA.X]$ to $subBegins[PA.X + 1]$ do
                $PB \leftarrow Bs.Points[j]$
                $TLS.res[PB.X][PA.Y] += PA.Value \cdot PB.Value$
            end for
        end for
        MergeResults
    end for
    Parallel end
end procedure
```

- Perform preparation of searches in time $O(A_{NZ} + B_{NZ})$
- Divide matrix A into sub-matrices As in time $O(A_NZ/p)$
- Initialize result matrices in time $O(B_X \cdot A_Y)$
- For each thread p in time $O(p)$
    - Master extracts the sub-matrix Bs in time $O(B_{NZ}/p)$
    - Master reorders the sub-matrix Bs in time $O((B_{NZ}/p) \cdot log(B_{NZ}/p))$
    - Master prepares searches in time $O(B_{NZ}/p)$
    - For each non-zero element for sub-matrix As in time $O(As_{NZ})$
        - for each non-zero element of row Bs_y in time $O(Bs_{ay})$
            - multiply and save value in time $O(1)$
    - Merge sub-results into result matrix in time $O((A_X/p) \cdot (B_Y/p))$

First for clarity we need to estimate a few values. If the matrix is reasonably even, then $O(As_{NZ})$ can be estimated as $O(A_{NZ}/p)$. Same goes for $O(Bs_{ay})$, which would become $O(B_{ay}/p)$.
By replacing estimated values, adding and multiplying provided complexities pre main loop we get:

$$O(pre\_main\_loop) = O(A_{NZ} + B_{NZ} + A_N Z/p + B_X \cdot A_Y)$$

The main loop is a little bit more tricky. The merging and Master tasks will be done sequentially once, however for the rest of the part master does the preparing, while rest of the processes do the merging part. So for the rest it depends on what takes more time. If we look at their complexities, the master tasks take $O(B_{NZ}/p(1 + log(B_{NZ}/p)))$ and the merging takes $O((A_X/p) \cdot (B_Y/p))$. So the preparations for the inner loop take:

$$O(inner\_loop\_prep) = O(B_{NZ}/p(1 + log(B_{NZ}/p)) + (A_X/p) \cdot (B_Y/p) +$$

$$+ max(B_{NZ}/p(1 + log(B_{NZ}/p)) + (A_X/p) \cdot (B_Y/p)) \cdot (p - 1))$$

This already takes into account all the runs done, so there's no need to multiply with p in the final complexity.
The inner loops are pretty simple:

$$O(inner\_loops) = O((A_{NZ}/p) \cdot (B_{ay}/p))$$

The overall complexity can be expressed as:

$$O(pre\_main\_loop) + O(inner\_loop\_prep) + p \cdot O(inner\_loops))$$

3.4 Summary

For sequential algorithms, CRS·CRS and XY·YX are similar if you consider only the time complexity from the most inner loop in these two, with CRS·CRS having the complexity of $O(A_{NZ} \cdot B_{ax})$ and XY·YX having $O(A_{NZ} \cdot B_{ay})$. The difference in the overhead is just with XY·YX having extra $O(B_{NZ})$, which is negligible. The YX·XY has time complexity $O(A_{NZ} \cdot B_{NZ})$, which is much slower than rest of these two and therefore isn't even worth measuring, as the results would be on completely different scale.

For parallel algorithms YX·XY and XY·YX have identical time complexity from the most inner loop of $O(A_{NZ} \cdot B_{ay}/p)$, however YX·XY handles the data a lot better and doesn't require any synchronization of data afterwards, and therefore should outperform the XY·YX. CRS·CRS algorithm has similar the most inner loop time complexity of $O(A_{NZ} \cdot B_{ax}/p)$, and considering that CRS is much simpler algorithm with very little overhead, it should be on par with YX·XY. This very much depends on if the streamlining of the data is worth the speed-up.

Chapter 4

Testing

This chapter contains all measurements done for implemented algorithms. Namely CRS·CRS in sequence (3.3.4) and in parallel (3.3.5), XY·YX in sequence (3.3.6) and in parallel (3.3.7) and YX·XY in parallel (3.3.9). It also contains measurements done to Eigen library (3.1.1.3). Eigen library is used for it's sparse matrix multiplication with sparse matrix and with dense matrix in sequence and in parallel.

