INSTITUTE FOR ANALYSIS AND SCIENTIFIC COMPUTING

NUMERICAL COMPUTATION - PROJECT:
STRASSEN ALGORITHM

Group 2

Members:
Christian GOLLMANN, 01435044
Peter HOLZNER, 01426733

Submission: January 30, 2020

Contents

1 The Strassen algorithm

The standard algorithm to multiply two matrices $A, B \in \mathbb{R}^{nxn}$ has complexity $\mathcal{O}\left(n^3\right)$. In 1969, Volker Strassen published the Strassen-Algorithm that reduced this complexity (i.e. the number of arithmetic operations) to $\mathcal{O}\left(n^{log_2(7)}\right)$.
The Strassen matrix multiplication is a divide and conquer algorithm which means the matrix multiplication is recursively split up into smaller matrix multiplications until a size is reached, where the multiplications are solved directly, i.e. by scalar multiplication (1x1 matrices) or with the standard matrix multiplication algorithm.

Let the matrices A, B and C be of dimension $n = 2^m$. Then they can be decomposed into 4 blocks of size 2^{m-1}

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22}, \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22}, \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22}, \end{pmatrix} \tag{1}$$

In block matrix notation ,the standard multiplication $AB = C$ then reads as

$$AB = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} \tag{2}$$

Volker Strassen noticed that another, equivalent, representation can be obtained in terms of the matrices

$$M_1 := (A_{11} + A_{22})(B_{11} + B_{22}) \tag{3}$$
$$M_2 := (A_{21} + A_{22})B_{11} \tag{4}$$
$$M_3 := A_{11}(B_{12} - B_{22}) \tag{5}$$
$$M_4 := A_{22}(B_{21} - B_{11}) \tag{6}$$
$$M_5 := (A_{11} + A_{12})B_{22} \tag{7}$$
$$M_6 := (A_{21} - A_{11})(B_{11} + B_{12}) \tag{8}$$
$$M_7 := (A_{12} - A_{22})(B_{21} + B_{22}) \tag{9}$$

and

$$C_{11} := M_1 + M_4 - M_5 + M_7 \tag{10}$$
$$C_{12} := M_3 + M_5 \tag{11}$$
$$C_{21} := M_2 + M_4 \tag{12}$$
$$C_{22} := M_1 - M_2 + M_3 + M_6 \tag{13}$$

The multiplications in (3) - (9) are done recursively until only 1 x 1 matrices (scalars) have to be multiplied.

2 Checking the correctness of the Strassen algorithm

In order to prove the algorithm's correctness, we will calculate the submatrices $C_{11} - C_{22}$ as the Strassen algorithm tells us to and then show that this yields the same result as the standard Matrix multiplication above.
In the next steps we relate to the matrices $M_1 - M_7$ given in the exercise sheet.

We start with C_{11}

$$\begin{aligned} C_{11} = M_1 + M_4 - M_5 + M_7 = \\ [A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{22}B_{22}] + [A_{22}B_{21} - A_{22}B_{11}] - \\ [A_{11}B_{22} + A_{12}B_{22}] + [A_{12}B_{21} - A_{22}B_{21} + A_{12}B_{22} - A_{22}B_{22}] = \\ A_{11}B_{11} + A_{12}B_{21} \end{aligned} \tag{14}$$

One sees that most of the terms cancel out and leave one with the definition of C_{11} derived from the standard multiplication. We proceed by checking the three remaining submatrices.

$$C_{12} = M_3 + M_5 = [A_{11}B_{12} - A_{11}B_{22}] + [A_{11}B_{22} + A_{12}B_{22}] = A_{11}B_{12} + A_{12}B_{22} \tag{15}$$

$$C_{21} = M_2 + M_4 = [A_{21}B_{11} + A_{22}B_{11}] + [A_{22}B_{21} - A_{22}B_{11}] = A_{21}B_{11} + A_{22}B_{21} \tag{16}$$

$$\begin{aligned} C_{22} = M_1 - M_2 + M_3 + M_6 = \\ [A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{22}B_{22}] - [A_{21}B_{11} + A_{22}B_{11}] + \\ [A_{11}B_{12} - A_{11}B_{22}] + [A_{21}B_{11} - A_{11}B_{11} + A_{21}B_{12} - A_{11}B_{12}] = \\ A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{17}$$

3 Proving the algorithm's complexity

There are actually two ways to derive the Strassen algorithm's complexity. One is to apply the so called Master theorem[1] for recursive algorithms and the other is to resolve the algorithm manually. We will start with the Master theorem.

