UPPSALA
UNIVERSITET

High Performance Programming

# Individual Project
# Matrix-matrix multiplication with Strassen algorithm

Jinglin Gao

March 17, 2023

# 1 Introduction

In this report, a matrix-matrix multiplication calculator using the Strassen algorithm has been implemented in C. The Strassen algorithm is faster than the standard multiplication algorithm for larger matrices, as it has a time complexity of $\mathcal{O}(N^{2.8074})$. This algorithm was first published in 1969 and made the first move to prove that the general matrix multiplication algorithm with a time complexity of $n^3$ was not optimal. It is a very important algorithm, and it is interesting to build a matrix-matrix multiplication solver. Additionally, this report includes a parallel version of the computation code.

# 2 Problem Description

The Strassen multiplication algorithm works for square matrices and assumes that all matrices being multiplied have a size of $2^n$. During the implementation of the code, we can find the next power of 2 based on the given dimension and fill it with zeros to make it a size of $2^n$. The basic idea of the Strassen algorithm will be explained below[2].

First, divide the matrices into four submatrices and reduce their dimensions by half of the original size.

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

Instead of directly multiplying and adding matrices to obtain the new result, the Strassen algorithm defines new matrices that allow for a more efficient calculation method.

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22})B_{11}; \\
M_3 &= A_{11}(B_{12} - B_{22}); \\
M_4 &= A_{22}(B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12})B_{22}); \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}),
\end{aligned} \tag{2}$$

Using only 7 multiplications instead of 8 in the standard algorithm reduces the time complexity and makes it faster.

Finally, these 7 matrices are combined together to form the new result matrix, which only requires addition and subtraction operations.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}. \tag{3}$$

This is the whole concept of the Strassen algorithm. During implementation, we just need to recursively use the Strassen algorithm until it reduces to single-digit multiplication.

The aim of this project is to create a high-performance matrix multiplication calculator utilizing the Strassen algorithm. I plan to achieve this by first completing the serial Strassen function and ensuring its accuracy. I will then parallelize the algorithm and optimize it to enhance its performance. By doing so, we hope to achieve faster and more efficient matrix calculations.

# 3 Solution method

## Serial Algorithm

The algorithm begins by retrieving the dimension of the matrix from the command line. Based on the assumptions made by the Strassen algorithm, the code checks whether the input value is a power of 2 using the *check_matrix* function. If the value is a power of 2, a random matrix is generated with each value between 0 and 9.

However, if the input value is not a power of 2, the *next_power_of_2* function is used to determine the next valid number that will result in a matrix with dimensions that are a power of 2. Once this number is found, the *random_matrix* function is used again to create a random matrix. This time, any "missing" rows and columns are filled with zeros to obtain the correct matrix dimensions.

In summary, this process ensures that the input matrix has the required dimensions and contains valid data for the Strassen algorithm to process.

The main part of the computation is the *Strassen* function, which computes the multiplication of two matrices. First, we divide each matrix into four submatrices and then calculate the 7 new matrices recursively. In order to make the calculation much more simple we defined the *add_matrix* and *sub_matrix* as helper functions. Since each function call will create a new matrix and allocate memory for it, we need to free all the matrices we created. This function is quite straightforward and easy to understand. After calculating all 7 new matrices, we can combine them in a specific way to obtain the resulting matrix.

In the *main* function, we simply need to call the *strassen* function to perform the computation and verify the correctness of the result. We also use *omp_get_wtime* to measure the time spent on the multiplication part, but there is no parallel part in the serial version of the Strassen algorithm.

## Parallelization

In this section, we parallelized the codes above using OpenMP to speed up the Strassen algorithm's recursive computations. As previously discussed, each calculation of a new matrix can be executed independently, making it a perfect candidate for parallelization. We can use task parallelism to achieve the parallelization part. We utilized the OpenMP directives *omp parallel*, *omp single*, and *omp task* to parallelize the computations of the seven new matrices. Specifically, we assigned a separate task for each computation of the seven new matrices.

In addition to using *omp tasks*, I also experimented with *omp sections*. While the two approaches work in a similar way, there are some differences. Specifically, *omp sections* schedules sections of code statically to threads and runs them in parallel. This can be useful in cases where the workload is evenly distributed across parallel tasks[1].

In the context of the *strassen* function, each parallel task requires almost the same amount of work. This means that there is theoretically no load balancing problem to worry about. As a result, we did not assign a different number of threads to each task. Instead, we relied on the default scheduling behavior of *omp sections* to evenly distribute the workload among available threads.

In addition to parallelizing the *strassen* algorithm, I attempted to optimize the *add_matrix*, *sub_matrix*, and *divide_matrix* functions by parallelizing them with the *# pragma omp parallel for collapse*(2) directive. However, this optimization did not yield the expected performance gains. In fact, the parallelization of these functions resulted in a significant slowdown. As a result, I ultimately decided to abandon this approach.