4.1 Methodology of measurement

For each result there were at least 10 instances of independent measurement over the same matrices. This was done to prevent the measurement errors and attempt to show the true potential of algorithms, which can be off set by various random events. However with longer running processes there are still noticeable measurement errors for the slower algorithms.

4.2 Hardware and software

Measurement was done on Star cluster on node-017 (gpu-01) with following configuration:

- CPU: 2x 6core Xeon 2620 v2 @ 2.1Ghz
- RAM: 31978 MB
- Compiler version: g++ 4.8.5
- OpenMP version: 3.1

All programs were compiled as C++ code with -O3 optimization.

4.3 Measurements

This section will contain individual measurements and comparisons done with some more details about why these measurements were done.

The variables in these testings are:

- Density - 1.2.1 describes how many elements are there in the matrix
- Size - This represents the order of the matrix
- Thread count - This applies only to parallel algorithms and represents how many parallel computations are launched at the same time.

4.3.1 Sequential algorithms

4.3.1.1 My algorithms

The figure 4.1 is representing one of the two base benchmarks of a sparse matrix multiplication algorithm. These benchmarks test performance of the algorithm over various sizes with equal density of the matrix. These two sequential algorithms (3.3.4, 3.3.8), although they are quite similar with their time complexity, have vast difference with bigger sizes and therefore more elements. From the figure we can see that at lower sizes the difference is quite negligible and the main reason, why one could be a lot faster than the other is cache optimization. The CRS·CRS algorithm uses the same array to store results in one main cycle, therefore this should result in less cache misses, while the YX·XY jumps with the result saving over almost the entire matrix.

The figure 4.2 represents the other base benchmark of a sparse matrix multiplication and that is various densities of the matrix with fixed size of the matrix. In this figure CRS·CRS algorithm is again superior, while XY·YX having comparable performance at the around 1% density and less and vastly different performance already at 2% density.

4.3.1.2 Eigen comparison

The figure 4.3 compares my best sequential algorithm with two sequential from Eigen (3.1.1.3) library. The sparse·sparse algorithm was chosen, because it is probably the most similar algorithm to the CRS·CRS (3.3.4). The sparse·dense algorithm was chosen, because it is the only Eigen matrix multiplication algorithm, that supports multi-threading. The CRS·CRS algorithm only slightly performs much better than even the sparse·sparse algorithm,

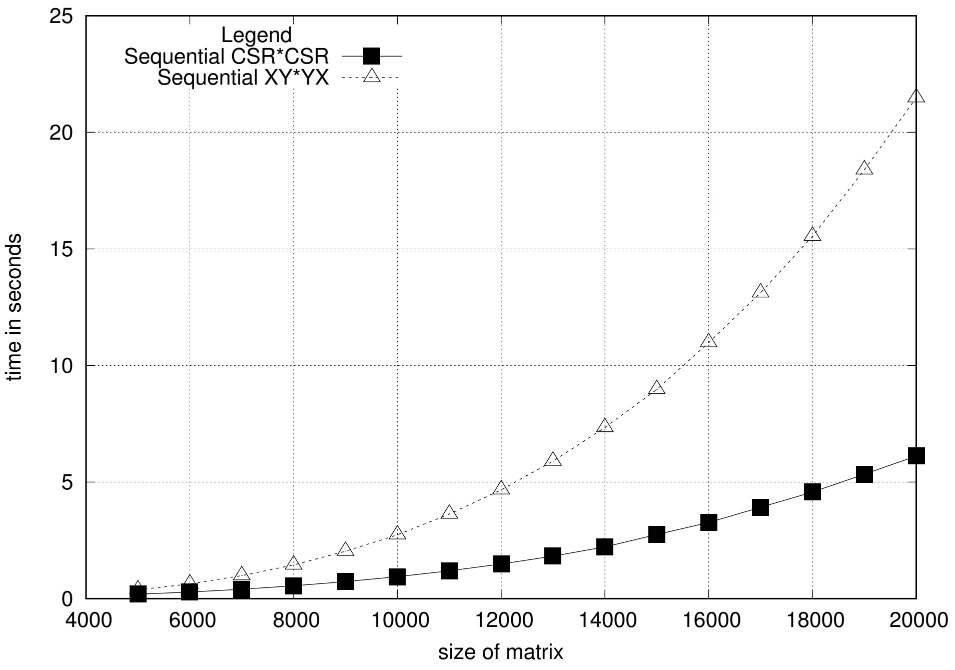

Figure 4.1: Comparison of my sequential algorithms depending on the size of the matrix with density of 1%

which can be explained by the output format, which is dense matrix in my algorithms and sparse·sparse algorithm produces sparse matrix. It also has much better performance over the sparse·dense algorithm, which can be explained by difference in the formats and that the algorithm needs to search the entire dense matrix during the multiplications.