3.1 Master Theorem

The Master theorem can be applied to recursive algorithms of the form

$$T(n) = aT(n/b) + f(n) \tag{18}$$

The theorem then distinguishes between three cases:

1. There exists a constant $\epsilon > 0$ so that $f(n) = \mathcal{O}\left(n^{log_b a - \epsilon}\right)$. Then $T(n) = \mathcal{O}\left(n^{log_b a}\right)$.
2. $f(n) = \mathcal{O}\left(n^{log_b a}\right)$, then $T(n) = \mathcal{O}\left(n^{log_b a} log(n)\right)$.
3. There exist a constant $\epsilon > 0$ so that $f(n) = \mathcal{O}\left(n^{log_b a + \epsilon}\right)$ and a constant $c < 1$ so that $af(n/b) \leq cf(n)$. Then $T(n) = \mathcal{O}\left(f(n)\right)$.

So we first have to determine a, b and $f(n)$ for our case. For the matrices M_1 - M_7 the Strassen algorithm relies on 7 matrix multiplications in which it calls itself recursively. Therefore $a = 7$. In those recursive calls, the matrices' dimensions have halved, so $b = 2$. In order to assemble the matrices M_1-M_7 and C_{11}-C_{22} we need 18 additions and substractions with matrices of dimension $\frac{n}{2}$. Therefore

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \tag{19}$$

and as

$$f(n) = \mathcal{O}\left(n^2\right) = \mathcal{O}\left(n^{log_2(7) - \epsilon}\right) \text{ with } \epsilon > 0 \tag{20}$$

we apply case 1

$$T(n) = \mathcal{O}\left(n^{log_a(b)}\right) = \mathcal{O}\left(n^{log_2(7)}\right) \blacksquare \tag{21}$$

[1] Introduction to algorithms, Cormen, Thomas H and Leiserson, Charles E and Rivest, Ronald L and Stein, Clifford, 2009, MIT press

3.2 Resolving the algorithm manually

We start by remembering that the to be multiplied matrices' dimensions are of $n = 2^m$. As seen in (19), there holds

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$$

We see that the algorithm calls itself recursively for dimension $n/2$. Substituting this recursive call with $T(n/2)$ leads to a recursive call for dimension $n/4$ and so on. By making use of this pattern and some clever rearrangement of terms, we derive the following

$$\begin{aligned}
T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \\
&= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2}n^2 \\
&= 7\left(7 \cdot T\left(\frac{n}{4}\right) + 18\left(\frac{n}{4}\right)^2\right) + \frac{9}{2}n^2 \\
&= 7\left(7 \cdot T\left(\frac{n}{4}\right) + \frac{9}{2}\left(\frac{n}{2}\right)^2\right) + \frac{9}{2}n^2 \\
&= 7\left(7\left(7 \cdot T\left(\frac{n}{8}\right) + \frac{9}{2}\left(\frac{n}{4}\right)^2\right) + \frac{9}{2}\left(\frac{n}{2}\right)^2\right) + \frac{9}{2}n^2 \\
&= ... \\
&= 7^m \cdot T\left(\frac{n}{2^m}\right) + \frac{9}{2}n^2\left(1 + 7\left(\frac{1}{2}\right)^2 + 7^2\left(\frac{1}{4}\right)^2 + ... + 7^{m-1}\left(\frac{1}{2^{m-1}}\right)^2\right) \\
&= 7^m \cdot T(1) + \frac{9}{2}n^2 \sum_{i=0}^{m-1}\left(\frac{7}{4}\right)^i \\
&= 7^m \cdot 1 + \frac{9}{2}n^2\left(\frac{1 - \left(\frac{7}{4}\right)^m}{1 - \frac{7}{4}}\right) \\
&= 7^m - 6n^2\left(1 - \left(\frac{7}{4}\right)^m\right) \\
&= 7^m - 6n^2 + 7^m \cdot 6\frac{n^2}{4^m} \\
&= 7^m - 6n^2 + 7^m \cdot 6\frac{n^2}{(2^m)^2} \\
&= 7^m - 6n^2 + 7^m \cdot 6\frac{n^2}{n^2} \\
&= 7^m - 6n^2 + 6 \cdot 7^m = 7 \cdot 7^m - 6n^2 \\
&= 7 \cdot \left(2^{log_2 7}\right)^m - 6n^2 \\
&= 7 \cdot (2^m)^{log_2 7} - 6n^2 \\
&= 7 \cdot n^{log_2 7} - 6n^2 \\
&= \mathcal{O}\left(n^{log_2 7}\right) \blacksquare
\end{aligned} \tag{22}$$

