UPPSALA UNIVERSITY

COURSE NAME: HIGH PERFORMANCE AND PARALLEL COMPUTING

COURSE CODE: 1TD064

# Project - Matrix-matrix multiplication with Strassens algorithm

*Author:*
Alhassan JAWAD

Supervisor: Peter Munch

25-05-2024

# Matrix-matrix multiplication with Strassens algorithm

## Abstract

This project looks into **utilizing OpenMP** to **parallelize Strassen's algorithm** for matrix-matrix multiplication. The implementation of the algorithm in C is examined, with an emphasis on parallelizing and optimizing the serial code to make use of a computer's multi-core system. The study describes the measures implemented, such as **loop fusion methods** and **memory management improvements**, to **minimize computational complexity**. The findings demonstrate the substantial runtime gains made possible by parallelization, highlighting the influence of matrix size and structure on the algorithm's performance.

## 1 Introduction

At the same rate as technology advances, computer's get more and more complex and powerful where more cores can be used. This means that the advantage of running programs over multiple cores/threads/processes (meaning parallel) increases. The goal of this project is to choose a suitable problem to solve, write a serial code for it and then parallelize the code.

### 1.1 Problem description

The issue selected for this study is the matrix-matrix multiplication of square matrices by the use of *Strassen's Algorithm*. The technique reduces the computational cost of the Matrix-Matrix multiplication.

Three loops of iteration are typically used to compute a matrix-matrix multiplication. The computational cost of multiplying two square matrices of size n is $\theta(n^3)$. *Divide and conquer* is another popular strategy for multiplying square matrices where n is a power of two. The essential concept is to divide the resultant matrix (denoted C) and the two matrices that need to be multiplied (denoted A and B) into four sub-matrices, as shown in the following equation.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}, A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{1.1.1}$$

The C matrix can then be calculated as:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix} \tag{1.1.2}$$

*8 recursive calls* are required to calculate the four C sub-matrices since the sub-matrices' multiplications are done recursively. The divide-and-conquer technique has the same computational complexity as the iterative method with three for-loops, meaning $\theta(n^3)$ [1] [2].

Thus, we need to find a way to reduce the computational complexity as much as possible.

#### 1.1.1 History of how to reduce the complexity of eq. 1.1.2

*Volker Strassen* demonstrated in 1968 that it is possible to rewrite the divide-and-conquer algorithm so that it only requires *7 recursive calls*. Rewriting equation 1.1.2 accomplishes this. Firstly, as the following equation demonstrates, seven equations are computed:

$$\begin{cases} M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22}) \cdot B_{11} \\ M_3 = A_{11} \cdot (B_{12} - B_{22}) \\ M_4 = A_{22} \cdot (B_{21} - B_{11}) \\ M_5 = (A_{11} + A_{12}) \cdot B_{22} \\ M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \end{cases} \Bigg| \tag{1.1.3}$$

We see that each equation contains only a single matrix-matrix multiplication, and thus only *7 recursive calls* are needed. If we were to rewrite eq. 1.1.2 again this time using the equations from 1.1.3 we get:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 - M_3 + M_6 \end{bmatrix} \tag{1.1.4}$$

And since only 7 recursive calls are made instead of 8, we get a computational complexity that is $\theta(n^{log_2(7)}) \approx \theta(n^{2.8})$ [1].

## 1.2 The chosen approach

First, we will create a serial C implementation of the matmul algorithm. This will provide us a starting point for comparison and help us comprehend the algorithm's fundamental features. Next, we will analyze the serial code to identify which parts may be efficiently parallelized using OpenMP so it can be run on multiple processes. Our goal is to enhance the code's performance so that larger matrices can be multiplied with reduced computational cost.

This report shall present the workings of a serial matmul (matrix multiplication) code, analyze it and then start optimizing the serial code. The program language to be used is C and the code will be parallelized using the Open Multi-Processing (OpenMP) library.

We talked above about square matrices where n was a power of 2, but what about when the matrices are not of the power of 2? I solved that problem by padding the matrices with zeroes if n was odd before dividing the matrices into sub-matrices. On the other hand, if n was even, there would be no need for padding since the matrices could be partitioned evenly into four sub-matrices.

## 2 Solution Method

### 2.1 The Serial Code

The serial code takes 2 arguments, the first is the input file where the two matrices to be multiplied, A and B, are located. The input file includes an integer indicating the matrices' sizes, followed by a row-by-row breakdown of the A matrix's elements and a row-by-row breakdown of the B matrix's elements. Doubles were used to hold the components of the A and B matrices.

The name of the output file is the second parameter. In this file, the size of the C-matrix is recorded as an integer, after which the elements of the generated C matrix are written row by row and saved as doubles[1].

**Summarizing the serial code:**

Within the main function, the input arguments were read and stored as variables. Afterwards, memory for the A, B, and C matrices was allocated once the size was received from the input file. After that, the data was read from the file and inserted into the designated matrices for the A and B matrices. A custom function that used Strassen's algorithm to conduct the matrix-matrix multiplication was launched along with a timer started before.

**Regarding the custom Strassen multiplication function**, the size variable and the matrices A, B, and C (were to store it) were inputs to the function. The function's default scenario (base case) was for its inputs to be matrices of size 1x1. If that was the case then A and B can be multiplied directly, with the outcome being saved in C.

For the case when size is not a power of 2, I started by taking size modulo two to see if it was odd or even. The size was divided by two if the result was zero; if not, one was added to the size, and the result was then divided by two.

But the above solution didn't work very good for when size was equal to 1 as then the result of modulo was 1 and the size was 1 again. So I rewrote this part:

```
        if (rest == 0) {
            size = size / 2;
        // Matrix size in not a power of 2 => padding needed
        } else {
            size = (size + 1) / 2; }
```

to a single if statement that checked if the result of the modulo operation **AND** the size was above 1 then:

```
        if (rest == 0 && size > 1) { // Only divide if size is not 1
            size = size / 2; }
```

The padding mechanism makes sure that even sizes that are not powers of two are ultimately reduced to either a power of two or the base case of one, even if the code doesn't address these situations directly in the first if statement. As a result, the code runs properly for all matrix sizes—even those that aren't powers of two.

Next, I allocated memory for the matrices $A_{11} - A_{22}$, $B_{11} - B_{22}$, $C_{11} - C_{22}$ and $M_1 - M_7$ together with memory locations for temporary matrices for addition/subtraction. A quick observation of equations 1.1.1 and 1.1.3 shows that we need 29 malloc calls as we need memory locations reserved for 7 M-matrices, 10 temporary metrics for addition/subtract and 12 A-B-V-Matrices.

The next stage was to split the A and B matrices into their respective sub-matrices after the memory was allocated. If the size had been divisible by 2, i.e., the remaining variable was 0, then eq. 2.1.1

[1]If there exists no output file then one is created with the given name

shows how the split was to be made, with the same being done on the B-matrix.

$$\begin{cases} A_{11}[i][j] = A[i][j] \\ A_{12}[i][j] = A[i][j + \text{size}] \\ A_{21}[i][j] = A[i + \text{size}][j] \\ A_{22}[i][j] = A[i + \text{size}][j + \text{size}] \end{cases} \tag{2.1.1}$$

where:

$$i = [0, 1, ..., \text{size} - 1] \text{ \& } [j = 0, 1, ..., \text{size} - 1]$$

and $size$ is the size of the sub-matrix.

For the case when the size is not divisible by 2, I made a small modification/addition to the code were eq. 2.1.1 was reused again but this time i and j went [0, ..., size-2] instead of [0, ..., size-1]. That we ended the iterations at size - 2 means that all the rows & columns of the matrices were transferred except the last one. This operation would have given the same result in the case of padding with extra 0's to "reintroduce" a size that is divisible with 2[2]. The same was done for matrix B and its sub-matrices.

Being done with allocating memory for all matrices and their sub-matrices, I went to the next stage of calculating the M matrices shown in equation 1.1.3. This was done by first computing the additions/subtractions and storing them in temporary matrices that were called upon with the function recursively to handle the multiplication and redo the above-mentioned steps again.

Having calculated the matrices of eq. 1.1.3, the C matrix was computed using the sub-equations of equation 1.1.4. Afterwards the sub-matrices of C were merged together and then all allocated memory was freed.

## 2.2 Optimization of the Serial Code

Reducing the amount of malloc and free calls within the function that calculates Strassen's algorithm was the primary goal of the serial code optimization. As the memory on the stack was not enough to accommodate the potentially huge matrices, we need to store the matrices on the heap (thus the usage of malloc[3]).

The heap management must search for accessible space to store data when data is allocated, do garbage collection and designate it as available when memory is free. This process makes allocating and releasing memory on the heap quite time-consuming [3].

To further decrease the amount of malloc and free calls, the matrices were stored as 1d arrays instead of 2d arrays. A single array of size 29sizesize could thus hold the 29 matrices. Additionally, the matrices were kept in sequential order, with pointers referring to each matrix's initial element. For instance, $A_{11}$ started on element 0, $A_{12}$ on element $size \cdot size$, $A_{11}$ on element $2 \cdot size \cdot size$, and so on. By storing the matrices in this manner, only one malloc call was required for each function call, down from 29 before <=> only 1 free call is needed to free everything.

