1 Introduction

In the first section, we will give mathematical reasoning of Strassen's algorithm for matrix multiplication. First a naïve method for matrix multiplication is explained, and then it is extended to more advanced Strassen's method. In the following section a description of programming language and framework is given, with explanation of the algorithm's implementation. Last two sections present the result and explain some interesting points with regard to the results.

2 Strassen's algorithm

In this section we will describe the mathematical background of naïve matrix multiplication and then we will introduce Strassen's algorithm.

2.1 Naïve matrix multiplication

Matrix operations are ones of the backbones of scientific calculations and they are in the background of a large number of applications. For example matrices are used in programs for data analysis, sound compression and games for creating 3D scenes. For this reason it is important that these methods are efficient. Let us first remind us of the regular method for matrix multiplication.

Let $A$ and $B$ be matrices with dimensions $n \times n$. From rudimentary knowledge of linear algebra, we know that matrix multiplication is defined as in (1).

$$[AB]_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}. \quad (1)$$

In pseudo-code, this algorithm can be expressed as is shown in Fig. 1.

![Naive matrix multiplication algorithm](image)

Regarding memory space, algorithm requires space complexity of $O(n^3)$. Also, as shown in Fig. 1, algorithm does $n^3$ multiplications and $n^3 \times (n-1)$ additions, which asymptotically leads to time complexity of $O(n^3)$.

2.2 Strassen's algorithm for matrix multiplication

This algorithm is defined only for square matrices. But in practice, this proves as no restriction, because implementation can be generalized for use with all kinds of matrices, by means of expanding matrices with zeros. Adding rows and columns solely composed of zeros does not affect addition and multiplication of matrices.

Primary idea behind the algorithm is to recursively divide matrices in blocks and then to multiply those sub-matrices (divide-and-conquer). Doing this, we reduce multiplications but we have more additions.
additions instead of multiplications is desirable because additions take less computational time than multiplications, since multiplication is a more complex operation (in terms of computational resources) than addition [1].

Let $A$, $B$ and $C$ be matrices with dimensions $n \times n$, where $C = A \times B$ and let $n = 2^k$ for some $k$ (and if this is not the case, we expand matrices with rows and columns of zeros, while $n$ becomes exponent of 2). First step is dividing matrices into sub-matrices with dimensions $(n/2) \times (n/2)$, as in (2), (3) and (4).

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \qquad (2)$$

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \qquad (3)$$

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} \qquad (4)$$

Then, we can calculate matrix $C$ as in (5).

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

$$ (5)$$

At first, this approach does not give any advantage, since we have not reduced the number of multiplications. There are still 8 multiplications needed for the calculation of $C$. Hence, we use Strassen’s idea (for details, see [2]) for matrix multiplication, which defines 7 new matrices, as shown in (6).

$P_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$

$P_2 = (A_{2,1} + A_{2,2})B_{1,1}$

$P_3 = A_{1,1}(B_{1,2} - B_{2,2})$

$P_4 = A_{2,2}(B_{2,1} - B_{1,1})$

$P_5 = (A_{1,1} + A_{1,2})B_{2,2}$

$P_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$

$P_7 = (A_{2,2} - A_{1,2})(B_{2,1} + B_{2,2})$

$$ (6)$$

Now we can use these $P$ matrices to express $C$ sub-matrices, with rules stated as in (7).

$C_{1,1} = P_1 + P_4 - P_5 + P_7$

$C_{1,2} = P_3 + P_5$

$C_{2,1} = P_5 + P_4$

$C_{2,2} = P_1 - P_2 + P_3 + P_6$

$$ (7)$$

At first, this looks more complex than the original method. But, with this, we reduced the number of multiplications to 7. Also, we can easily prove the validity of the method by expanding members of matrix $C$ and we should get the same results with both methods.

This process is now recursively repeated until matrices degenerate into numbers. Or, which is more efficient in practice, until the dimension of matrices reaches a small-enough number (crossover point). From that point on, naïve matrix multiplication is used.

This method has time complexity of $O(n^2 \log^2 n)$. Regarding space complexity of this algorithm, in ideal implementation, it would be $O(n)$ for original matrices and additional $O(n^2)$ for $P$ matrices.