## Optimization

### Compiler optimisation

There are numerous optimization options available for the compiler. Some optimization flags have been experimented with on different platforms. The optimization flags used and tested during the evaluation are introduced in Table 1.

Table 1: Compiler Optimization Flags and Performance

| Flag | Description |
|---|---|
| -O1 | Basic optimization |
| -O2 | Standard optimization |
| -O3 | Full optimization |
| -Ofast | Aggressive optimization |
| -march=native | Optimize for native CPU |
| -ffast-math | Increase speed of math operations |

### Code optimisation

- Use bit-wise operations:
  Instead of using the multiplication and division operators, use bit-wise operations to calculate the next power of 2. For example, instead of $i = i * 2$, use $i <<= 1$.
- Loop unrolling:
  In the functions "$add_matrix$" and "$sub_matrix$", I have implemented loop unrolling as a performance optimization technique. Given our assumption that all matrices have a size that is a power of 2, it is straightforward to unroll the loop for a more efficient computation. By unrolling the loop, we can reduce the number of iterations required and minimize the overhead associated with loop control. This can significantly enhance the speed of our code and improve its overall performance.

# 4 Experiments

## Correctness of code

Ensuring the correctness of code involves two main aspects. The first aspect is verifying its time complexity by analyzing the algorithm used and identifying any potential bottlenecks. The second aspect entails validating the accuracy of the computed result matrix to ensure the code produces the expected output and eliminates any errors that may arise.

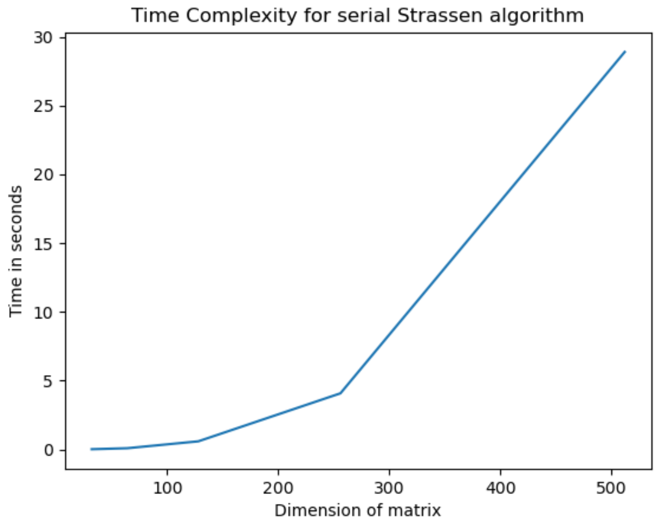


Figure 1: Measured execution time for Strassen algorithm

Through a careful analysis of the algorithm's execution time, we have determined that its time complexity is approximate $\mathcal{O}(N^{2.8})$, which closely aligns with our initial expectations.

I have also developed a naive function to verify the accuracy of our Strassen algorithm. For testing purposes, we have chosen matrices of different sizes, including both sizes that are powers of 2 and those that are not.

I checked the sizes of 2, 35, 64, 100, 128, and 500, and the results showed that the Strassen algorithm correctly calculated the matrices.