4.3.2 Parallel algorithms

4.3.2.1 My algorithms

The figures 4.4 4.5 repeat the two base benchmark tests with parallel algorithms using 12 threads. Although parallel XY·YX (3.3.7) is considerably slower than the two other algorithms, the rest of them are quite similar in their performance, although parallel CRS·CRS (3.3.5) is really simple algorithm while the parallel YX·XY one is quite complex. This was probably caused by the big overhead that is needed to make YX·XY more efficient.

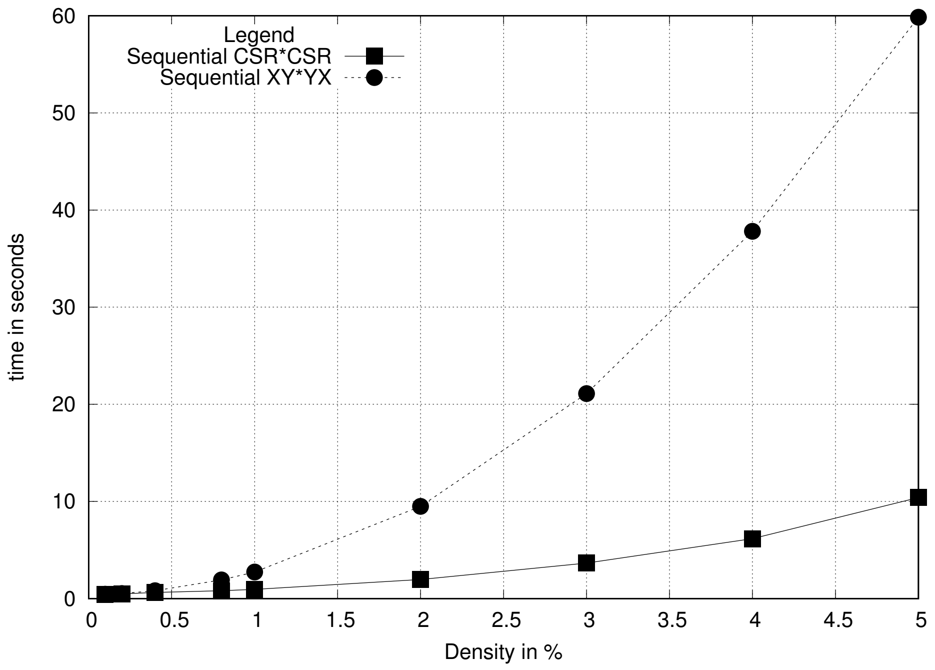

Figure 4.2: Comparison of my sequential algorithms depending on the density of the matrix with fixed size of 10000

4.3.2.2 Eigen comparison

The figure 4.6 compares my two best parallel algorithms with the only Eigen parallel matrix multiplication algorithm in one of the benchmarks, that is equal density of 1% and 12 threads. As this is sparse·dense matrix, it is a bit different, and as such, it can be explained, that my algorithms performed a lot better than the Eigen algorithm.

The figure 4.7 compares the same algorithms, however over a different benchmarks. This time the size of a matrix is fixed at 20000 and density is fixed at 1%, but the number of threads vary. My two best algorithms are similar in performance, while the Eigen is considerably slower, which is probably due to the different format that Eigen library is using and able to parallelize.