Loop fusion was another optimization method used. This required combining several for-loops into one, increasing effort for each iteration and decreasing loop control overhead due to fewer total iterations. AS the ten additions/subtractions of eq. 1.1.3 were stored in ten independent matrices, then ten operations can be done in a single for-loop so that we can save time. Similar optimization can be done for the splitting of the A and B matrices into sub-matrices and the computation and merging of the C-matrix.

Regarding optimization flags, *-O3* allowed the compiler to optimize aggressively. *-Wall* was used to capture any software problems or warnings. The compiler was made to optimize the code for the processor architecture it was compiled on by using the *-march=native* option.

[2]Didn't use padding so 1 extra row/column in matrix A didn't need to be allocated

[3]Otherwise we could use alloca() [3]

## 2.3 Parallelization of the Serial Code

For the parallelized code, I introduced a third parameter that will specify the number of processes as an input parameter to the program. As the I/O operations need to be done sequentially, they were left out of the parallelization and their loops were not parallelized.

In this section, I will just talk about parallelizing the optimized serial code. In case you want to check how to parallelize the non-optimized code, then check the references where I have all 4 code versions (non-optimized serial, optimized serial, parallel non-optimized serial, parallel optimized serial).

The first thing I did was using ***pragma omp parallel for schedule(static)*** to parallelize the part of the strassen function that calculates the matrix-matrix multiplication with Strassen's algorithm. Because of the static scheduling, the loop is split up across the processes quite equally. Static scheduling was chosen as the same amount of work can be done between all processes and thus there is no *load balancing* issues.

Regarding the 7 recursive calls that compute the M-matrices, the task directive of OpenMP was used to parallelize them within the code. First, ***pragma omp parallel*** is used to specify the parallel section. The various tasks are then created by a single thread using ***pragma omp single***. The tasks are defined using ***pragma omp task***, and are thereafter carried out in parallel.

## 3 Experiments

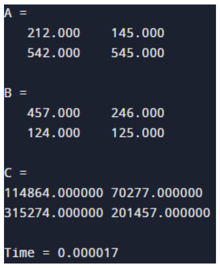
No specific files or instructions where given about how to test the codes' performance. That is why I created my own input files containing different random matrices using a simple MatLab code[4].

### 3.1 Hardware specifications

All performance experiments were performed on the *@vitsippa.it.uu.se* Linux machine hosted by Uppsala University [4]. The Linux machines are configured to have an AMD Opteron (Bulldozer) 6282SE@2.6GHz (16-core, dual socket) CPU, 128 GB memory and an Ubuntu 22.04.4 LTS operating system (OS) with a total of 467 processes according to the terminal information[5].

### 3.2 Performance of the code

At first, small scale tests using 2x2, 3x3 and 4x4 matrices were made to make sure the code is right.



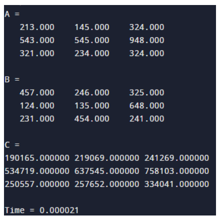
```
A =
   212.000     145.000
   542.000     545.000

B =
   457.000     246.000
   124.000     125.000

C =
114864.000000 70277.000000
315274.000000 201457.000000

Time = 0.000017
```

(a) 2x2 Matrices



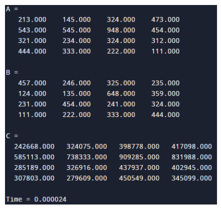
```
A =
   213.000     145.000     324.000
   543.000     545.000     948.000
   321.000     234.000     324.000

B =
   457.000     246.000     325.000
   124.000     135.000     648.000
   231.000     454.000     241.000

C =
190165.000000 219069.000000 241269.000000
534719.000000 637545.000000 758103.000000
250557.000000 257652.000000 334041.000000

Time = 0.000021
```

(b) 3x3 Matrices



```
A =
   213.000     145.000     324.000     473.000
   543.000     545.000     948.000     454.000
   321.000     234.000     324.000     312.000
   444.000     333.000     222.000     111.000

B =
   457.000     246.000     325.000     235.000
   124.000     135.000     648.000     359.000
   231.000     454.000     241.000     324.000
   111.000     222.000     333.000     444.000

C =
  242668.000    324075.000    398778.000    417098.000
  585113.000    738333.000    909285.000    831988.000
  285189.000    326916.000    437937.000    402945.000
  307803.000    279609.000    450549.000    345099.000

Time = 0.000024
```

(c) 4x4 Matrices

Figure 3.2.1: The matrices A & B and their element-wise multiplication matrix C, computed using Strassen's algorithm, for different dimensions.

The matrices were printed using nested-loops placed inside the code[6]:

```
printf("A = \n");
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        printf("%lf ", A[i][j]); }
    printf("\n"); }

printf("B = \n");
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        printf("%lf ", B[i][j]); }
    printf("\n"); }
...
...
printf("C = \n");
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        printf("%lf ", C[i][j]); }
    printf("\n"); }
```

As can be seen from the 3 figures illustrated in fig. 3.2.1, the code produces correct multiplication of A$\cdot B$ for all the cases with 2x2, 3x3 and 4x4 matrices.

[4]Given in references
[5]Seen after logging into vitsippa
[6]Removed them before submitting the final code

The next stage of performance experiments involved testing the program's execution time for various sizes and processes. Initially, power 2 matrices were examined.

Table 3.2.1: Time evaluations for matrix multiplication of power-of-2 matrices. As these matrices have an even size, it was never necessary to pad them with zeros. The used code is the parallelized optimized serial code.

| Matrix size: | 64 x 64 | 128 x 128 | 256 x 256 | 512 x 512 | 1024 x 1024 |
|---|---|---|---|---|---|
| Processes: 1 | Time(s): 0.126 | Time(s): 0.879 | Time(s): 6.143 | Time(s): 42.494 | Time(s): 302.135 |
| Processes: 2 | Time(s): 0.081 | Time(s): 0.569 | Time(s): 3.934 | Time(s): 27.612 | Time(s): 195.814 |
| Processes: 3 | Time(s): 0.061 | Time(s): 0.431 | Time(s): 2.982 | Time(s): 20.924 | Time(s): 148.730 |
| Processes: 4 | Time(s): 0.043 | Time(s): 0.302 | Time(s): 2.096 | Time(s): 14.697 | Time(s): 103.936 |
| Processes: 5 | Time(s): 0.042 | Time(s): 0.297 | Time(s): 2.061 | Time(s): 14.398 | Time(s): 102.488 |
| Processes: 6 | Time(s): 0.041 | Time(s): 0.288 | Time(s): 1.998 | Time(s): 14.047 | Time(s): 99.405 |
| Processes: 7 | Time(s): 0.023 | Time(s): 0.158 | Time(s): 1.102 | Time(s): 7.676 | Time(s): 54.931 |
| Processes: 8 | Time(s): 0.022 | Time(s): 0.155 | Time(s): 1.080 | Time(s): 7.500 | Time(s): 53.000 |

Afterwards, matrices that are not of power of 2 were used with the code:

Table 3.2.2: Time evaluations for matrix multiplication of not-power-of-2 matrices. As these matrices have an odd size, it is necessary that we pad them with zeros to make them even. The used code is the parallelized optimized serial code.

| Matrix size: | 111 x 111 | 222 x 222 | 444 x 444 | 555 x 555 | 666 x 666 |
|---|---|---|---|---|---|
| Processes: 1 | Time(s): 0.983 | Time(s): 6.619 | Time(s): 47.185 | Time(s): 92.342 | Time(s): 143.031 |
| Processes: 2 | Time(s): 0.615 | Time(s): 4.282 | Time(s): 30.112 | Time(s): 59.001 | Time(s): 92.364 |
| Processes: 3 | Time(s): 0.491 | Time(s): 3.344 | Time(s): 22.896 | Time(s): 45.000 | Time(s): 69.032 |
| Processes: 4 | Time(s): 0.322 | Time(s): 2.328 | Time(s): 15.915 | Time(s): 31.931 | Time(s): 48.782 |
| Processes: 5 | Time(s): 0.318 | Time(s): 2.289 | Time(s): 15.611 | Time(s): 30.687 | Time(s): 47.575 |
| Processes: 6 | Time(s): 0.308 | Time(s): 2.131 | Time(s): 15.543 | Time(s): 30.300 | Time(s): 46.371 |
| Processes: 7 | Time(s): 0.178 | Time(s): 1.227 | Time(s): 8.399 | Time(s): 16.568 | Time(s): 25.203 |
| Processes: 8 | Time(s): 0.169 | Time(s): 1.186 | Time(s): 8.581 | Time(s): 17.730 | Time(s): 26.351 |

With the needed data collected, we can start plotting graphs that show the effectiveness of the code.