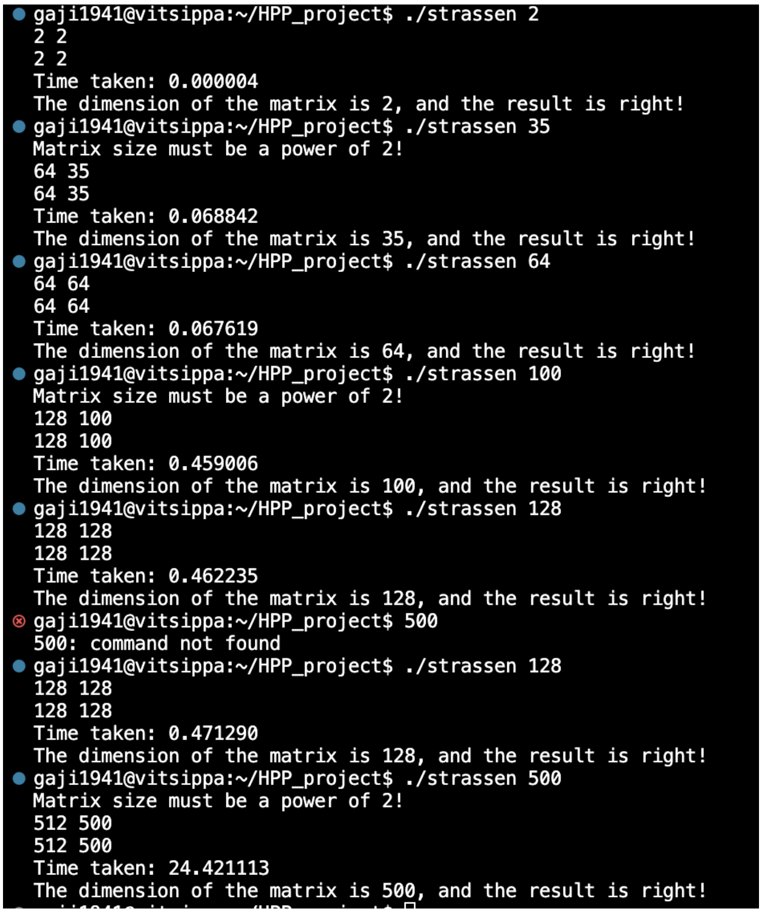


Figure 2: Check correctness of the Strassen algorithm

## Evaluation Performance

### Configuration

The execution times are measured using a Linux virtual machine, and its specifications are listed in Table 2 below. The following optimization experiments are mostly conducted on this machine, except for the best timing test.

Table 2: Server Specifications

| **Component** | **Specification** |
|---|---|
| CPU | AMD Opteron (Bulldozer) 6282SE, 2.6 GHz, 16-cores, dual socket |
| Memory | 128 GB |
| Operating System | Ubuntu 22.04 |
| Compiler Version | gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0 |
| Server Name | vitsippa.it.uu.se |

In order to improve the performance of our code, we employ compiler optimization techniques to identify potential areas for enhancement. We then evaluate the effectiveness of these optimizations by measuring the time required to execute the program under different optimization levels. The results of these tests are presented in the figure below, which shows the elapsed time for each optimization level.

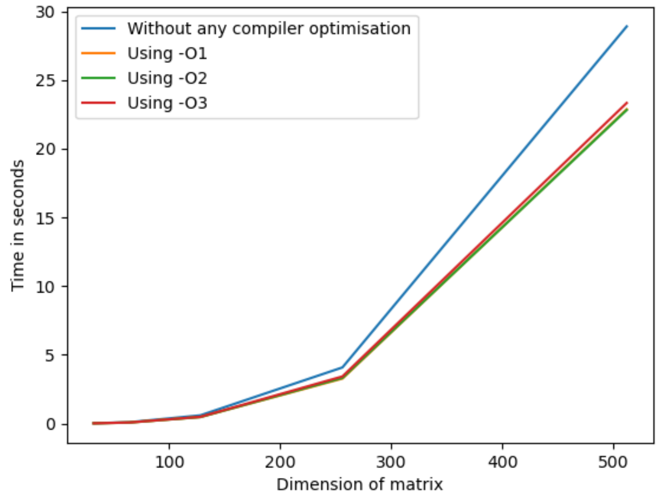


Figure 3: Different compiler optimization techniques performance

By analyzing the figure we obtained above, it is clear that all three compiler flags result in a performance improvement. However, compared to -O3, -O1 and -O2 seem more suitable for the problem. In addition, I also tried –ffast-math, -march=native, and -Ofast flags, which did not appear in the figure. –ffast-math actually provides a similar improvement to -O2, but the other two flags do not significantly improve performance.

-O3 may not have improved performance because it enables additional optimization options that can increase the executable size. When the number of instructions is too high, the instruction cache missing rate may increase, which can negatively affect the program's performance.

After parallelizing the Strassen algorithm, I compared the performance of the parallelized Strassen algorithm to the serial Strassen algorithm. The figure below shows the speedup achieved with the number of threads on the x-axis and the achieved speedup on the y-axis for the experiment with matrix sizes of 512 and 1024, respectively.

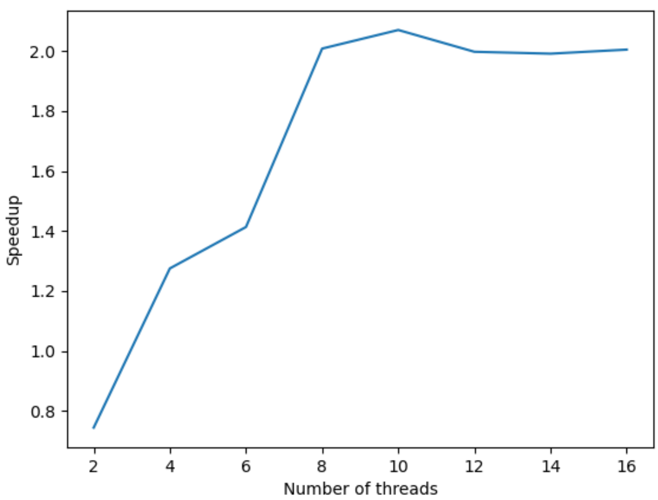


Figure 4: Speed up using different number of threads (n=512)