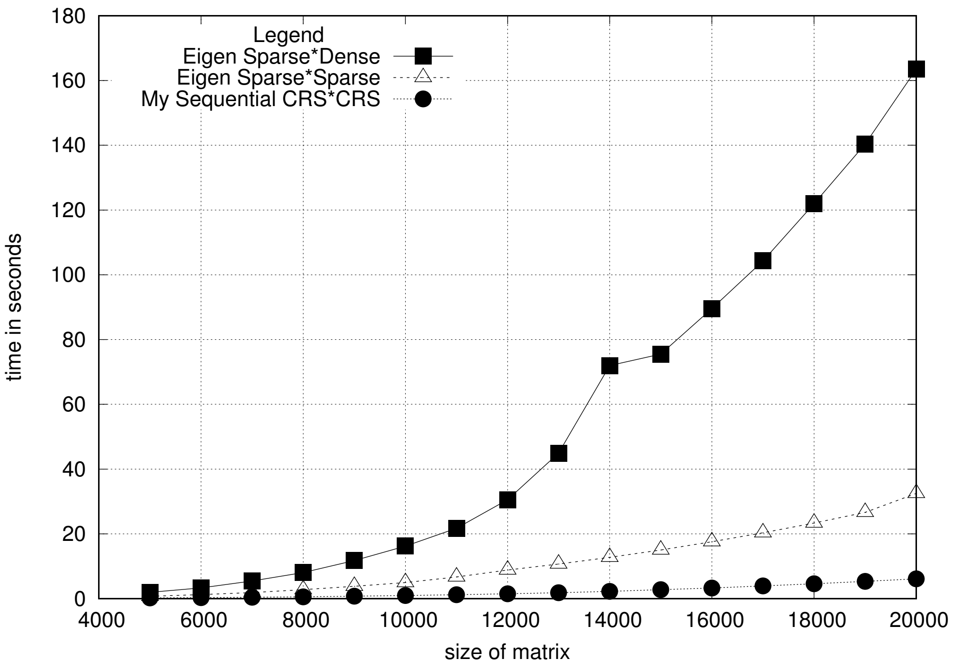

Figure 4.3: Comparison of my best sequential algorithm with Eigen sequential computations with fixed density of 1%

4.3.3 Comparison of speed-up

This subsection will show speed-ups between sequential and parallel versions for all used algorithms. YX·XY sequential algorithm is really slow and wouldn't properly represent the speed-up and the parallel algorithm is quite similar to the XY·YX, therefore I will compare the YX·XY parallel algorithm with XY·YX sequential algorithm. All speed-ups are calculated as $speed\_up = sequential\_time/parallel\_time$. The matrices used are of size 18000 and density 1%. The parallel time is from a test result, that used 12 threads. All times are in seconds.

	YX·XY	XY·YX	CRS·CRS	Eigen Sp·De
seq time	15.542777	15.542777	4.566569	121.982469
par time	3.885607	13.179994	1.984337	31.408147
speed-up	4.00	1.18	2.30	3.88

Although CRS·CRS is the fastest, the speed-up is only 2.30, which is second lowest. The XY·YX offers very poor increase in performance (speed-up is only 1.18). At lower sizes of matrices the time of parallel algorithm was even slower than a sequential one, which is probably due to the need to merge the results after the multiplication. For example at size 10000, the parallel XY·YX

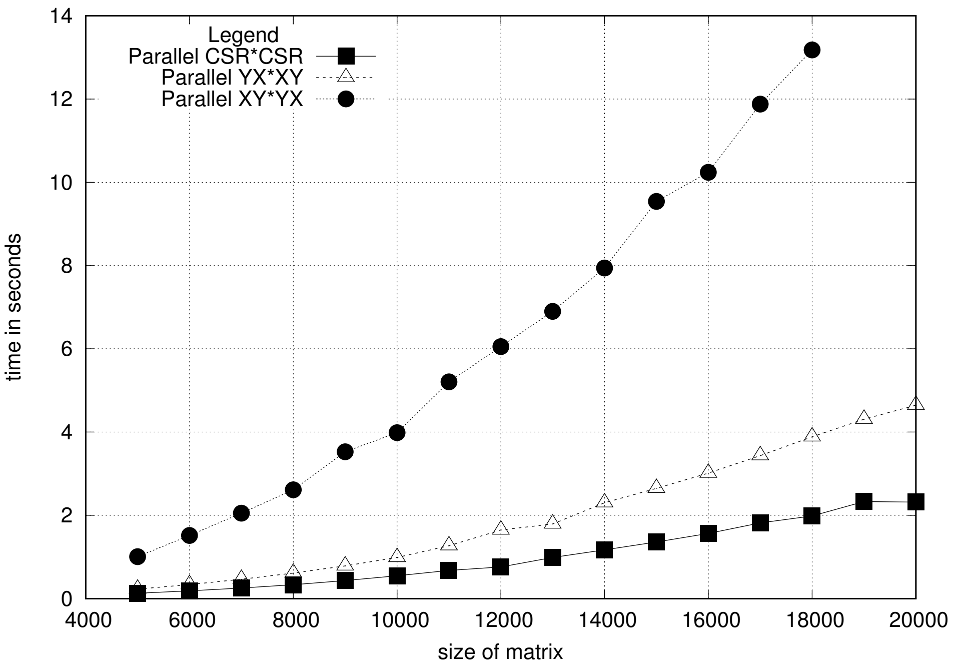

Figure 4.4: Comparison of my parallel algorithms depending on the size of the matrix with density of 1%

took 3.98 seconds, while the sequential took only 2.74 seconds. YX·XY and Eigen both offer very good increase in performance (speed-up is 4.00 and 3.88 respectively), however Eigen algorithm is the slowest one of them all, which was probably due to different result storage format.