4 Implementation - the memory challenge

An important task for us was to implement the algorithm in a way so that it doesn't allocate any additional memory apart from a workspace we give it in the beginning. This workspace is denoted as a n x n matrix. In the following we want to explain how we achieve that.

We take advantage of the fact, that only one of the seven matrices $M_1 - M_7$ has to be present at each time. This matrix can then be added to the respective submatrices of the result, $C_{11} - C_{22}$. After that, memory is free to be overwritten and can be used to calculate the next M-matrix.

Let's walk through this process using the example of computing M_1 as it is visualised in figure 1. First we calculate the sum of A_{11} and A_{22} which gets stored in W_{11}. Then we do the same with $B_{11} + B_{22}$ and store it in W_{12}. For the multiplication we call Strassen recursively and give it W_{21} as workspace. The result of this calculation, M_1 then gets added to C_{11} and C_{22}. After that, the whole workspace is freed and we continue with the calculation of M_2 which overwrites it.

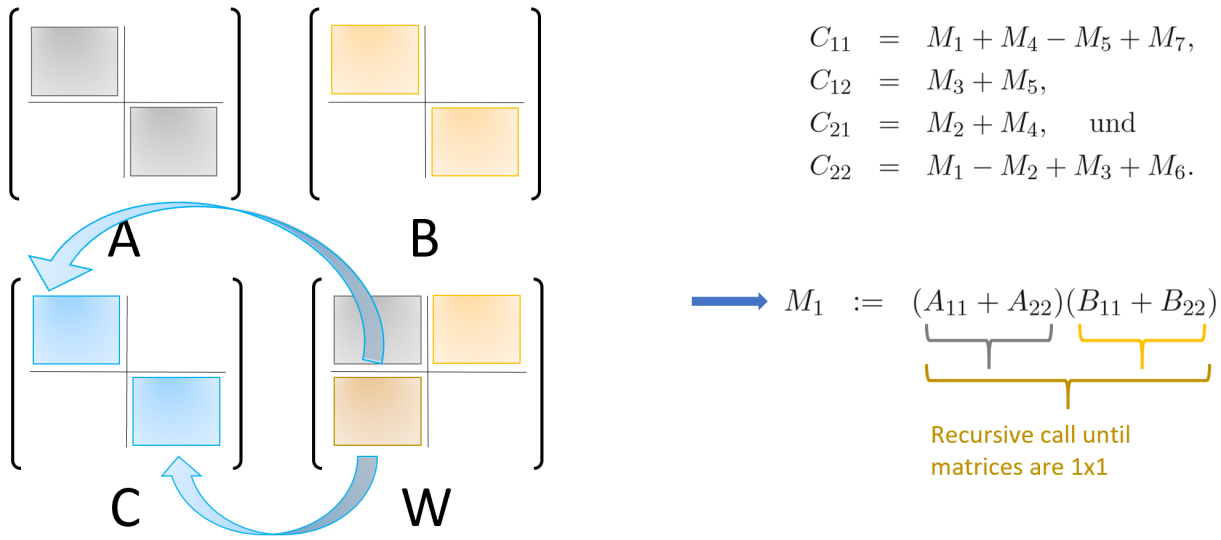

Figure 1: Process of calculating M_1 using the workspace

5 Testing our implementation

First we want to test our implementation on two randomly generated 4x4 matrices. In figure 2, one can see a console snapshot of the two matrices A and B. The results of the standard and Strassen matrix multiplication are shown in figure 3, together with their respective execution times. In order to check the standard algorithm's correctness, we initialized the matrices A and B in Matlab and calculated their product $C = AB$. The result of this calculation can be seen in figure 4.

```
Starting test of Strassen algorithm

Seed for generating the matrices: 67

-----A-----
Matrix:
3.89335 3.23111 6.80739 6.02504
7.7532  2.11856 8.44331 5.50285
4.44136 2.77479 4.77596 6.63104
4.85517 5.40882 2.06974 4.34389

-----B-----
Matrix:
9.46635 7.89245 2.89962 9.65206