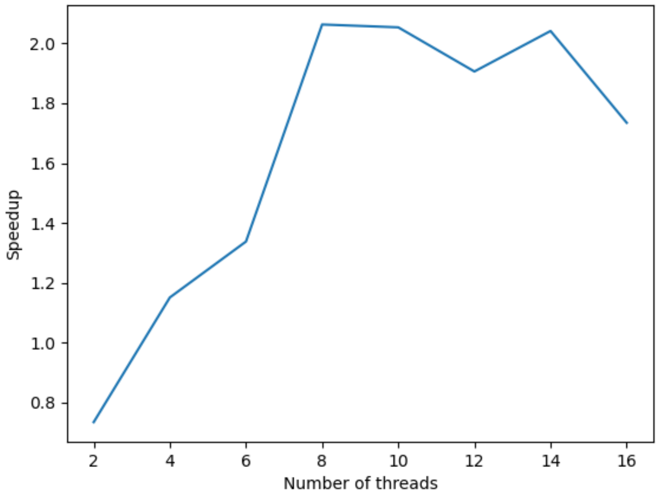


Figure 5: Speed up using different number of threads (n=1024)

As it shows, the maximum speedup I achieved is about 2 times, however I expected it to reach 8 times speedup.

There are several reasons why this may happen. First, the program's performance is limited by memory operations. Second, there are too many memory allocations throughout the program, which may limit the performance due to the available memory bandwidth. Third, parallelizing the algorithm may increase the amount of data that needs to be transferred between the CPU and memory, leading to cache thrashing and other performance issues. Finally, the program's structure may not be optimized for parallelization, as there are numerous function calls within the *strassen* function. Removing these calls and performing the necessary additions and subtractions directly within the *strassen* function could simplify parallelization and potentially improve performance.

# 5 Conclusions

In this project, I have implemented a matrix multiplication calculator using the Strassen algorithm. After successfully implementing the serial algorithm and verifying the accuracy of the program, I aimed to optimize it by parallelizing the code using two different methods.

After rigorous testing, I discovered that using 8 cores produced the best performance, providing a remarkable speedup of around 2 times. However, it is important to note that this result is not conclusive, and there may be other ways to optimize the entire method further.

One potential bottleneck in the program is memory allocation, which can slow down the program. Therefore, it is crucial to ensure that memory allocation is optimized to ensure that the program runs as efficiently as possible.

In addition, when dealing with 2D arrays, such as matrices in this case, cache performance is an essential consideration. By reducing jumps between rows, we can lower the cache miss rate, which can significantly improve the program's performance.

Overall, this project can correctly perform matrix multiplication using the Strassen algorithm, and with parallelization, the computation time can be reduced by approximately 2 times. However, there are still potential optimizations that can be made, such as optimizing memory allocation and reducing cache misses, to further enhance the program's performance.

# References


[1] Nwe Zin Oo and Panyayot Chaikan. Power efficient strassen's algorithm using avx512 and openmp in a multi-core architecture. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 17(1):46–59, Jan. 2023.

[2] Wikipedia. Strassen algorithm — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Strassen%20algorithm&oldid=1144122394`, 2023. [Online; accessed 17-March-2023].