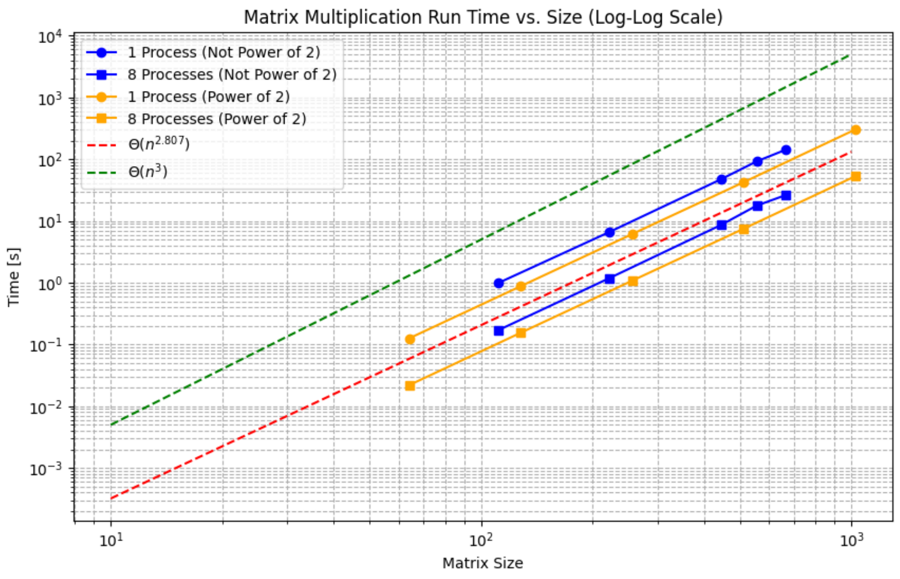


Figure 3.2.2: Runtime of the code plotted against matrix size.

The above figure combines both the results of tables 3.2.1 and 3.2.2. It shows how effective the algorithm is for power-of-2 matrices, for not-power-of-2 matrices, and how the computational time differs for sequential runs with 1 process versus parallel runtime with 8 processes.

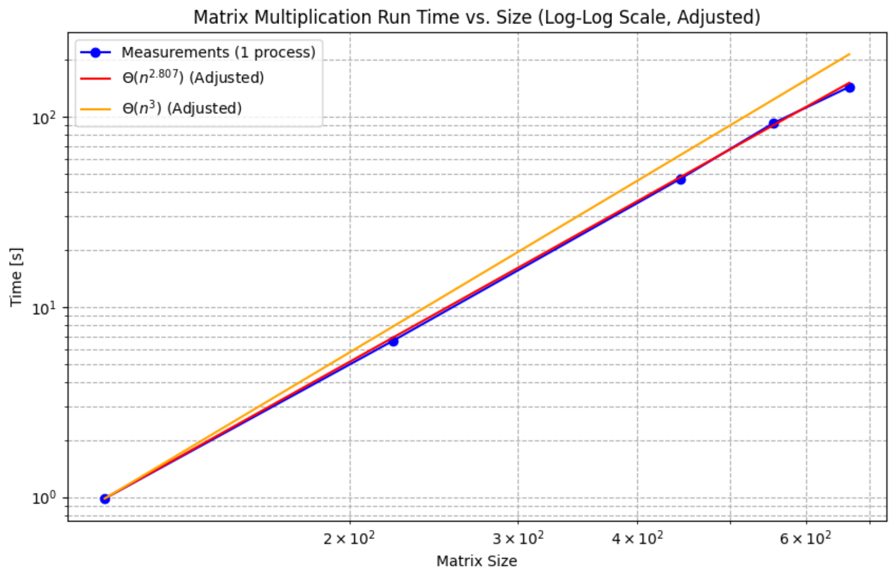


Figure 3.2.3: This figure shows the values of table 3.2.2 plotted as runtime versus matrix size

The following figure will show the speedup gotten for the power-of-2 matrices. Speedup is calculated as the parallel time divided with the sequential time (time for 1 processes).

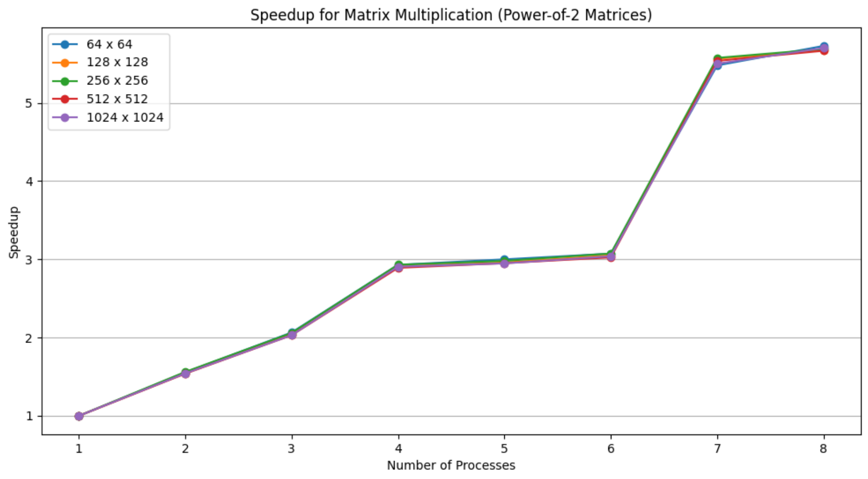


Figure 3.2.4: Speedup for the power-of-2 matrices plotted versus the number of processes.

## 4 Conclusion

This project strove to write a code that calculates the element-wise matrix multiplication of 2 matrices A & B and stores the result within matrix C. The algorithm used was Strassen's algorithm and the code works as it should, as shown in figure 3.2.1.

**Some key observations from figure 3.2.2:**

- Non-power-of-2 matrices always have a longer runtime than power-of-2 matrices. This is likely due to overhead creation because of padding the matrices with zeros so that their dimensions are even.
- The green line represents the runtime for power-of-2 matrices, which nearly matches the theoretical complexity of cubic time. This implies that for these matrix sizes, the algorithm is effective.
- For both types of matrices, the runtime is significantly reduced when the number of processes is increased from 1 to 8. This illustrates how parallelization in matrix multiplication works well.
- Runtime grows dramatically with matrix size, especially for non-power-of-2 matrices. This emphasizes how crucially effective algorithms are for big matrices.

The above observations conclude that the dimensions and structure of the matrices can have a significant impact on the effectiveness of matrix multiplication algorithms. In addition, runtime can be greatly increased via parallelization, particularly for big matrices, and non-power-of-2 matrices are more computationally expensive than power-of-2 matrices, most likely as a result of the addition of additional overhead.

**What we also see from figure 3.2.2** is that the written algorithm has a computational time complexity that is closer to $\theta(n^{2.8})$ than to $\theta(n^3)$, which is expected as the code implements Strassen's algorithm. The same thing can also be seen from figure 3.2.3 that shows how well the observed time matched the line displaying a $\theta(n^{2.8})$ time complexity.

**Figure 3.2.4 showed** a behavior that was noteworthy. When four processes were used, the speedup reached around 3. After that, the speedup was hardly noticeable between 4 and 6 processes until making a big surge at 7 processes, after which it leveled off. It seems that the behavior remained consistent over matrix sizes ranging from 1 to 6 processes, varying slightly between 7 and 8 processes.

As Strassen's algorithm have seven jobs executing the recursive matrix-matrix multiplication, it could be explained that 7 processes makes the code have ideal speedup. That is reasonable as with 7 processes, each process will only need to execute 1 time.

On the other hand, when for example 4 processes were employed, 4 processes completed the first four tasks, and then 3 processes completed the final three tasks while one processes waited. The same can be said for 5 and 6 processes in which 3 and 1 processes will wait and waste time, respectively.

## 4.1 Further improvements

**Looking at the code**, if we were to use this code to multiply bigger matrices than the ones I used here, then we need to modify the code so that the data type of the size variable becomes *long*.

The greatest matrix-matrix multiplication that can be computed as of now is around a size of 8740, as 29 matrices are allocated at once. This was calculated in the following way:

AVAILABLE MEMORY IN BYTES: 2GB IS EQUAL TO 2 * 1024 * 1024 * 1024 BYTES

MEMORY PER MATRIX: A SINGLE MATRIX OF SIZE N X N WOULD REQUIRE N * N * 8 BYTES OF MEMORY.

TOTAL MEMORY FOR 29 MATRICES: STORING 29 SUCH MATRICES WOULD NEED APPROXIMATELY 29 * N * N * 8 BYTES.

Setting the above "equations" equal to each other and solving for N gives us:

$$=> 29 \cdot N \cdot N \cdot 8 \approx 2 \cdot 1024 \cdot 1024 \cdot 1024$$

$$N^2 \approx \frac{(2 \cdot 1024 \cdot 1024 \cdot 1024)}{(29 \cdot 8)}$$

$$N \approx \sqrt{\frac{(2 \cdot 1024 \cdot 1024 \cdot 1024)}{(29 \cdot 8)}} => N \approx 8740$$

Since the size variable is an *int*, it will **overflow** if bigger matrices are used. This will likely result in a segmentation fault since an un-mapped address will be called.

**Another thing that** can be further improved is the usage of 10 different temporary matrices. In the serial code, 2 temporary matrices were allocated and used throughout the whole code. This is because 2 additions/subtraction were needed for the calculations of each M-matrix. Furthermore, because the serial code did everything sequentially, it was possible to reuse the same temporary matrices over and over again.