8.64676 7.49263 3.74251 3.91106
2.39225 3.68031 7.78365 3.23218
9.76965 9.15534 7.26638 3.01705
```

Figure 2: Generating A and B

```
_____TEST START_____
Testing the Strassen algorithm vs standard (naive) Matrix-Matrix-Multiplication

-----CRef-----(standard (naive) Matrix-Matrix-Multiplication)
Matrix:
139.942 135.152 120.148 90.3965
165.673 158.52  136.116 127.013
142.245 134.13  108.621 89.1636
140.119 126.233 81.9952 87.8121

Standard took 0.004 millisec


-----CStr-----(Strassen algorithm)Matrix:
139.942 135.152 120.148 90.3965
165.673 158.52  136.116 127.013
142.245 134.13  108.621 89.1636
140.119 126.233 81.9952 87.8121

Stassen took 0.007 millisec
```

Figure 3: Result Standard and Strassen algorithm

```
Command Window
>> StrassenVergleich

A =

    3.8933    3.2311    6.8074    6.0250
    7.7532    2.1186    8.4433    5.5028
    4.4414    2.7748    4.7760    6.6310
    4.8552    5.4088    2.0697    4.3439


B =

    9.4664    7.8925    2.8996    9.6521
    8.6468    7.4926    3.7425    3.9111
    2.3923    3.6803    7.7836    3.2322
    9.7697    9.1553    7.2664    3.0170


C =

  139.9420  135.1522  120.1483   90.3965
  165.6726  158.5198  136.1156  127.0128
  142.2446  134.1301  108.6210   89.1636
  140.1191  126.2325   81.9952   87.8121

fx >>
```

Figure 4: Calculating $C = AB$ in Matlab

We followed this procedure several times with different n and different seeds for the random generation of the matrices A and B. Every time, the results of our standard algorithm and Matlab matched. We therefore conclude that our implementation of the standard algorithm is correct.

Knowing that our implementation of the standard matrix-matrix multiplication is correct, we want to check if the Strassen algorithm yields the same result as well. First, we can see in figure 3 that for dimension $n = 4$, the Strassen multiplication gives the correct result.
Next, we do the same comparison for input matrices of dimension $n = 8$ as shown in figure 5 and figure 6.