# Appendix

## Serial Strassen Algorithm

```

# include <stdio.h>
# include <stdlib.h>
#include <stdbool.h>
# include <omp.h>

int check_matrix(int N) {
    if (N <= 0) {
        printf("Matrix size must be larger than 0!");
        return 0;
    }

    // check if the matrix size is a power of 2
    while( N != 1) {
        if (N % 2 != 0) {
            printf("Matrix size must be a power of 2!\n");
            return -1;
        }
        N = N / 2;
    }
    return 1;
}

// find the next power of 2
int next_power_of_2(int N) {
    int i = 1;
    while (i < N) {
        i <<= 1;
    }
    return i;
}

// allocate memory for a matrix
int ** create_matrix(int n) {
    int *data = (int *)malloc(n * n * sizeof(int));
    int **array = (int **)malloc(n * sizeof(int *));
    // check if the memory is allocated
    if (data == NULL || array == NULL) {
        printf("Matrix memory allocation failed!");
        exit(-1);
    }

    for (int i = 0; i < n; i++) {
        array[i] = &(data[i * n]);
    }
    return array;
}

// randomly generate a matrix
void random_matrix(int n, int N, int **A) {
    int i, j;
    printf("%d %d\n", n, N);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = rand() % 10; // generate a random number between 0 and 9
        }
    }

    if (n != N) {
        for (i = N; i < n; i++) {
            for (j = N; j < n; j++) {
                A[i][j] = 0;
            }
        }
    }
}

// function to print the matrix
void print_matrix(int N, int **A) {
    int i, j;
```

```
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%d\t", A[i][j]);
        }
        printf("\n");
    }
}

int ** add_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j = j + 2) {
            result[i][j] = A[i][j] + B[i][j];
            result[i][j+1] = A[i][j+1] + B[i][j+1];
        }
    }
    return result;
}

int ** sub_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j = j + 2) {
            result[i][j] = A[i][j] - B[i][j];
            result[i][j+1] = A[i][j+1] - B[i][j+1];
        }
    }
    return result;
}

// divide the matrix into 4 sub-matrixs
int ** divide_matrix(int n, int **A, int i, int j) {
    int **result;
    result = create_matrix(n / 2);
    int x, y;
    for (x = 0; x < n / 2; x++) {
        for (y = 0; y < n / 2; y++) {
            result[x][y] = A[x + i][y + j];
        }
    }
    return result;
}

// a naive function to calculate the result
int **naive_calculate(int n, int **A, int**B) {
    int **result;
    result = create_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j=0; j < n; j++) {
            result[i][j] = 0;
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

// check accuracy of the algorithm
bool check_accuracy(int n, int **mat1, int **mat2) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (mat1[i][j] != mat2[i][j]) {
                return false;
            }
        }
    }
    return true;
}

int ** strassen(int n, int **A, int **B) {
    int **C;
```

```c
    C = create_matrix(n);
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return C;
    }

    // decline the matrix demension
    int m = n / 2;
    // printf("m = %d\n", m);
    int **A11 = divide_matrix(n, A, 0, 0);
    int **A12 = divide_matrix(n, A, 0, m);
    int **A21 = divide_matrix(n, A, m, 0);
    int **A22 = divide_matrix(n, A, m, m);
    int **B11 = divide_matrix(n, B, 0, 0);
    int **B12 = divide_matrix(n, B, 0, m);
    int **B21 = divide_matrix(n, B, m, 0);
    int **B22 = divide_matrix(n, B, m, m);

    // calculate the 7 sub-matrixs
    int **add1 = add_matrix(m, A11, A22);
    int **add2 = add_matrix(m, B11, B22);
    int **M1 = strassen(m, add1, add2);
    free(add1);
    free(add2);

    int **add3 = add_matrix(m, A21, A22);
    int **M2 = strassen(m, add3, B11);
    free(add3);

    int **sub1 = sub_matrix(m, B12, B22);
    int **M3 = strassen(m, A11, sub1);
    free(sub1);

    int **sub2 = sub_matrix(m, B21, B11);
    int **M4 = strassen(m, A22, sub2);
    free(sub2);

    int **add4 = add_matrix(m, A11, A12);
    int **M5 = strassen(m, add4, B22);
    free(add4);

    int **sub3 = sub_matrix(m, A21, A11);
    int **add5 = add_matrix(m, B11, B12);
    int **M6 = strassen(m, sub3, add5);
    free(sub3);
    free(add5);

    int **sub4 = sub_matrix(m, A12, A22);
    int **add6 = add_matrix(m, B21, B22);
    int **M7 = strassen(m, sub4, add6);
    free(sub4);
    free(add6);

    // calculate the 4 sub-matrixs of the result matrix
    int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
    int **C12 = add_matrix(m, M3, M5);
    int **C21 = add_matrix(m, M2, M4);
    int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);

    // combine the 4 sub-matrixs into a matrix
    int i, j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < m; j++) {
            C[i][j] = C11[i][j];
            C[i][j + m] = C12[i][j];
            C[i + m][j] = C21[i][j];
            C[i + m][j + m] = C22[i][j];
        }
    }

    free(A11);
    free(A12);
    free(A21);
    free(A22);
    free(B11);
    free(B12);
```

```
    free(B21);
    free(B22);
    free(M1);
    free(M2);
    free(M3);
    free(M4);
    free(M5);
    free(M6);
    free(M7);
    free(C11);
    free(C12);
    free(C21);
    free(C22);

    return C;
}

int main(int argc, char *argv[]) {
    // check the number of arguments
    if (argc != 2) {
        printf("Usage: %s <matrix size>", argv[0]);
        return -1;
    }

    // create the matrixs to be multiplied
    int N = atoi(argv[1]);
    int n = N;

    // check the matrix size
    if (check_matrix(N) == -1) {
        n = next_power_of_2(N);
    }

    int **A = create_matrix(n); // should return a array of pointers
    random_matrix(n, N, A);

    int **B = create_matrix(n);
    random_matrix(n, N, B);

    double starttime = omp_get_wtime();

    int **C = strassen(n, A, B);

    double endtime = omp_get_wtime();

    printf("Time taken: %lf\n", endtime - starttime);

    int **result = naive_calculate(n, A, B);

    // check the accuracy of the Strassen algorithm
    if (check_accuracy) {
        printf("The dimension of the matrix is %d, and the result is right!\n", N);
    }


    free(A);
    free(B);
    free(C);
    return 0;
}
```