This was a challenge, though, when the code was parallelized, as different processes may alter the temporary matrices at the same time, providing the respective recursive function with the incorrect input matrix. This was why I introduced 8 additional temporary in the parallel code.

If we could somehow reduce the number of temporary matrices, for example by computing the first five temporary matrices (meaning that we calculating the $M_1 - M_4$) in parallel[7] and then reuse the same five temporary matrices again to compute the final four M-matrices, it should be feasible to lower the memory needed as the needed number of matrices would drop from 29 to 24.

[7]Possibly with nested parallelism enabled

## 5 References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.z *Introduction to Algorithms, Third Edition*, The MIT Press, 2009, page 75-82.

[2] Volker S., *Strassen algorithm*, in Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Strassen_algorithm, [Online]. Accessed on: May 27, 2024

[3] Agner Fog., *Optimizing software in C++*[8], Technical University of Denmark, 2004, page 95-96.

[8] Course Literature

## Appendix

### Non-optimized Serial Code

```c
/* High Performance and Parallel Computing 1TD064
   Chosen subject: Project - Matrix-matrix multiplication with Strassens algorithm
    The project is about implementing the Strassen algorithm for matrix-matrix
        multiplication.
    The algorithm is a divide and conquer algorithm that reduces the number of
        multiplications needed
    for the matrix-matrix multiplication from 8 to 7.
   @version
     1.0
   @since
     22-05-2024
   @desc
    The Strassens algorithm is implemented in C and the optimized serial version is
        implemented in this file.
    The parallel version is implemented in the file strassen_parallel.c.
   @inputs
    1. The filename of the file containing the A and B matrices to be multiplied.
    2. The filename of the file where the resulting C matrix will be stored.
    3. The last input argument was the used number of processes for
       running the program in parallel (used in the parallel version of the program)
  @outputs
    The program writes the resulting C matrix to the output file.
  @throwbacks
    Time taken fro the multiplication
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
//Main strassen function. The function is recursive and will call itself until the
    base case is reached.
// matmult = matrix multiplication
static void strassen_matmult(double **A, double **B, double **C, int size){

    // Base case when the matrix is 1x1
    if (size == 1) {
        C[0][0] = A[0][0] * B[0][0];
    } else {
        int rest = size % 2; // Remainder when dividing the matrix size by 2

        // Matrix size is a power of 2 => no padding needed
        if (rest == 0 && size > 1) { // Only divide if size is not 1
            size = size / 2; }
        /*
           Allocating memory for the sub-matrices, the 7 M matrices, and two
               temporary matrices
           to be used when addition is needed before multiplication.
        */

        double **C11 = (double**)malloc(size *sizeof(double*)); // Matrix C11
        double **C12 = (double**)malloc(size *sizeof(double*)); // Matrix C12
        double **C21 = (double**)malloc(size *sizeof(double*)); // Matrix C21
        double **C22 = (double**)malloc(size *sizeof(double*)); // Matrix C22

        double **A11 = (double**)malloc(size *sizeof(double*)); // Matrix A11
        double **A12 = (double**)malloc(size *sizeof(double*)); // Matrix A12
        double **A21 = (double**)malloc(size *sizeof(double*)); // Matrix A21
        double **A22 = (double**)malloc(size *sizeof(double*)); // Matrix A22
double **B11 = (double**)malloc(size *sizeof(double*)); // Matrix B11
double **B12 = (double**)malloc(size *sizeof(double*)); // Matrix B12
double **B21 = (double**)malloc(size *sizeof(double*)); // Matrix B21
double **B22 = (double**)malloc(size *sizeof(double*)); // Matrix B22

double **M1 = (double**)malloc(size *sizeof(double*)); // Matrix M1
double **M2 = (double**)malloc(size *sizeof(double*)); // Matrix M2
double **M3 = (double**)malloc(size *sizeof(double*)); // Matrix M3
double **M4 = (double**)malloc(size *sizeof(double*)); // Matrix M4
double **M5 = (double**)malloc(size *sizeof(double*)); // Matrix M5
double **M6 = (double**)malloc(size *sizeof(double*)); // Matrix M6
double **M7 = (double**)malloc(size *sizeof(double*)); // Matrix M7

double **temp1 = (double**)malloc(size *sizeof(double*)); // Temporary
    matrix to be used for addition
double **temp2 = (double**)malloc(size *sizeof(double*)); // Temporary
    matrix to be used for addition
// Allocating memory for the elements in the matrices
for (int i = 0; i<size; i++) {
    A11[i] = (double*)malloc(size *sizeof(double));
    A12[i] = (double*)malloc(size *sizeof(double));
    A21[i] = (double*)malloc(size *sizeof(double));
    A22[i] = (double*)malloc(size *sizeof(double));

    B11[i] = (double*)malloc(size *sizeof(double));
    B12[i] = (double*)malloc(size *sizeof(double));
    B21[i] = (double*)malloc(size *sizeof(double));
    B22[i] = (double*)malloc(size *sizeof(double));

    C11[i] = (double*)malloc(size *sizeof(double));
    C12[i] = (double*)malloc(size *sizeof(double));
    C21[i] = (double*)malloc(size *sizeof(double));
    C22[i] = (double*)malloc(size *sizeof(double));


    M1[i] = (double*)malloc(size *sizeof(double));
    M2[i] = (double*)malloc(size *sizeof(double));
    M3[i] = (double*)malloc(size *sizeof(double));
    M4[i] = (double*)malloc(size *sizeof(double));
    M5[i] = (double*)malloc(size *sizeof(double));
    M6[i] = (double*)malloc(size *sizeof(double));
    M7[i] = (double*)malloc(size *sizeof(double));

    temp1[i] = (double*)malloc(size *sizeof(double));
    temp2[i] = (double*)malloc(size *sizeof(double));
}

// Distributing the A and B matrices into the sub-matrices, only if the
    matrix size is power of 2
if (rest == 0){
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j+size];
            A21[i][j] = A[i+size][j];
            A22[i][j] = A[i+size][j+size];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j+size];
            B21[i][j] = B[i+size][j];
            B22[i][j] = B[i+size][j+size];
        }
    }

/*
```

```
    If the matrix is not of power of 2, the first to second to last rows and
        columns
    are transferred from the A and B matrix and the last row and column is
        determined
    by which sub-matrix it is, for example the last row and column of A22
        will be zeroes.
 */
 } else {
     for (int i = 0; i<(size-1); i++) {
         for (int j = 0; j<(size-1); j++) {
             A11[i][j] = A[i][j];
             A12[i][j] = A[i][j+size];
             A21[i][j] = A[i+size][j];
             A22[i][j] = A[i+size][j+size];

             B11[i][j] = B[i][j];
             B12[i][j] = B[i][j+size];
             B21[i][j] = B[i+size][j];
             B22[i][j] = B[i+size][j+size];
         }

         // The last rows and columns are (depending on the sub-matrix)
             either taken from A or B, or is padded with zeros
         A11[i][size-1] = A[i][size-1];
         A11[size-1][i] = A[size-1][i];
         A12[i][size-1] = 0;
         A12[size-1][i] = A[size-1][i+size];
         A21[i][size-1] = A[i+size][size-1];
         A21[size-1][i] = 0;
         A22[i][size-1] = 0;
         A22[size-1][i] = 0;




         B11[i][size-1] = B[i][size-1];
         B11[size-1][i] = B[size-1][i];
         B12[i][size-1] = 0;
         B12[size-1][i] = B[size-1][i+size];
         B21[i][size-1] = B[i+size][size-1];
         B21[size-1][i] = 0;
         B22[i][size-1] = 0;
         B22[size-1][i] = 0;
     }
     A11[size-1][size-1] = A[size-1][size-1];
     A12[size-1][size-1] = 0;
     A21[size-1][size-1] = 0;
     A22[size-1][size-1] = 0;

     B11[size-1][size-1] = B[size-1][size-1];
     B12[size-1][size-1] = 0;
     B21[size-1][size-1] = 0;
     B22[size-1][size-1] = 0;
}
```

```
/*
   For the following calculations, the two temporary matrices are used for
       addition
   before using Strassen's algorithm for multiplication.
*/
// M1 = (A11+A22)*(B11+B22)
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = A11[i][j] + A22[i][j];
        temp2[i][j] = B11[i][j] + B22[i][j];
    }
}
strassen_matmult(temp1, temp2, M1, size);

// M2 = (A21+A22)*B11
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = A21[i][j] + A22[i][j];
    }
}
strassen_matmult(temp1, B11, M2, size);

// M3 = A11*(B12-B22)
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = B12[i][j] - B22[i][j];
    }
}
strassen_matmult(A11, temp1, M3, size);

// M4 = A22*(B21-B11)
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = B21[i][j] - B11[i][j];
    }
}
strassen_matmult(A22, temp1, M4, size);

// M5 = (A11+A12)*B22
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = A11[i][j] + A12[i][j];
    }
}
strassen_matmult(temp1, B22, M5, size);

// M6 = (A21-A11)*(B11+B12)
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = A21[i][j] - A11[i][j];
        temp2[i][j] = B11[i][j] + B12[i][j];
    }
}
strassen_matmult(temp1, temp2, M6, size);

// M7 = (A12-A22)*(B21+B22)
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        temp1[i][j] = A12[i][j] - A22[i][j];
        temp2[i][j] = B21[i][j] + B22[i][j];
    }
}
strassen_matmult(temp1, temp2, M7, size);
```

```
// C11 = M1 + M4 - M5 + M7
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
    }
}

// C12 = M3 + M5
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        C12[i][j] = M3[i][j] + M5[i][j];
    }
}

// C21 = M2 + M4
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        C21[i][j] = M2[i][j] + M4[i][j];
    }
}

// C22 = M1 - M2 + M3 + M6
for (int i = 0; i<size; i++){
    for (int j = 0; j<size; j++){
        C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
    }
}

// If C is a power of 2, the sub-matrices are merged into C
if (rest == 0) {
    for (int i = 0; i<size; i++) {
        for (int j = 0; j<size; j++) {
            C[i][j] = C11[i][j];
            C[i][j+size] = C12[i][j];
            C[i+size][j] = C21[i][j];
            C[i+size][j+size] = C22[i][j];
        }
    }

// If C is not a power of 2, the padding is removed
} else {
    for (int i = 0; i<(size-1); i++) {
        for (int j = 0; j<(size-1); j++) {
            C[i][j] = C11[i][j];
            C[i][j+size] = C12[i][j];
            C[i+size][j] = C21[i][j];
            C[i+size][j+size] = C22[i][j];
        }
        C[i][size-1] = C11[i][size-1];
        C[size-1][i] = C11[size-1][i];
        C[size-1][i+size] = C12[size-1][i];
        C[i+size][size-1] = C21[i][size-1];
    }
    C[size-1][size-1] = C11[size-1][size-1]; }

// Free Memory
for (int i = 0; i<size; i++) {
    // Matrix C's elements
    free(C11[i]);
    free(C12[i]);
    free(C21[i]);
    free(C22[i]);
```

```
            // Matrix A's elements
            free(A11[i]);
            free(A12[i]);
            free(A21[i]);
            free(A22[i]);

            // Matrix B's elements
            free(B11[i]);
            free(B12[i]);
            free(B21[i]);
            free(B22[i]);

            // Matrix M's elements
            free(M1[i]);
            free(M2[i]);
            free(M3[i]);
            free(M4[i]);
            free(M5[i]);
            free(M6[i]);
            free(M7[i]);

            // Temporary matrix's elements
            free(temp1[i]);
            free(temp2[i]);
        }
        // Matrix C
        free(C11);
        free(C12);
        free(C21);
        free(C22);

        // Matrix A
        free(A11);
        free(A12);
        free(A21);
        free(A22);

        // Matrix B
        free(B11);
        free(B12);
        free(B21);
        free(B22);

        // Matrix M
        free(M1);
        free(M2);
        free(M3);
        free(M4);
        free(M5);
        free(M6);
        free(M7);

        // Temporary matrix
        free(temp1);
        free(temp2);
    }
}
```

```
int main(int argc, char **argv){
    // Checking the number of arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input file> <output file>\n", argv[0]);
        exit(1);
    }
    // Initializing variables
    char filename[60]; // 60 characters for the filename
    strcpy(filename, argv[1]); // strcpy copies the string from argv[1] to filename
    char outname[60];
    strcpy(outname, argv[2]);

    // Reading the input file
    int size;
    FILE *input;
    input = fopen(filename, "r"); // r = read
    // Checking if the file can be opened
    if (input == NULL) {
        fprintf(stderr, "The following file can not be opened: %s", filename);
        exit(1);
    }
    // read_res = number of elements read
    int read_res = fread(&size, sizeof(int), 1, input);

    //Allocating memory for the A and B matrices to be multiplied and for the C
         matrix where the result will be stored
    double **A = (double**)malloc(size *sizeof(double*)); // A matrix
    double **B = (double**)malloc(size *sizeof(double*)); // B matrix
    double **C = (double**)malloc(size *sizeof(double*)); // C matrix

    for (int i = 0; i < size; i++) {
        A[i] = (double*)malloc(size *sizeof(double));
        B[i] = (double*)malloc(size *sizeof(double));
        C[i] = (double*)malloc(size *sizeof(double));
    }

    // Extract the matrices A and B from the input file
    // Matrix A
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            read_res = fread(&A[i][j], sizeof(double), 1, input); }
    }
    // Matrix B
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            read_res = fread(&B[i][j], sizeof(double), 1, input); }
    }
    fclose(input); // Close the input file

    // Start measuring the time
    clock_t start = clock();
    strassen_matmult(A, B, C, size);
    clock_t end = clock();
    double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for multiplication: %f seconds\n", cpu_time_used);
    // Store the resulting matrix C in the output file
    FILE *output;
    output = fopen(outname, "w"); // w = write
    // Checking if the file can be opened
    if (output == NULL) {
        fprintf(stderr, "The following file can not be opened: %s", outname);
        exit(1);
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fwrite(&C[i][j], sizeof(double), 1, output); }
    }
    fclose(output); // Close the output file

    // Free Memory
    for (int i = 0; i < size; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);
    return 0;
}
```