2.3 Related work and our contribution

Povezani radovi i naš doprinos

There is an efficient and portable serial implementation of Strassen’s algorithm called DGEFMM provided in ATLAS [3]. Most implementations of Strassen’s algorithm were designed for distributed memory architectures and are compared to matrix multiplication algorithms in standard numerical packages such as BLAS, LAPACK, ScALAPACK (for details, see [4, 5, 6]).

Song, Dongarra and Moore gave several experiments with Strassen’s algorithm with NetSolve [5] using task-parallel and data-parallel approach based on MPI paradigm.

We give an implementation of Strassen’s algorithm in C# where we establish a task-parallel approach on multithreaded programming environment. The idea is to harvest computing power of personal computers with multi-core processors. We give test results experimented on several computer configurations.

Multi-core processors are common now, and although they are especially efficient for multi-thread applications, there is much place for improvement. Our goal was to investigate advantages of this kind of processors, in the context of matrix multiplication. Basic principles of multi-core processors and parallel programming for them can be found in [7] and [8].

This solution cannot compete with optimized solutions, ran on dedicated super-computers or grid-computers (results in [3] shows that faster solutions were available already in 2006, but ran on 2 × 2 grid computers). But it can show advantages of using parallel approach, and relation between their efficiency and number of cores in processor.

3 Implementation in C#

Implementacija u C#

This section describes implementation of algorithm with overview of .NET Framework environment and C# programming language.

3.1 .NET Framework and C#

.NET Framework i C#

This implementation is built using .NET Framework 2.0, which consists of two parts: large library of classes commonly used in programming; and virtual machine used for running applications designed for this framework. We decided to build our implementation using this platform for several reasons: ease and simplicity of programming; large community of users and great support; cost (which is zero).

More about the framework can be found in [9].

We selected C# as the programming language, one of the languages native for .NET Framework, combined in integrated development environment (IDE) Microsoft Visual Studio Express. This IDE was chosen again for its
price and availability (it is freeware).

This combination provides a very simple and efficient platform for development of different kind of programs: from small desktop applications through complex business application to web applications. Implementing multithreaded approach is very simple: developer just needs to include provided libraries and classes for threading and then to create threads for particular functions. Of course, advanced properties and methods are also available (e.g. scheduling of threads using mutex).

Although using C++ with optimized compiler would probably result with even better performance (because it avoids .NET Framework level of execution that induces slowness due to interpretation of commands), we chose this environment because it provides efficient mechanisms for easy development of multi-thread applications and it is widespread among the users. Also, advantages and flaws of .NET Framework in scientific research have already been addressed by Gilani [10] and Lutz and Laplante [11].

Alternatively, instead of .NET Framework, as programming platform can be used Java. It shares a similar model (has runtime and libraries with done functions) and syntax [12]. It is also free and has a wide user base.

### 3.1 Implementation of naïve matrix multiplication

**Implementacija naivnog množenja matrica**

Multiplication of two matrices is one of first things that are learned on programming course or school class. Fig. 2 shows implementation of this algorithm in C# (this code, with minimal adjustments in syntax, will work in most of now popular programming languages).

```csharp
short[,] Multiply(int n, short[,] m1, short[,] m2) {
    short[,] z = new short[n, n];
    for (short i = 0; i < n; i++) {
        for (short j = 0; j < n; j++) {
            z[i, j] = 0;
            for (short k = 0; k < n; k++) {
                z[i, j] += (short)(m1[i, k] * m2[k, j]);
            }
        }
    }
    return z;
}
```

*Figure 2 C# implementation of naïve matrix multiplication*

Slika 2. Implementacija naivnog množenja matrica u C#-u

As can be seen from Fig. 2, we are iterating through members of resultant matrix by rows and columns and calculating them using equation (1).

Variables and functions in this implementation are defined in type short, because this type uses only 2 bytes of memory. In regular application, we would use some floating-point type (e.g. double), but this application has demonstrational purpose, so we deliberately used this type. Using decimal variables leads to different problems, like much bigger memory consumption and reduced numerical stability (see [13, 14, 15]).