## Openmp Version 1

```c
# include <stdio.h>
# include <stdlib.h>
# include <stdbool.h>
# include <omp.h>

int check_matrix(int N) {
    if (N <= 0) {
        printf("Matrix size must be larger than 0!");
        return 0;
    }

    // check if the matrix size is a power of 2
    while( N != 1) {
        if (N % 2 != 0) {
            printf("Matrix size must be a power of 2!\n");
            return -1;
        }
        N = N / 2;
    }
    return 1;
}

// find the next power of 2
int next_power_of_2(int N) {
    int i = 1;
    while (i < N) {
        i <<= 1;
    }
    return i;
}

// allocate memory for a matrix
int ** create_matrix(int n) {
    int *data = (int *)malloc(n * n * sizeof(int));
    int **array = (int **)malloc(n * sizeof(int *));
    // check if the memory is allocated
    if (data == NULL || array == NULL) {
        printf("Matrix memory allocation failed!");
        exit(-1);
    }

    for (int i = 0; i < n; i++) {
        array[i] = &(data[i * n]);
    }
    return array;
}

// randomly generate a matrix
void random_matrix(int n, int N, int **A) {
    int i, j;
    // printf("%d %d\n", n, N);
    #pragma omp parallel for collapse(2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = rand() % 10; // generate a random number between 0 and 9
        }
    }

    if (n != N) {
        for (i = N; i < n; i++) {
            for (j = N; j < n; j++) {
                A[i][j] = 0;
            }
        }
    }
}

// function to print the matrix
void print_matrix(int N, int **A) {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%d\t", A[i][j]);
        }
```

```c
        printf("\n");
    }
}

int ** add_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    // #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    return result;
}

int ** sub_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    // #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
    return result;
}

// divide the matrix into 4 sub-matrixs
int ** divide_matrix(int n, int **A, int i, int j) {
    int **result;
    result = create_matrix(n / 2);
    int x, y;
    // #pragma omp parallel for collapse(2)
    for (x = 0; x < n / 2; x++) {
        for (y = 0; y < n / 2; y++) {
            result[x][y] = A[x + i][y + j];
        }
    }
    return result;
}

// a naive function to calculate the result
int **naive_calculate(int n, int **A, int**B) {
    int **result;
    result = create_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j=0; j < n; j++) {
            result[i][j] = 0;
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

// check accuracy of the algorithm
bool check_accuracy(int n, int **mat1, int **mat2) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (mat1[i][j] != mat2[i][j]) {
                return false;
            }
        }
    }
    return true;
}

int ** strassen(int n, int **A, int **B) {
    int **C;
    C = create_matrix(n);
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
```

```c
        return C;
    }

    // decline the matrix demension
    int m = n / 2;
    // printf("m = %d\n", m);
    int **A11 = divide_matrix(n, A, 0, 0);
    int **A12 = divide_matrix(n, A, 0, m);
    int **A21 = divide_matrix(n, A, m, 0);
    int **A22 = divide_matrix(n, A, m, m);
    int **B11 = divide_matrix(n, B, 0, 0);
    int **B12 = divide_matrix(n, B, 0, m);
    int **B21 = divide_matrix(n, B, m, 0);
    int **B22 = divide_matrix(n, B, m, m);

    int **M1;
    int **M2;
    int **M3;
    int **M4;
    int **M5;
    int **M6;
    int **M7;

    // calculate the 7 sub-matrixs
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        {
            // int **M1;
            #pragma omp task
            {
            int **add1 = add_matrix(m, A11, A22);
            int **add2 = add_matrix(m, B11, B22);
            M1 = strassen(m, add1, add2);
            free(add1);
            free(add2);
            }
            // int **M2;
            #pragma omp task
            {
                int **add3 = add_matrix(m, A21, A22);
                M2 = strassen(m, add3, B11);
                free(add3);
            }
            // int **M3;
            #pragma omp task
            {
                int **sub1 = sub_matrix(m, B12, B22);
                M3 = strassen(m, A11, sub_matrix(m, B12, B22));
                free(sub1);
            }
            // int **M4;
            #pragma omp task
            {
                int **sub2 = sub_matrix(m, B21, B11);
                M4 = strassen(m, A22, sub2);
                free(sub2);
            }
            // int **M5;
            #pragma omp task
            {
                int **add4 = add_matrix(m, A11, A12);
                M5 = strassen(m, add4, B22);
                free(add4);
            }
            // int **M6;
            #pragma omp task
            {
                int **sub3 = sub_matrix(m, A21, A11);
                int **add5 = add_matrix(m, B11, B12);
                M6 = strassen(m, sub3, add5);
                free(sub3);
                free(add5);
            }
            // int **M7;
            #pragma omp task
```

```
            {
                int **sub4 = sub_matrix(m, A12, A22);
                int **add6 = add_matrix(m, B21, B22);
                M7 = strassen(m, sub4, add6);
                free(sub4);
                free(add6);
            }

        }

        // calculate the 4 sub-matrixs of the result matrix
        #pragma omp taskwait
        int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
        int **C12 = add_matrix(m, M3, M5);
        int **C21 = add_matrix(m, M2, M4);
        int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);

        // combine the 4 sub-matrixs to get the final result matrix
        #pragma omp taskwait
        int i, j;
        for (i = 0; i < m; i++) {
            for (j = 0; j < m; j++) {
                C[i][j] = C11[i][j];
                C[i][j + m] = C12[i][j];
                C[i + m][j] = C21[i][j];
                C[i + m][j + m] = C22[i][j];
        }
    }

    }

    // #pragma omp taskwait

    free(A11);
    free(A12);
    free(A21);
    free(A22);
    free(B11);
    free(B12);
    free(B21);
    free(B22);
    free(M1);
    free(M2);
    free(M3);
    free(M4);
    free(M5);
    free(M6);
    free(M7);

    return C;
}

int main(int argc, char *argv[]) {
    // check the number of arguments
    if (argc != 2) {
        printf("Usage: %s <matrix size>", argv[0]);
        return -1;
    }

    // create the matrixs to be multiplied
    int N = atoi(argv[1]);
    int n = N;

    // check the matrix size
    if (check_matrix(N) == -1) {
        n = next_power_of_2(N);
    }

    int **A = create_matrix(n); // should return a array of pointers
    random_matrix(n, N, A);

    int **B = create_matrix(n);
    random_matrix(n, N, B);

    int **C;