## Optimized Serial Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

static void strassen_matmult(double *A, double *B, double *C, int size) {
    // Allocate memory for all matrices combined
    double *combined_matrices = (double *)malloc(29 * size * size * sizeof(double));
    if (combined_matrices == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }

    // Pointers to mark the beginning of each matrix within the combined array
    double *A11 = combined_matrices;
    double *A12 = A11 + size * size;
    double *A21 = A12 + size * size;
    double *A22 = A21 + size * size;

    double *B11 = A22 + size * size;
    double *B12 = B11 + size * size;
    double *B21 = B12 + size * size;
    double *B22 = B21 + size * size;

    double *C11 = B22 + size * size;
    double *C12 = C11 + size * size;
    double *C21 = C12 + size * size;
    double *C22 = C21 + size * size;

    double *M1 = C22 + size * size;
    double *M2 = M1 + size * size;
    double *M3 = M2 + size * size;
    double *M4 = M3 + size * size;
    double *M5 = M4 + size * size;
    double *M6 = M5 + size * size;
    double *M7 = M6 + size * size;

    double *temp1 = M7 + size * size;
    double *temp2 = temp1 + size * size;
    double *temp3 = temp2 + size * size;
    double *temp4 = temp3 + size * size;
    double *temp5 = temp4 + size * size;
    double *temp6 = temp5 + size * size;
    double *temp7 = temp6 + size * size;

    // Initialize matrices A and B to the combined array
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            A11[i * size + j] = A[i * size + j];
            A12[i * size + j] = A[i * size + j + size];
            A21[i * size + j] = A[(i + size) * size + j];
            A22[i * size + j] = A[(i + size) * size + j + size];

            B11[i * size + j] = B[i * size + j];
            B12[i * size + j] = B[i * size + j + size];
            B21[i * size + j] = B[(i + size) * size + j];
            B22[i * size + j] = B[(i + size) * size + j + size];
        }
    }
    // Compute M1 to M7
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            // Calculate M1 = (A11+A22)*(B11+B22)
            temp1[i * size + j] = (A11[i * size + j] + A22[i * size + j]) * (B11[i *
                size + j] + B22[i * size + j]);

            // Calculate M2 = (A21+A22)*B11
            temp2[i * size + j] = (A21[i * size + j] + A22[i * size + j]) * B11[i *
                size + j];

            // Calculate M3 = A11*(B12-B22)
            temp3[i * size + j] = A11[i * size + j] * (B12[i * size + j] - B22[i *
                size + j]);

            // Calculate M4 = A22*(B21-B11)
            temp4[i * size + j] = A22[i * size + j] * (B21[i * size + j] - B11[i *
                size + j]);

            // Calculate M5 = (A11+A12)*B22
            temp5[i * size + j] = (A11[i * size + j] + A12[i * size + j]) * B22[i *
                size + j];

            // Calculate M6 = (A21-A11)*(B11+B12)
            temp6[i * size + j] = (A21[i * size + j] - A11[i * size + j]) * (B11[i *
                size + j] + B12[i * size + j]);

            // Calculate M7 = (A12-A22)*(B21+B22)
            temp7[i * size + j] = (A12[i * size + j] - A22[i * size + j]) * (B21[i *
                size + j] + B22[i * size + j]);
        }
    }

    // Recursively compute C matrices using M1 to M7
    strassen_matmult(temp1, temp2, M1, size);
    strassen_matmult(temp3, B22, M2, size);
    strassen_matmult(A11, temp4, M3, size);
    strassen_matmult(A22, temp5, M4, size);
    strassen_matmult(temp6, temp7, M5, size);

    // Calculate C11, C12, C21, C22 using M1 to M7
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C11[i * size + j] = M1[i * size + j] + M4[i * size + j] - M5[i * size +
                j] + M7[i * size + j];
            C12[i * size + j] = M3[i * size + j] + M5[i * size + j];
            C21[i * size + j] = M2[i * size + j] + M4[i * size + j];
            C22[i * size + j] = M1[i * size + j] - M2[i * size + j] + M3[i * size +
                j] + M6[i * size + j];
        }
    }