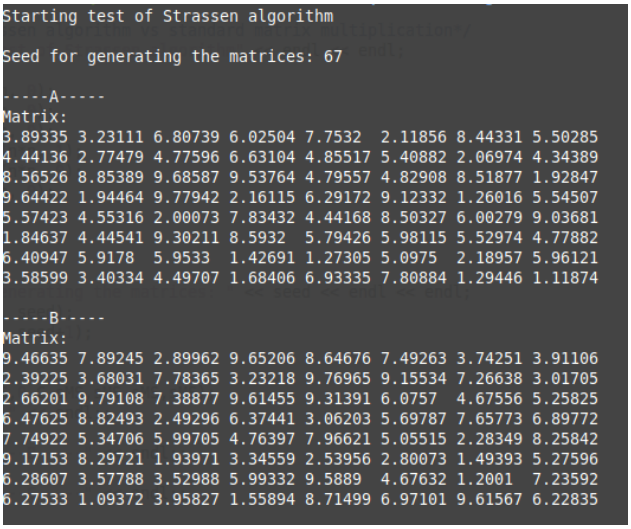

Figure 5: Generating A and B for $n = 8$

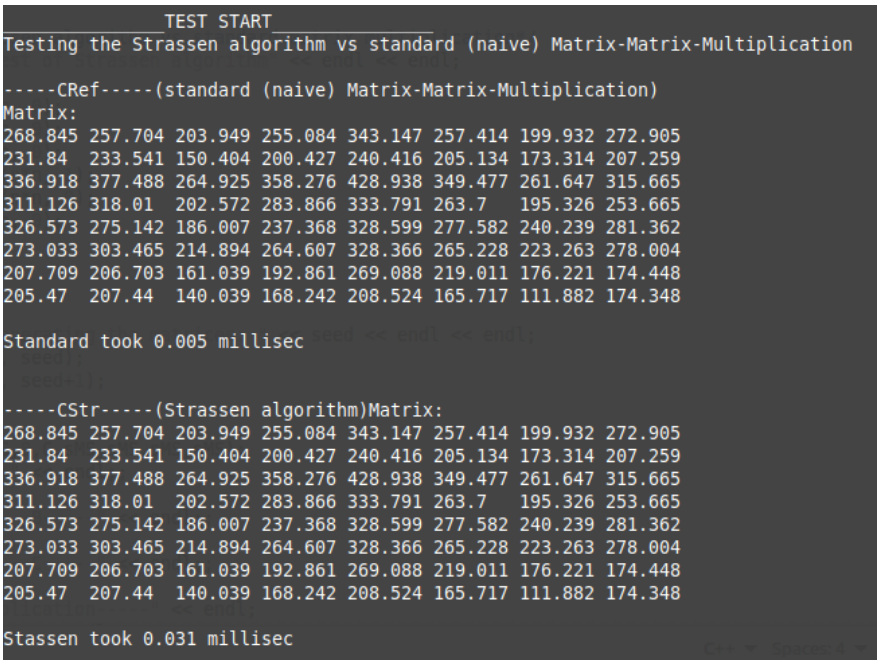

Figure 6: Result Standard and Strassen algorithm for $n = 8$

Since the standard and Strassen matrix match again, we conclude that our implementation of the Strassen algorithm works properly.

6 Comparison of runtimes

We conclude by comparing the actual CPU execution times of the Standard and Strassen algorithm. We executed multiplications for dimensions $n = 2^m$, where $m = 6...12$, and collected the execution times[2]. They are plotted in figure 7. Additionally, we tried hybrid implementations, which use the Strassen algorithm until a minimal size of the submatrices is reached. At this point, the recursion is resolved by directly computing the matrix multiplication via the standard algorithm.