```

```c
    double starttime = omp_get_wtime();
    omp_set_num_threads(8);

    C = strassen(n, A, B);

    double endtime = omp_get_wtime();
    printf("Time taken: %lf\n", endtime - starttime);
    // print_matrix(N, C);

    // int **result = naive_calculate(n, A, B);

    // // check the accuracy of the Strassen algorithm
    // if (check_accuracy) {
    //     printf("The dimension of the matrix is %d, and the result is right!\n", N);
    // }

    free(A);
    free(B);
    free(C);
    return 0;
}
```

## OpenMP Version 2

```c
# include <stdio.h>
# include <stdlib.h>
# include <stdbool.h>
# include <omp.h>

int check_matrix(int N) {
    if (N <= 0) {
        printf("Matrix size must be larger than 0!");
        return 0;
    }

    // check if the matrix size is a power of 2
    while( N != 1) {
        if (N % 2 != 0) {
            printf("Matrix size must be a power of 2!\n");
            return -1;
        }
        N = N / 2;
    }
    return 1;
}

// find the next power of 2
int next_power_of_2(int N) {
    int i = 1;
    while (i < N) {
        i <<= 1;
    }
    return i;
}

// allocate memory for a matrix
int ** create_matrix(int n) {
    int *data = (int *)malloc(n * n * sizeof(int));
    int **array = (int **)malloc(n * sizeof(int *));
    // check if the memory is allocated
    if (data == NULL || array == NULL) {
        printf("Matrix memory allocation failed!");
        exit(-1);
    }

    for (int i = 0; i < n; i++) {
        array[i] = &(data[i * n]);
    }
    return array;
}

// randomly generate a matrix
void random_matrix(int n, int N, int **A) {
    int i, j;
    // printf("%d %d\n", n, N);
    #pragma omp parallel for collapse(2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = rand() % 10; // generate a random number between 0 and 9
        }
    }

    if (n != N) {
        for (i = N; i < n; i++) {
            for (j = N; j < n; j++) {
                A[i][j] = 0;
            }
        }
    }
}

// function to print the matrix
void print_matrix(int N, int **A) {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%d\t", A[i][j]);
        }
```

```c
        printf("\n");
    }
}

int ** add_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    // #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    return result;
}

int ** sub_matrix(int n, int **A, int **B) {
    int i, j;
    int **result;
    result = create_matrix(n);
    // #pragma omp parallel for collapse(2)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }
    return result;
}

// divide the matrix into 4 sub-matrixs
int ** divide_matrix(int n, int **A, int i, int j) {
    int **result;
    result = create_matrix(n / 2);
    int x, y;
    // #pragma omp parallel for collapse(2)
    for (x = 0; x < n / 2; x++) {
        for (y = 0; y < n / 2; y++) {
            result[x][y] = A[x + i][y + j];
        }
    }
    return result;
}

// a naive function to calculate the result
int **naive_calculate(int n, int **A, int**B) {
    int **result;
    result = create_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j=0; j < n; j++) {
            result[i][j] = 0;
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

// check accuracy of the algorithm
bool check_accuracy(int n, int **mat1, int **mat2) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (mat1[i][j] != mat2[i][j]) {
                return false;
            }
        }
    }
    return true;
}

int ** strassen(int n, int **A, int **B) {
    int **C;
    C = create_matrix(n);
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
```

```c
        return C;
    }

    // decline the matrix demension
    int m = n / 2;
    // printf("m = %d\n", m);
    int **A11 = divide_matrix(n, A, 0, 0);
    int **A12 = divide_matrix(n, A, 0, m);
    int **A21 = divide_matrix(n, A, m, 0);
    int **A22 = divide_matrix(n, A, m, m);
    int **B11 = divide_matrix(n, B, 0, 0);
    int **B12 = divide_matrix(n, B, 0, m);
    int **B21 = divide_matrix(n, B, m, 0);
    int **B22 = divide_matrix(n, B, m, m);

    int **M1;