    // Free memory after function execution
    free(combined_matrices);
}
```

```
int main(int argc, char **argv) {
    // Check the number of arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input file> <output file>\n", argv[0]);
        exit(1);
    }

    // Initialize variables
    char filename[60];
    strcpy(filename, argv[1]);
    char outname[60];
    strcpy(outname, argv[2]);

    // Read the input file
    int size;
    FILE *input;
    input = fopen(filename, "r");
    if (input == NULL) {
        fprintf(stderr, "Error: Unable to open input file: %s\n", filename);
        exit(1);
    }
    int read_res = fread(&size, sizeof(int), 1, input);

    // Allocate memory for matrices A, B, and C
    double *A = (double*)malloc(size * size * sizeof(double));
    double *B = (double*)malloc(size * size * sizeof(double));
    double *C = (double*)malloc(size * size * sizeof(double));

    // Read matrices A and B from the input file
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            read_res = fread(&A[i * size + j], sizeof(double), 1, input);
            if (read_res != 1) {
                fprintf(stderr, "Error: Failed to read from input file\n");
                exit(1);
            }}}
    fclose(input);

    // Perform matrix multiplication
    clock_t start = clock();
    strassen_matmult(A, B, C, size);
    clock_t end = clock();
    double cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for multiplication: %f seconds\n", cpu_time_used);

    // Write the resulting matrix C to the output file
    FILE *output;
    output = fopen(outname, "w");
    if (output == NULL) {
        fprintf(stderr, "Error: Unable to open output file: %s\n", outname);
        exit(1);}
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fwrite(&C[i * size + j], sizeof(double), 1, output);
        }}
    fclose(output);

    // Free allocated memory
    free(A);
    free(B);
    free(C);

    return 0;
}
```

**Parallelized Non-optimized Serial Code**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

//Main strassen function. The function is recursive and will call itself until the
    base case is reached.
// matmult = matrix multiplication
static void strassen_matmult(double **A, double **B, double **C, int size){

   // Base case when the matrix is 1x1
   if (size == 1) {
       C[0][0] = A[0][0] * B[0][0];
   } else {
       int rest = size % 2; // Remainder when dividing the matrix size by 2

       // Matrix size is a power of 2 => no padding needed
       if (rest == 0 && size > 1) { // Only divide if size is not 1
           size = size / 2; }
       /*
          Allocating memory for the sub-matrices, the 7 M matrices, and two
              temporary matrices
          to be used when addition is needed before multiplication.
       */

       double **C11 = (double**)malloc(size *sizeof(double*)); // Matrix C11
       double **C12 = (double**)malloc(size *sizeof(double*)); // Matrix C12
       double **C21 = (double**)malloc(size *sizeof(double*)); // Matrix C21
       double **C22 = (double**)malloc(size *sizeof(double*)); // Matrix C22

       double **A11 = (double**)malloc(size *sizeof(double*)); // Matrix A11
       double **A12 = (double**)malloc(size *sizeof(double*)); // Matrix A12
       double **A21 = (double**)malloc(size *sizeof(double*)); // Matrix A21
       double **A22 = (double**)malloc(size *sizeof(double*)); // Matrix A22

       double **B11 = (double**)malloc(size *sizeof(double*)); // Matrix B11
       double **B12 = (double**)malloc(size *sizeof(double*)); // Matrix B12
       double **B21 = (double**)malloc(size *sizeof(double*)); // Matrix B21
       double **B22 = (double**)malloc(size *sizeof(double*)); // Matrix B22

       double **M1 = (double**)malloc(size *sizeof(double*)); // Matrix M1
       double **M2 = (double**)malloc(size *sizeof(double*)); // Matrix M2
       double **M3 = (double**)malloc(size *sizeof(double*)); // Matrix M3
       double **M4 = (double**)malloc(size *sizeof(double*)); // Matrix M4
       double **M5 = (double**)malloc(size *sizeof(double*)); // Matrix M5
       double **M6 = (double**)malloc(size *sizeof(double*)); // Matrix M6
       double **M7 = (double**)malloc(size *sizeof(double*)); // Matrix M7

       double **temp1 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp2 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp3 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp4 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp5 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp6 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
       double **temp7 = (double**)malloc(size *sizeof(double*)); // Temporary
           matrix to be used for addition
```

```
double **temp8 = (double**)malloc(size *sizeof(double*)); // Temporary
    matrix to be used for addition
double **temp9 = (double**)malloc(size *sizeof(double*)); // Temporary
    matrix to be used for addition
double **temp10 = (double**)malloc(size *sizeof(double*)); // Temporary
    matrix to be used for addition
// Allocating memory for the elements in the matrices
// Here omp parallel for is used to parallelize the for loop
#pragma omp parallel for {
    for (int i = 0; i < size; i++) {
        A11[i] = (double*)malloc(size *sizeof(double));
        A12[i] = (double*)malloc(size *sizeof(double));
        A21[i] = (double*)malloc(size *sizeof(double));
        A22[i] = (double*)malloc(size *sizeof(double));

        B11[i] = (double*)malloc(size *sizeof(double));
        B12[i] = (double*)malloc(size *sizeof(double));
        B21[i] = (double*)malloc(size *sizeof(double));
        B22[i] = (double*)malloc(size *sizeof(double));

        C11[i] = (double*)malloc(size *sizeof(double));
        C12[i] = (double*)malloc(size *sizeof(double));
        C21[i] = (double*)malloc(size *sizeof(double));
        C22[i] = (double*)malloc(size *sizeof(double));

        M1[i] = (double*)malloc(size *sizeof(double));
        M2[i] = (double*)malloc(size *sizeof(double));
        M3[i] = (double*)malloc(size *sizeof(double));
        M4[i] = (double*)malloc(size *sizeof(double));
        M5[i] = (double*)malloc(size *sizeof(double));
        M6[i] = (double*)malloc(size *sizeof(double));
        M7[i] = (double*)malloc(size *sizeof(double));

        temp1[i] = (double*)malloc(size *sizeof(double));
        temp2[i] = (double*)malloc(size *sizeof(double));
        temp3[i] = (double*)malloc(size *sizeof(double));
        temp4[i] = (double*)malloc(size *sizeof(double));
        temp5[i] = (double*)malloc(size *sizeof(double));
        temp6[i] = (double*)malloc(size *sizeof(double));
        temp7[i] = (double*)malloc(size *sizeof(double));
        temp8[i] = (double*)malloc(size *sizeof(double));
        temp9[i] = (double*)malloc(size *sizeof(double));
        temp10[i] = (double*)malloc(size *sizeof(double));
    }
}

// Distributing the A and B matrices into the sub-matrices, only if the
    matrix size is power of 2
if (rest == 0) {
    // Here omp parallel for is used to parallelize the for loop
    #pragma omp parallel for {
        for(int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                // Matrix A division
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j+size];
                A21[i][j] = A[i+size][j];
                A22[i][j] = A[i+size][j+size];
                // Matrix B division
                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j+size];
                B21[i][j] = B[i+size][j];
                B22[i][j] = B[i+size][j+size];
            }
        }}
/*
   If the matrix is not of power of 2, the first to second to last rows and
       columns
   are transferred from the A and B matrix and the last row and column is
       determined
   by which sub-matrix it is, for example the last row and column of A22
       will be zeroes.
*/
} else {
    #pragma omp parallel for {
        for (int i = 0; i < (size-1); i++) {
            for (int j = 0; j < (size-1); j++) {
                A11[i][j] = A[i][j];
                A12[i][j] = A[i][j+size];
                A21[i][j] = A[i+size][j];
                A22[i][j] = A[i+size][j+size];

                B11[i][j] = B[i][j];
                B12[i][j] = B[i][j+size];
                B21[i][j] = B[i+size][j];
                B22[i][j] = B[i+size][j+size];
            }

            // The last rows and columns are (depending on the sub-matrix)
                either taken from A or B, or is padded with zeros
            A11[i][size-1] = A[i][size-1];
            A11[size-1][i] = A[size-1][i];
            A12[i][size-1] = 0;
            A12[size-1][i] = A[size-1][i+size];
            A21[i][size-1] = A[i+size][size-1];
            A21[size-1][i] = 0;
            A22[i][size-1] = 0;
            A22[size-1][i] = 0;

            B11[i][size-1] = B[i][size-1];
            B11[size-1][i] = B[size-1][i];
            B12[i][size-1] = 0;
            B12[size-1][i] = B[size-1][i+size];
            B21[i][size-1] = B[i+size][size-1];
            B21[size-1][i] = 0;
            B22[i][size-1] = 0;
            B22[size-1][i] = 0;
        }
    }
    A11[size-1][size-1] = A[size-1][size-1];
    A12[size-1][size-1] = 0;
    A21[size-1][size-1] = 0;
    A22[size-1][size-1] = 0;

    B11[size-1][size-1] = B[size-1][size-1];
    B12[size-1][size-1] = 0;
    B21[size-1][size-1] = 0;
    B22[size-1][size-1] = 0;
}
```

```
/*
   For the following calculations, the two temporary matrices are used for
       addition
   before using Strassen's algorithm for multiplication.
*/
#pragma omp parallel
{
    // single directive ensures execution by a single thread at a time
    #pragma omp single
    {
        // M1 = (A11+A22)*(B11+B22)
        // task directive ensures that the tasks are created for each thread
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp1[i][j] = A11[i][j] + A22[i][j];
                    temp2[i][j] = B11[i][j] + B22[i][j];
                }
            }
            strassen_matmult(temp1, temp2, M1, size);
        }

        // M2 = (A21+A22)*B11
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp3[i][j] = A21[i][j] + A22[i][j];
                }
            }
            strassen_matmult(temp3, B11, M2, size);
        }


        // M3 = A11*(B12-B22)
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp4[i][j] = B12[i][j] - B22[i][j];
                }
            }
            strassen_matmult(A11, temp4, M3, size);
        }


        // M4 = A22*(B21-B11)
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp5[i][j] = B21[i][j] - B11[i][j];
                }
            }
            strassen_matmult(A22, temp5, M4, size);
        }
        //Calculating M5 = (A11+A12)*B22
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp6[i][j] = A11[i][j] + A12[i][j];
                }
            }
            strassen_matmult(temp6, B22, M5, size);
        }


        // M6 = (A21-A11)*(B11+B12)
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp7[i][j] = A21[i][j] - A11[i][j];
                    temp8[i][j] = B11[i][j] + B12[i][j];
                }
            }
            strassen_matmult(temp7, temp8, M6, size);
        }


        // M7 = (A12-A22)*(B21+B22)
        #pragma omp task
        {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    temp9[i][j] = A12[i][j] - A22[i][j];
                    temp10[i][j] = B21[i][j] + B22[i][j];
                }
            }
            strassen_matmult(temp9, temp10, M7, size);
        }
    }
}
// C11 = M1 + M4 - M5 + M7
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
    }
}

// C12 = M3 + M5
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        C12[i][j] = M3[i][j] + M5[i][j];
    }
}

// C21 = M2 + M4
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        C21[i][j] = M2[i][j] + M4[i][j];
    }
}

// C22 = M1 - M2 + M3 + M6
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
    }
}
```

```
// If C is a power of 2, the sub-matrices are merged into C
if (rest == 0) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i][j] = C11[i][j];
            C[i][j+size] = C12[i][j];
            C[i+size][j] = C21[i][j];
            C[i+size][j+size] = C22[i][j];
        }
    }
// If C is not a power of 2, the padding is removed
} else {
    for (int i = 0; i < (size-1); i++) {
        for (int j = 0; j < (size-1); j++) {
            C[i][j] = C11[i][j];
            C[i][j+size] = C12[i][j];
            C[i+size][j] = C21[i][j];
            C[i+size][j+size] = C22[i][j];
        }
        C[i][size-1] = C11[i][size-1];
        C[size-1][i] = C11[size-1][i];
        C[size-1][i+size] = C12[size-1][i];
        C[i+size][size-1] = C21[i][size-1];
    }
    C[size-1][size-1] = C11[size-1][size-1];
}

// Free Memory
for (int i = 0; i < size; i++){
    // Matrix A's elements
    free(A11[i]);
    free(A12[i]);
    free(A21[i]);
    free(A22[i]);
    // Matrix B's elements
    free(B11[i]);
    free(B12[i]);
    free(B21[i]);
    free(B22[i]);
    // Matrix C's elements
    free(C11[i]);
    free(C12[i]);
    free(C21[i]);
    free(C22[i]);
    // Matrix M's elements
    free(M1[i]);
    free(M2[i]);
    free(M3[i]);
    free(M4[i]);
    free(M5[i]);
    free(M6[i]);
    free(M7[i]);
    // Temporary matrices' elements
    free(temp1[i]);
    free(temp2[i]);
    free(temp3[i]);
    free(temp4[i]);
    free(temp5[i]);
    free(temp6[i]);
    free(temp7[i]);
    free(temp8[i]);
    free(temp9[i]);
    free(temp10[i]);
}
```

```
        // Matrix A
        free(A11);
        free(A12);
        free(A21);
        free(A22);
        // Matrix B
        free(B11);
        free(B12);
        free(B21);
        free(B22);
        // Matrix C
        free(C11);
        free(C12);
        free(C21);
        free(C22);
        // Matrix M
        free(M1);
        free(M2);
        free(M3);
        free(M4);
        free(M5);
        free(M6);
        free(M7);
        // Temporary matrices
        free(temp1);
        free(temp2);
        free(temp3);
        free(temp4);
        free(temp5);
        free(temp6);
        free(temp7);
        free(temp8);
        free(temp9);
        free(temp10);
    }
}


int main(int argc, char **argv) {
    // Checking the number of arguments
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <input file> <output file> <number of
            threads>\n", argv[0]);
        exit(1);
    }

    // Initializing variables
    char filename[60]; // 60 characters for the filename
    strcpy(filename, argv[1]); // strcpy copies the string from argv[1] to filename
    char outname[60];
    strcpy(outname, argv[2]);
    const int threads = atoi(argv[3]);
    printf("Threads = %d\n", threads);
    omp_set_num_threads(threads); // Setting the number of threads to be used in
        the parallel region

    // Reading the input file
    int size;
    FILE *input;
    input = fopen(filename, "r"); // r = read mode
    // Checking if the file can be opened
    if (input == NULL) {
        fprintf(stderr, "The following file can not be opened: %s", filename);
        exit(1);
    }
    // read_res = number of elements read
    int read_res;
    read_res = fread(&size, sizeof(int), 1, input);

    //Allocating memory for the A and B matrices to be multiplied and for the C
        matrix where the result will be stored
    double **A = (double**)malloc(size *sizeof(double*)); // A matrix
    double **B = (double**)malloc(size *sizeof(double*)); // B matrix
    double **C = (double**)malloc(size *sizeof(double*)); // C matrix
    // Here omp parallel for is used to parallelize the for loop
    #pragma omp parallel for
        for (int i = 0; i < size; i++) {
            A[i] = (double*)malloc(size*sizeof(double));
            B[i] = (double*)malloc(size*sizeof(double));
            C[i] = (double*)malloc(size*sizeof(double));
        }


    // Extract the matrices A and B from the input file
    // Matrix A
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            read_res = fread(&A[i][j], sizeof(double), 1, input);
        }
    }
    // Matrix B
    for(int i = 0; i<size; i++){
        for(int j = 0; j<size; j++){
            read_res = fread(&B[i][j], sizeof(double), 1, input);
        }
    }
    fclose(input); // Closing the input file
    // Start the timer
    double start_time = omp_get_wtime();
    strassen_matmult(A, B, C, size);
    double end_time = omp_get_wtime();
    double elapsed_time = end_time - start_time;
    printf("Time taken for multiplication: %f\n", elapsed_time);

    // Store the resulting matrix C in the output file
    FILE *output;
    output = fopen(outname, "w"); // w = write mode
    if (output == NULL) {
        fprintf(stderr, "The following file can not be opened: %s", outname);
        exit(1);
    }
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            fwrite(&C[i][j], sizeof(double), 1, output);
        }
    }
    fclose(output); // Closing the output file

    // Free memory
    for (int i = 0; i<size; i++) {
            free(A[i]);
            free(B[i]);
            free(C[i]);
    }
    free(A);
    free(B);
    free(C);
    return 0;
}
```

