



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.

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

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.

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

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.

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

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.

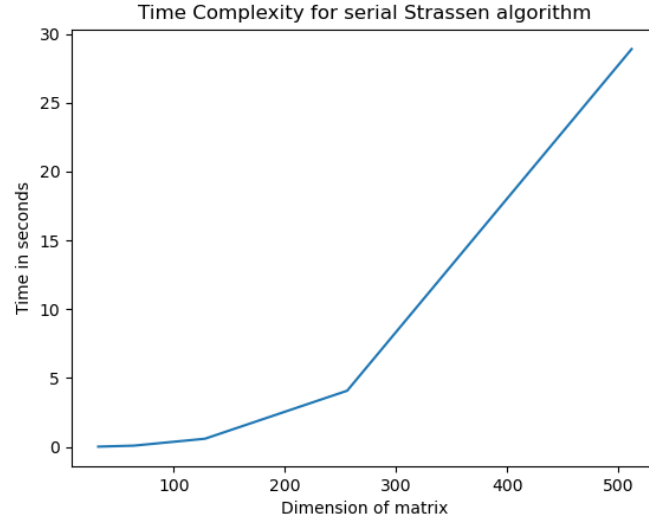


Figure 1: Measured execution time for Strassen algorithm

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.

```

● gaj11941@vitsippa:~/HPP_project$ ./strassen 2
2 2
2 2
Time taken: 0.000004
The dimension of the matrix is 2, and the result is right!
● gaj11941@vitsippa:~/HPP_project$ ./strassen 35
64 35
64 35
Time taken: 0.068842
The dimension of the matrix is 35, and the result is right!
● gaj11941@vitsippa:~/HPP_project$ ./strassen 64
64 64
64 64
Time taken: 0.067619
The dimension of the matrix is 64, and the result is right!
● gaj11941@vitsippa:~/HPP_project$ ./strassen 100
Matrix size must be a power of 2!
128 100
128 100
Time taken: 0.459006
The dimension of the matrix is 100, and the result is right!
● gaj11941@vitsippa:~/HPP_project$ ./strassen 128
128 128
128 128
Time taken: 0.462235
The dimension of the matrix is 128, and the result is right!
⊗ gaj11941@vitsippa:~/HPP_project$ 500
500: command not found
● gaj11941@vitsippa:~/HPP_project$ ./strassen 128
128 128
128 128
Time taken: 0.471290
The dimension of the matrix is 128, and the result is right!
● gaj11941@vitsippa:~/HPP_project$ ./strassen 500
Matrix size must be a power of 2!
512 500
512 500
Time taken: 24.421113
The dimension of the matrix is 500, and the result is right!

```

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.

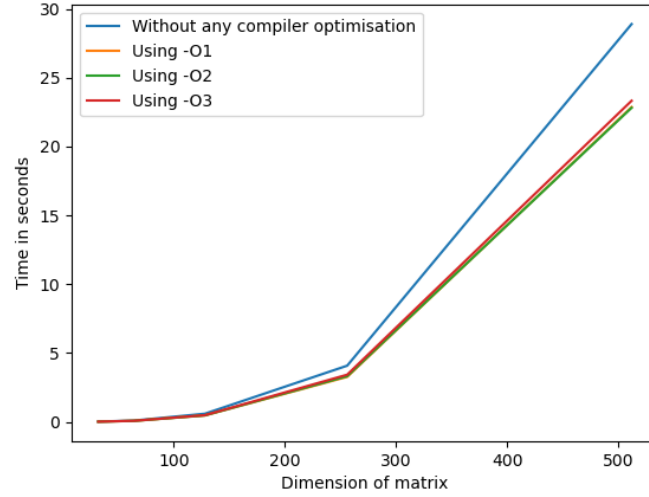


Figure 3: Different compiler optimization techniques performance

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 `-fast-math`, `-march=native`, and `-Ofast` flags, which did not appear in the figure. `-fast-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.