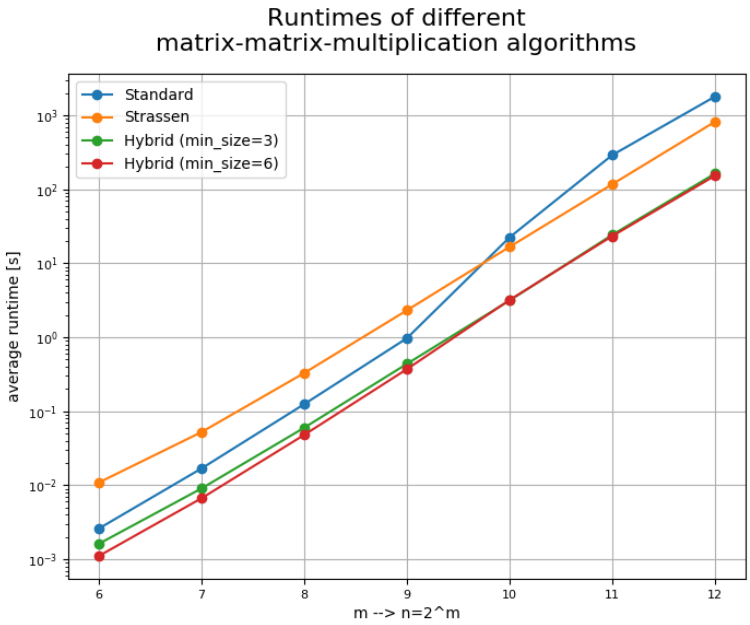

Figure 7: Comparison of runtimes. *Strassen* resolves the recursion once 1×1 matrices (scalars) are reached. The *Hybrid* methods resolve by standard matrix multiplication once $(min\_size) \times (min\_size)$ matrices are reached.

It can be seen that the Strassen algorithm only displays its advantages after $m = 9$. Why is that?

If we apply the same procedure as in equation 22 to the standard algorithm, we end up with

$$T_{Standard}(n) = 8^m - n^2 + n^3 \tag{23}$$

Let's find the matrix dimension n for when the Standard's and Strassen algorithm's runtimes are

[2]Ideally, one would repeat these multiplications a number of times, e.g. 10 times per m, and collect the averages. We didn't do this, because it would have been impractical for our purposes for $m >= 10$.

equal.

$$\begin{aligned} T_{Standard}(n) &= T_{Strassen}(n) \\ 8^m - n^2 + n^3 &= 7 \cdot 7^m - 6n^2 \\ \left(2^3\right)^m - n^2 + n^3 &= 7 \cdot \left(2^{log_2 7}\right)^m - 6n^2 \\ n^3 - n^2 + n^3 &= 7 \cdot n^{log_2 7} - 6n^2 \\ 2n^3 - 7n^{log_2 7} + 5n^2 &= 0 \\ \Rightarrow n &\approx 654 \\ \Rightarrow m = log_2(n) &= log_2(654) \approx 9.35 > 9 \end{aligned} \tag{24}$$

Our interpretation of this behaviour is the following. For smaller matrix sizes ($m \leq 9$), the Strassen algorithm actually needs more FLOPs to compute compared to the standard algorithm. The process of constructing the matrices $M1 - M7$ actually costs more than what is later gained by reducing the number of matrix multiplications to be done by one. Additionally, the Strassen recursion algorithm might come with additional overhead that is hard to quantify (partitioning of the matrices, etc.). Especially, when one has to take into account modern CPU architectures, instruction set extensions and shared/distributed memory (parallelization). We did not look further into it as it is beyond the scope of this project.
As can be seen in figure 7, the best results were achieved when combining both methods (*Hybrid*). This approach seems to combine the strengths of both approaches as it outperforms both the *standard* and pure *Strassen* implementations across the board. In other words, subdividing the matrices down to 1×1 matrices does not pay off and straight forward matrix multiplication is to be preferred. However, even for smaller matrices, e.g. $m = 6$, subdividing them a couple of times, but not down to 1×1 matrices, improves the runtime even compared to the standard matrix-matrix multiplication. In the following, we give an example to justify our assumption.
We compare the cost of the standard matrix-matrix multiplication with the cost of a hybrid implementation that only subdivides once. Mathematically, it reads as the following

$$\begin{aligned} T_{Standard}(n) &> T_{Hybrid}(n) \\ 2n^3 - n^2 &> 7 \cdot T_{Standard}\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \\ 2n^3 - n^2 &> 7 \cdot 2\left(\frac{n}{2}\right)^3 - 7\left(\frac{n}{2}\right)^2 + 18\left(\frac{n}{2}\right)^2 \\ &\ldots \\ n &> 7 \end{aligned} \tag{25}$$

The above shows, that for $n > 7 \Leftrightarrow m \geq 3$ the above mentioned hybrid approach is faster than pure standard matrix-matrix multiplication.