    int **M2;
    int **M3;
    int **M4;
    int **M5;
    int **M6;
    int **M7;

    // calculate the 7 sub-matrixs
    #pragma omp parallel num_threads(8)
    {
        #pragma omp sections
        {
            // int **M1;
            #pragma omp section
            {
            int **add1 = add_matrix(m, A11, A22);
            int **add2 = add_matrix(m, B11, B22);
            M1 = strassen(m, add1, add2);
            free(add1);
            free(add2);
            }
            // int **M2;
            #pragma omp section
            {
                int **add3 = add_matrix(m, A21, A22);
                M2 = strassen(m, add3, B11);
                free(add3);
            }
            // int **M3;
            #pragma omp section
            {
                int **sub1 = sub_matrix(m, B12, B22);
                M3 = strassen(m, A11, sub_matrix(m, B12, B22));
                free(sub1);
            }
            // int **M4;
            #pragma omp section
            {
                int **sub2 = sub_matrix(m, B21, B11);
                M4 = strassen(m, A22, sub2);
                free(sub2);
            }
            // int **M5;
            #pragma omp section
            {
                int **add4 = add_matrix(m, A11, A12);
                M5 = strassen(m, add4, B22);
                free(add4);
            }
            // int **M6;
            #pragma omp section
            {
                int **sub3 = sub_matrix(m, A21, A11);
                int **add5 = add_matrix(m, B11, B12);
                M6 = strassen(m, sub3, add5);
                free(sub3);
                free(add5);
            }
            // int **M7;
            #pragma omp section
```

```
            {
                int **sub4 = sub_matrix(m, A12, A22);
                int **add6 = add_matrix(m, B21, B22);
                M7 = strassen(m, sub4, add6);
                free(sub4);
                free(add6);
            }

        }
    }


        int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
        int **C12 = add_matrix(m, M3, M5);
        int **C21 = add_matrix(m, M2, M4);
        int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);

        // combine the 4 sub-matrixs to get the final result matrix
        // #pragma omp taskwait
        int i, j;
    #pragma omp parallel for collapse(2)
        for (i = 0; i < m; i++) {
            for (j = 0; j < m; j++) {
                C[i][j] = C11[i][j];
                C[i][j + m] = C12[i][j];
                C[i + m][j] = C21[i][j];
                C[i + m][j + m] = C22[i][j];
        }
    }

    // }

    // #pragma omp taskwait

    free(A11);
    free(A12);
    free(A21);
    free(A22);
    free(B11);
    free(B12);
    free(B21);
    free(B22);
    free(M1);
    free(M2);
    free(M3);
    free(M4);
    free(M5);
    free(M6);
    free(M7);

    return C;
}

int main(int argc, char *argv[]) {
    // check the number of arguments
    if (argc != 2) {
        printf("Usage: %s <matrix size>", argv[0]);
        return -1;
    }

    // create the matrixs to be multiplied
    int N = atoi(argv[1]);
    int n = N;

    // check the matrix size
    if (check_matrix(N) == -1) {
        n = next_power_of_2(N);
    }

    int **A = create_matrix(n); // should return a array of pointers
    random_matrix(n, N, A);
    // printf("Matrix A:\n");
    // print_matrix(n, A);
    // printf("\n");

    int **B = create_matrix(n);
```

```c
    random_matrix(n, N, B);
    // print_matrix(n, B);
    // printf("\n");

    int **C;

    double starttime = omp_get_wtime();
    omp_set_num_threads(8);

    C = strassen(n, A, B);

    double endtime = omp_get_wtime();
    printf("Time taken: %lf\n", endtime - starttime);
    // print_matrix(N, C);

    int **result = naive_calculate(n, A, B);

    // check the accuracy of the Strassen algorithm
    if (check_accuracy) {
        printf("The dimension of the matrix is %d, and the result is right!\n", N);
    }

    free(A);
    free(B);
    free(C);
    return 0;
}
```