**Parallelized Optimised Serial Code**

```
/* High Performance and Parallel Computing 1TD064
   Chosen subject: Project - Matrix-matrix multiplication with Strassens algorithm
    The project is about implementing the Strassen algorithm for matrix-matrix
        multiplication.
    The algorithm is a divide and conquer algorithm that reduces the number of
        multiplications needed
    for the matrix-matrix multiplication from 8 to 7.
   @version
     1.4
   @since
     23-05-2024
   @desc
    The Strassens algorithm is implemented in C and the parallel version is
        implemented in this file.
    The optimized serial version is implemented in the file
        strassen_serial_optimized.c.
   @inputs
    1. The filename of the file containing the A and B matrices to be multiplied.
    2. The filename of the file where the resulting C matrix will be stored.
    3. The last input argument was the used number of processes for
       running the program in parallel
  @outputs
    The program writes the resulting C matrix to the output file.
  @throwbacks
    Time taken fro the multiplication
*/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

static void strassen_matmult(double *A, double *B, double *C, int size) {
    // Allocate memory for all matrices combined
    double *combined_matrices = (double *)malloc(29 * size * size * sizeof(double));
    if (combined_matrices == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    // Pointers to mark the beginning of each matrix within the combined array
    double *A11 = combined_matrices;
    double *A12 = A11 + size * size;
    double *A21 = A12 + size * size;
    double *A22 = A21 + size * size;

    double *B11 = A22 + size * size;
    double *B12 = B11 + size * size;
    double *B21 = B12 + size * size;
    double *B22 = B21 + size * size;

    double *C11 = B22 + size * size;
    double *C12 = C11 + size * size;
    double *C21 = C12 + size * size;
    double *C22 = C21 + size * size;

    double *M1 = C22 + size * size;
    double *M2 = M1 + size * size;
    double *M3 = M2 + size * size;
    double *M4 = M3 + size * size;
    double *M5 = M4 + size * size;
    double *M6 = M5 + size * size;
    double *M7 = M6 + size * size;
double *temp1 = M7 + size * size;
double *temp2 = temp1 + size * size;
double *temp3 = temp2 + size * size;
double *temp4 = temp3 + size * size;
double *temp5 = temp4 + size * size;
double *temp6 = temp5 + size * size;
double *temp7 = temp6 + size * size;

// Initialize matrices A and B to the combined array
#pragma omp parallel for collapse(2)
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        A11[i * size + j] = A[i * size + j];
        A12[i * size + j] = A[i * size + j + size];
        A21[i * size + j] = A[(i + size) * size + j];
        A22[i * size + j] = A[(i + size) * size + j + size];

        B11[i * size + j] = B[i * size + j];
        B12[i * size + j] = B[i * size + j + size];
        B21[i * size + j] = B[(i + size) * size + j];
        B22[i * size + j] = B[(i + size) * size + j + size];
    }
}

// Compute M1 to M7
#pragma omp parallel for collapse(2) shared(A11, A22, B11, B22, temp1, size)
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        // Calculate M1 = (A11+A22)*(B11+B22)
        temp1[i * size + j] = (A11[i * size + j] + A22[i * size + j]) * (B11[i *
            size + j] + B22[i * size + j]);
    }
}

// Compute M2 to M7 in parallel
#pragma omp parallel for schedule(static) shared(A, B, size, temp2, temp3,
    temp4, temp5, temp6, temp7)
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        // Calculate M2 = (A21+A22)*B11
        temp2[i * size + j] = (A21[i * size + j] + A22[i * size + j]) * B11[i *
            size + j];

        // Calculate M3 = A11*(B12-B22)
        temp3[i * size + j] = A11[i * size + j] * (B12[i * size + j] - B22[i *
            size + j]);

        // Calculate M4 = A22*(B21-B11)
        temp4[i * size + j] = A22[i * size + j] * (B21[i * size + j] - B11[i *
            size + j]);

        // Calculate M5 = (A11+A12)*B22
        temp5[i * size + j] = (A11[i * size + j] + A12[i * size + j]) * B22[i *
            size + j];

        // Calculate M6 = (A21-A11)*(B11+B12)
        temp6[i * size + j] = (A21[i * size + j] - A11[i * size + j]) * (B11[i *
            size + j] + B12[i * size + j]);

        // Calculate M7 = (A12-A22)*(B21+B22)
        temp7[i * size + j] = (A12[i * size + j] - A22[i * size + j]) * (B21[i *
            size + j] + B22[i * size + j]);
    }
}
```

```
    // Recursively compute C matrices using M1 to M7
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        strassen_matmult(temp1, temp2, M1, size);
        #pragma omp task
        strassen_matmult(temp3, B22, M2, size);
        #pragma omp task
        strassen_matmult(A11, temp4, M3, size);
        #pragma omp task
        strassen_matmult(A22, temp5, M4, size);
        #pragma omp task
        strassen_matmult(temp6, temp7, M5, size);
        #pragma omp taskwait
    }

    // Calculate C11, C12, C21, C22 using M1 to M7
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C11[i * size + j] = M1[i * size + j] + M4[i * size + j] - M5[i * size +
                j] + M7[i * size + j];
            C12[i * size + j] = M3[i * size + j] + M5[i * size + j];
            C21[i * size + j] = M2[i * size + j] + M4[i * size + j];
            C22[i * size + j] = M1[i * size + j] - M2[i * size + j] + M3[i * size +
                j] + M6[i * size + j];
        }
    }

    // Free memory after function execution
    free(combined_matrices);
}

int main(int argc, char **argv) {
    // Checking the number of arguments
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <input file> <output file> <number of
            threads>\n", argv[0]);
        exit(1);
    }

    // Initialize variables
    char filename[60];
    strcpy(filename, argv[1]);
    char outname[60];
    strcpy(outname, argv[2]);
    const int threads = atoi(argv[3]);
    printf("Threads = %d\n", threads);
    omp_set_num_threads(threads); // Setting the number of threads to be used in
        the parallel region

    // Read the input file
    int size;
    FILE *input;
    input = fopen(filename, "r");
    if (input == NULL) {
        fprintf(stderr, "Error: Unable to open input file: %s\n", filename);
        exit(1);
    }
    int read_res = fread(&size, sizeof(int), 1, input);
    // Allocate memory for matrices A, B, and C
    double *A = (double*)malloc(size * size * sizeof(double));
    double *B = (double*)malloc(size * size * sizeof(double));
    double *C = (double*)malloc(size * size * sizeof(double));

    // Read matrices A and B from the input file
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            read_res = fread(&A[i * size + j], sizeof(double), 1, input);
            if (read_res != 1) {
                fprintf(stderr, "Error: Failed to read from input file\n");
                exit(1);
            }
        }
    }
    fclose(input);
    // Perform matrix multiplication
    clock_t start = clock();
    strassen_matmult(A, B, C, size);
    clock_t end = clock();
    double cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for multiplication: %f seconds\n", cpu_time_used);

    // Write the resulting matrix C to the output file
    FILE *output;
    output = fopen(outname, "w");
    if (output == NULL) {
        fprintf(stderr, "Error: Unable to open output file: %s\n", outname);
        exit(1);
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fwrite(&C[i * size + j], sizeof(double), 1, output);
        }
    }
    fclose(output);

    // Free allocated memory
    free(A);
    free(B);
    free(C);

    return 0;
}
```