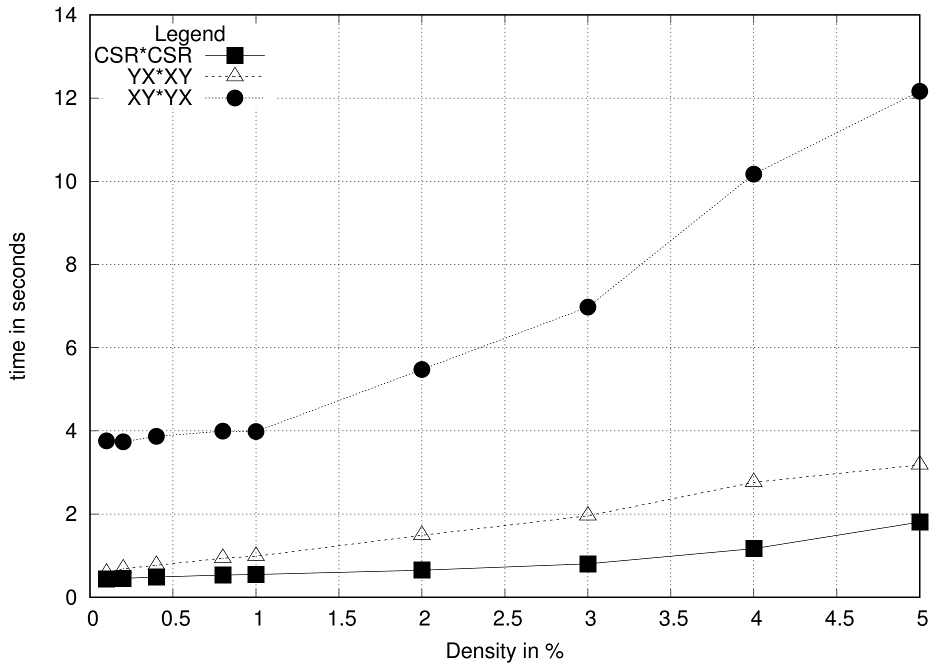

Figure 4.5: Comparison of my parallel algorithms depending on the density of the matrix with fixed size of 10000

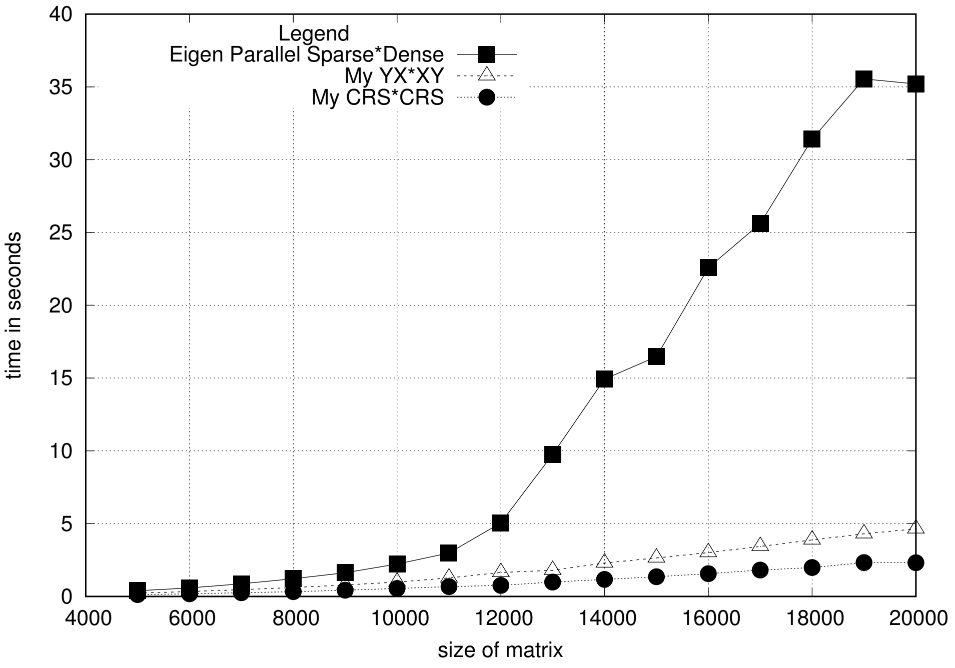

Figure 4.6: Comparison of my parallel algorithms with Eigen parallel computation with fixed density of 1%

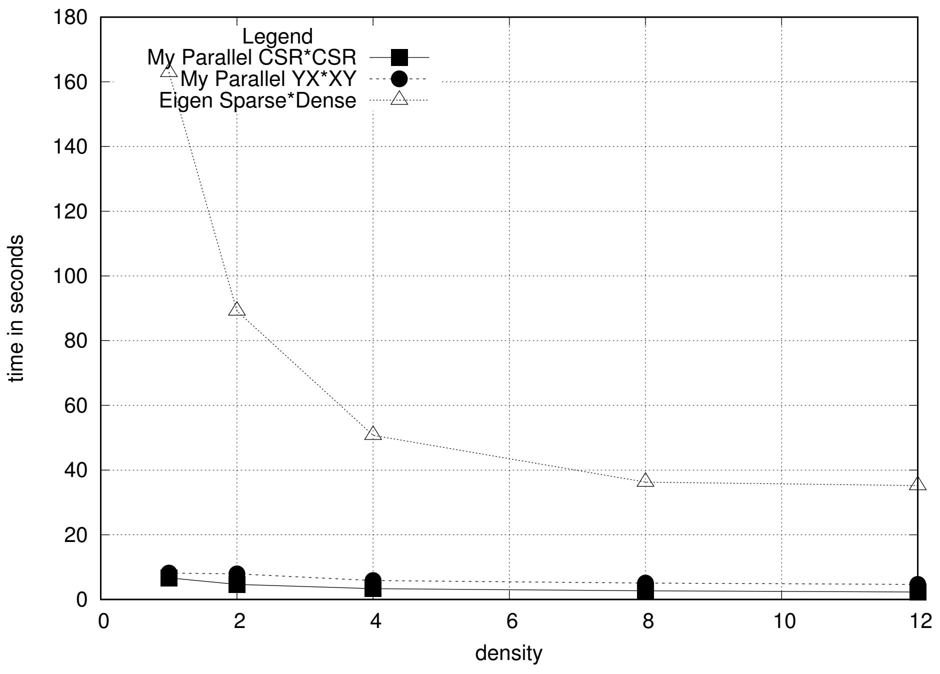

Figure 4.7: Comparison of my best parallel algorithms with Eigen parallel computation with fixed size of 20000 and fixed density of 1%

Conclusion

The goal of this thesis was to compare different storage formats of matrices and algorithms of their multiplication and their effectiveness with parallel computing. In this thesis I implemented dense, CRS, XY and YX formats and I implemented XY·YX, YX·XY and CSR·CSR as both sequential and parallel algorithms.

I analysed these implemented algorithms and their time complexities and compared them with each other and with Eigen library.

The XY·YX sequential algorithm seriously underperformed compared to CRS·CRS sequential algorithm. To my surprise sequential CRS·CRS algorithm outperformed even the Eigen sparse multiplication. The parallel algorithm of CRS·CRS also performed a little bit better than parallel YX·XY. And again to my surprise both of these performed vastly better, than the only parallel implementation that Eigen library offers. My expectation was that the YX·XY will be better at higher densities, due to better handling of the input and result matrices. However this was simply not true and is probably due to big overhead, that YX·XY parallel algorithm had.

YX·XY algorithm achieved a speed-up of an impressive 400%. However the simplicity of CRS·CRS algorithm won against any other type of algorithm used. CRS·CRS dominated all tests for sequential and parallel algorithms. The only case where anything came close was parallel multiplication, where at the same conditions CRS·CRS achieved 1.98 seconds and YX·XY achieved 3.89 seconds.