### 3.3 Implementation of Strassen's algorithm – serial version

**Implementacija Strassenovog algoritma – serijska verzija**

First step in speeding up matrix multiplication is implementing Strassen’s algorithm in C#. For that, we use equations (2) to (7). In pseudo-code, algorithm is defined as shown in Fig. 3. Using this algorithm we should be able to calculate matrix multiplications few times faster than using naïve method, without using any specialized architectures. It can be expected that in the case of small matrices, the naïve method will actually be faster, because Strassen’s method requires time for parting and combining matrices.

```csharp
StrassenMatrixMultiplication(matrix A, matrix B)
if size < cross-over point then
    calculate matrix using naïve method
else
    divide matrix A in sub-matrices A11, A12, A21, A22
    divide matrix B in sub-matrices B11, B12, B21, B22
    P1 is calculated as (A11+A22)*(B11+B22) by recursive StrassenMatrixMultiplication
    P2 is calculated as (A21+A22)*B11 by recursive StrassenMatrixMultiplication
    P3 is calculated as A11*(B21+B22) by recursive StrassenMatrixMultiplication
    P4 is calculated as A22*(B21-B11) by recursive StrassenMatrixMultiplication
    P5 is calculated as (A11+A12)*B22 by recursive StrassenMatrixMultiplication
    P6 is calculated as (A21-A11)*B11+B12) by recursive StrassenMatrixMultiplication
    P7 is calculated as (A12-A22)*(B21+B22) by recursive StrassenMatrixMultiplication
    matrix C and return it as result of function StrassenMatrixMultiplication

Figure 3 Serial Strassen's algorithm in pseudo-code
Slika 3. Serijski Strassenov algoritam u pseudo-kodu
```

As shown in Fig. 3, the idea is to replace every matrix multiplication with recursive call to function for Strassen’s matrix multiplication. This is done while size of matrices is larger than crossover point, as mentioned before. If this parameter is too large, the method will not be the most efficient, because naïve method will be mostly used. And if this number is too small, processing time will be too much used on dividing and combining matrices. So the crossover point is best to be determined by an experiment as it depends on the size of matrices.

Other necessary functions are those for dividing matrices to quarters, for adding and subtracting them and these for combining quarters into whole matrices. All these functions are simple to implement, using two for loops and basic arithmetical operations.

### 3.4 Implementation of Strassen’s algorithm – parallel version

**Implementacija Strassenovog algoritma – paralelna verzija**

Modern computer architectures, even those common for home usage, promotes multi-core CPU-s, so it would be more efficient to exploit this fact. Also, Strassen’s algorithm is very suitable for parallel implementation,
because of the fact that, after division of matrices and calculation of \( P \) matrices, they are independent between them (as can be seen from (2) to (7)). That means that each primary \( P \) matrix multiplication (those \( P \) matrices that are made after division of original matrices) can later be recursively calculated using separate Strassen's algorithm, each on their own thread, as shown in Fig. 4.

If we look deeper at pseudo-code of serial implementation of Strassen's algorithm, we can see that the algorithm goes in depth (as in depth-first search, DFS [16]). Parallel implementation also goes in depth, but only after the first step, and then there are seven simultaneous branches of algorithm.

.NET Framework already has a Thread class available, and its usage is quite simple. It consists of creating seven new Thread objects, each with unique function that will be executed on thread and starting them. These functions are just calls for regular, serial Strassen's calculations of \( P \) matrices.

Advantage of this is higher speed of calculations on modern, multi-core processors, but on older, single-core processors, this will result in lower speed, because the system will be unnecessarily encumbered with handling threads. Also, it will result in larger memory requirements because of the approach .NET Framework has to memory and variables. Instead of forcing programmer to do the allocating, initializing, etc. of variables, .NET Framework utilizes a mechanism called Garbage Collector (GC) that takes charge of tracking memory usage and knowing when memory will be released [9]. This greatly simplifies programming, but unfortunately, as this mechanism is not perfect, it induces some extra memory consumption. This is not so important in normal desktop application, but in applications that utilize processor very much, Garbage Collector cannot manage to release all memory. .NET Framework of course allows this to be manually managed, but this was not the case in our implementation.

4 Results

A sample application, displayed in Fig. 5, was made for testing, with all three methods (naïve, serial and parallel) implemented. Size of matrices, method and crossover point (for Strassen's algorithm) were performed. The experiment started with the appointed size of matrices and the crossover point. The program then created matrices with random members that were used for calculations.

Next it was necessary to select the method and leave the application to finish the calculation. Afterwards, the method, size and COP were changed and the calculation was redone.