## MatLab code for creating the input files

```
function create_matrices_and_store_in_file(size)
    % Generate random matrices A and B with elements between 100 and 999
    A = 100 + 899 * rand(size);
    B = 100 + 899 * rand(size);

    % Create the filename based on the size
    filename = sprintf('input%d.txt', size);

    % Open the file for writing
    fid = fopen(filename, 'w');

    % Check if the file opened successfully
    if fid == -1
        error('Cannot open file: %s', filename);
    end

    % Write the size of the matrices
    fprintf(fid, '%d\n', size);

    % Write matrix A row by row with specified format
    fprintf(fid, '%.3f ', A'); % Transpose to write row-wise, 3 decimal places

    % Add newline after matrix A
    fprintf(fid, '\n');

    % Write matrix B row by row with specified format
    fprintf(fid, '%.3f ', B');

    % Close the file
    fclose(fid);

    % Display the matrices (optional)
    disp('Matrix A:');
    disp(A);
    disp('Matrix B:');
    disp(B);
end
```

## Python code for creating the plots

```
import matplotlib.pyplot as plt
import numpy as np

# Combining both tables for power-of-2 and not power-of-2 matrices
matrix_sizes_np2 = [111, 222, 444, 555, 666]
times_1_np2 = [0.983, 6.619, 47.185, 92.342, 143.031]
times_8_np2 = [0.169, 1.186, 8.581, 17.730, 26.351]

matrix_sizes_p2 = [64, 128, 256, 512, 1024]
times_1_p2 = [0.126, 0.879, 6.143, 42.494, 302.135]
times_8_p2 = [0.022, 0.155, 1.080, 7.500, 53.000]

plt.figure(figsize=(10, 6))

plt.loglog(matrix_sizes_np2, times_1_np2, 'o-', color='blue', label='1 Process (Not
    Power of 2)')
plt.loglog(matrix_sizes_np2, times_8_np2, 's-', color='blue', label='8 Processes
    (Not Power of 2)')
plt.loglog(matrix_sizes_p2, times_1_p2, 'o-', color='orange', label='1 Process
    (Power of 2)')
plt.loglog(matrix_sizes_p2, times_8_p2, 's-', color='orange', label='8 Processes
    (Power of 2)')
# Theoretical complexity lines
n = np.linspace(10, 1000, 100) # Range for n
plt.loglog(n, 0.0000005 * n**2.807, '--', color='red', label=r'$\Theta(n^{2.807})$')
plt.loglog(n, 0.000005 * n**3, '--', color='green', label=r'$\Theta(n^3)$')

plt.xlabel('Matrix Size')
plt.ylabel('Time [s]')
plt.title('Matrix Multiplication Run Time vs. Size (Log-Log Scale)')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()

# Close to which theoretical complexity line
matrix_sizes = np.array([111, 222, 444, 555, 666])
times_1 = np.array([0.983, 6.619, 47.185, 92.342, 143.031])

plt.figure(figsize=(10, 6))
plt.loglog(matrix_sizes, times_1, 'o-', color='blue', label='Measurements (1
    process)')

# Theoretical complexity lines
n = np.linspace(111, 666, 100)
base_x = matrix_sizes[0]
base_y = times_1[0]
coeff_2807 = base_y / base_x**2.807
coeff_3 = base_y / base_x**3

# Theta(n^2.807) in red
plt.loglog(n, coeff_2807 * n**2.807, '-', color='red', label=r'$\Theta(n^{2.807})$
    (Adjusted)')
# Theta(n^3) in orange
plt.loglog(n, coeff_3 * n**3, '-', color='orange', label=r'$\Theta(n^3)$
    (Adjusted)')

plt.xlabel('Matrix Size')
plt.ylabel('Time [s]')
plt.title('Matrix Multiplication Run Time vs. Size (Log-Log Scale, Adjusted)')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```