Further work

Further work could include more input storage formats, some specific types of matrices and specific algorithms for these types of matrices.

Bibliography

[1] Wikipedia, the free encyclopedia: Matrix Multiplication. 2019. Dostupné z: https://en.wikipedia.org/wiki/Matrix_multiplication#/media/File:Matrix_multiplication_diagram_2.svg

[2] Wikipedia, the free encyclopedia: Diagram of a generic dual-core processor. 2019. Dostupné z: https://en.wikipedia.org/wiki/Multi-core_processor#/media/File:Dual_Core_Generic.svg

[3] Dongarra, J.: Compressed Row Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node91.html

[4] Dongarra, J.: Compressed Column Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node92.html

[5] Dongarra, J.: Block Compressed Row Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node93.html

[6] Dongarra, J.: Compressed Diagonal Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node94.html

[7] Dongarra, J.: Jagged Diagonal Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node95.html

[8] Dongarra, J.: Skyline Storage. Nov 1995. Dostupné z: http://netlib.org/linalg/html_templates/node96.html

[9] Šimeček, I.: Sparse Matrix Computations with Quadtrees. In *Proceedings of Workshop 2008*, ročník 1, Prague: CTU, 2008, ISBN 978-80-01-04016-4, s. –.

[10] Dostupné z: https://math.nist.gov/mcsd/savg/tutorial/ansys/FEM/

[11] Sipser, M.: Introduction to the Theory of Computation. Course Technology Inc, 2006, ISBN 0-619-21764-2.

[12] Adve, S. V.; Adve, V. S.; Agha, G.; aj.: *Parallel Computing Research at Illinois.*

[13] Research, J. P.: *The Balance of Power in Gaming.*

[14] Ankalkoti, P.: *A Relative Study on Bitcoin Mining.* Dostupné z: `https://www.researchgate.net/publication/318850089_A_Relative_Study_on_Bitcoin_Mining`

[15] Dostupné z: `https://boinc.berkeley.edu/trac/wiki/VolunteerComputing`

[16] Unger, S. H.: *Hazards, critical races, and metastability.* Dostupné z: `https://ieeexplore.ieee.org/document/391185?tp=&arnumber=391185`

[17] Thompson, M.: Aug 2011. Dostupné z: `https://dzone.com/articles/false-sharing`

[18] Developers, B.: . Dostupné z: `https://theboostcpplibraries.com/boost.thread-thread-local-storage`

[19] Dongarra, J.: 1995. Dostupné z: `http://netlib.org/linalg/html_templates/node91.html`

[20] MatrixMarket: 2019. Dostupné z: `http://math.nist.gov/MatrixMarket/mmio-c.html`

[21] Developers, O.: 2019. Dostupné z: `https://www.openmp.org/`

[22] Developers, E.: 2019. Dostupné z: `http://eigen.tuxfamily.org/index.php?title=Main_Page`

[23] Wolfram: 2019. Dostupné z: `http://www.wolfram.com/mathematica`

[24] Developers, D.: 2019. Dostupné z: `http://www.stack.nl/~dimitri/doxygen/index.html`

[25] Developers, G.: 2019. Dostupné z: `http://www.graphviz.org/Home.php`

[26] Developers, G.: 2019. Dostupné z: `http://www.gnuplot.info/`

[27] Developers, G.: 2019. Dostupné z: `http://git-scm.com/`

[28] Developers, B.: 2019. Dostupné z: `http://bitbucket.org/`

[29] Developers, V.: 2019. Dostupné z: `http://valgrind.org/`

Appendix A

List of used abbreviations

XY	Coordinate format column major
YX	Coordinate format row major
CRS	Compressed row storage
RAM	Random-access memory
CPU	Central processing unit
GPU	Graphics processing unit

Appendix B

Contents of enclosed CD

`readme.txt`	Brief description of contents of this CD
`src`	Folder for all source code
`src/mult`	Source code for the main program for multiplication
`src/gen`	Source code for the matrix generator
`src/eigen`	Source code for the Eigen multiplication
`src/thesis`	Source code of this thesis in LaTeX
`text`	Folder for this thesis
`text/thesis.pdf`	This thesis in PDF format