The calculations were run on several different configurations, common for home or business applications (Intel and AMD processors, ranging from older single-core, to modern dual- and quad-cores), to show differences and dependency on number of cores in CPU. The test application, after finishing the calculation, also recorded the model of CPU, the amount of RAM (working memory) and the operating system. Due to the restrictions of .NET Framework, testing was done only on Microsoft Windows operating systems.
Fig. 6 shows overall data gathered in testing. The configurations are directly compared; matrices are of size 1024 and the crossover point is 64. At first glance, we can see that Strassen's method is several times faster than naïve method for matrix multiplication, and that parallel method is even faster than serial Strassen's algorithm implementation, which was expected. There are also some other conclusions that can be reached after analysis of data.

Naïve and Strassen's serial implementation cannot use possibilities that multi-core processors offer. The second group of results in Fig. 6 (results obtained using serial Strassen method) has the same mutual relations as the first group (results obtained using naïve method). But, in the third group, relations between results start to differentiate. For example, when using naïve and serial Strassen's method, AMD Athlon 64 3000+ CPU is twice faster than Intel Atom N270 CPU, although the former is an older single-core processor and the latter a modern dual-core CPU. But, when using parallel Strassen's method, Athlon is actually slower, because the system loses time on handling several threads, executing each for short time, giving only an impression of parallel execution. On the other hand, Atom uses its advantage of having two cores to compensate for its slower clock rate. Similar effects can be observed comparing triple- and quad-core processors to dual-core ones, although the difference is not so great.

We must emphasize that Strassen's algorithm's advantage in speed becomes apparent only for larger matrices (e.g. over 1000 rows/columns). This can be seen in Fig. 7; Strassen's method is just slightly faster than naïve for matrices with size of 128, while it is many times faster for matrices size 4096. With larger matrices, this difference should only increase. It then requires much stricter handling of memory and variables.

If we take that the calculation results using naïve method represent raw power of CPU (since it uses only one of the processor's cores, in case it has more), we can see that with parallel implementation, the number of cores has more weight than just the speed of CPU.

In Table 1 we can see that the triple-core AMD processor in this example (size 1024, crossover point 64) is 1.75 times slower than the more powerful, dual-core Intel processor with naïve method of calculation. But with serial Strassen's algorithm, this difference falls down to only 1.22 times slower. And with parallel Strassen's method, the triple-core processor is actually faster, due to the fact that it can simultaneously run more threads than dual core processor.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Calculation time / s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve method</td>
<td>Serial Strassen's</td>
</tr>
<tr>
<td>Intel Core 2 Duo T9300 @ 2.5 GHz</td>
<td>20.20</td>
</tr>
<tr>
<td>AMD Phenom 8600B Triple-Core</td>
<td>35.30</td>
</tr>
</tbody>
</table>

Table 2 Comparison between Dual and Quad-Core Processors

<table>
<thead>
<tr>
<th>CPU</th>
<th>Calculation time / s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve method</td>
<td>Serial Strassen's</td>
</tr>
<tr>
<td>Intel Core 2 Duo T9300 @ 2.5 GHz</td>
<td>20.20</td>
</tr>
<tr>
<td>Intel Core 2 Quad Q6600 @ 2.4 GHz</td>
<td>23.75</td>
</tr>
</tbody>
</table>

Next, we can compare dual- and quad-core processors. Once again, processor with more cores is slower with naïve and serial Strassen's method for matrix multiplication, but it is almost twice faster with parallel Strassen's method.

5 Conclusion

After data analysis, we can see that the parallel C#
implementation of Strassen's algorithm for matrix multiplication is several times faster than the naïve method for matrix multiplication, if applied on large enough matrices. And, more important, that the difference grows with the number of cores in a processor. These results are expected. As processors with more cores are becoming more common, the parallel programming will become an important paradigm.

It was not a purpose of this research to provide a solution that can compete with commercial solutions, but to demonstrate the potential of multithread applications on multi-core processors.

A broader conclusion that can be deduced from this example is about the importance of multi-threading in programming, with the goal of efficient usage of resources. Although this point was stressed in the past few years, software still lies behind hardware. It also shows how some processor-time demanding application can be adapted to work on home-available hardware. This all leads to a more efficient application, and to a lower cost, which is always important in industry.

6 References

Literatura