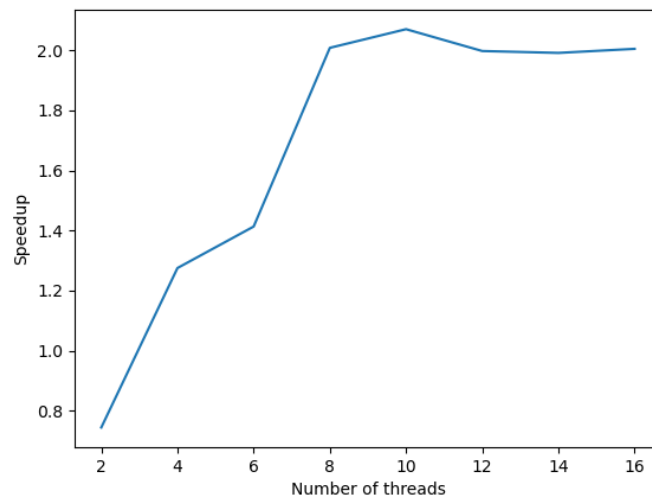


Figure 4: Speed up using different number of threads (n=512)

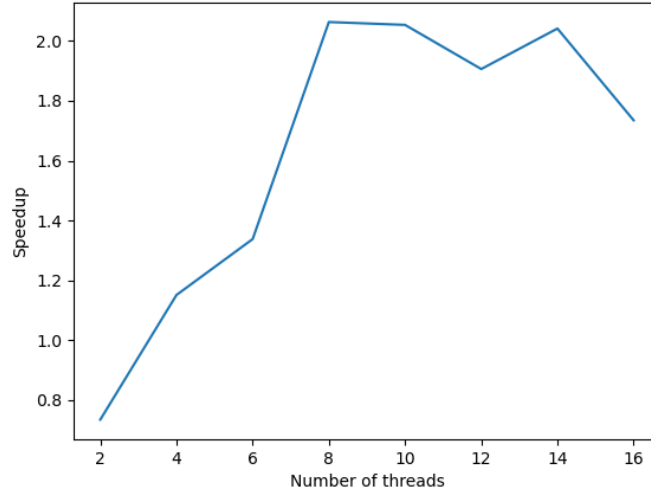


Figure 5: Speed up using different number of threads (n=1024)

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

```
1
2 # include <stdio.h>
3 # include <stdlib.h>
4 #include <stdbool.h>
5 # include <omp.h>
6
7 int check_matrix(int N) {
8     if (N <= 0) {
9         printf("Matrix size must be larger than 0!");
10        return 0;
11    }
12
13    // check if the matrix size is a power of 2
14    while( N != 1) {
15        if (N % 2 != 0) {
16            printf("Matrix size must be a power of 2!\n");
17            return -1;
18        }
19        N = N / 2;
20    }
21    return 1;
22 }
23
24 // find the next power of 2
25 int next_power_of_2(int N) {
26     int i = 1;
27     while (i < N) {
28         i <<= 1;
29     }
30     return i;
31 }
32
33 // allocate memory for a matrix
34 int ** create_matrix(int n) {
35     int *data = (int *)malloc(n * n * sizeof(int));
36     int **array = (int **)malloc(n * sizeof(int *));
37     // check if the memory is allocated
38     if (data == NULL || array == NULL) {
39         printf("Matrix memory allocation failed!");
40         exit(-1);
41     }
42
43     for (int i = 0; i < n; i++) {
44         array[i] = &(data[i * n]);
45     }
46     return array;
47 }
48
49 // randomly generate a matrix
50 void random_matrix(int n, int N, int **A) {
51     int i, j;
52     printf("%d %d\n", n, N);
53     for (i = 0; i < N; i++) {
54         for (j = 0; j < N; j++) {
55             A[i][j] = rand() % 10; // generate a random number between 0 and 9
56         }
57     }
58
59     if (n != N) {
60         for (i = N; i < n; i++) {
61             for (j = N; j < n; j++) {
62                 A[i][j] = 0;
63             }
64         }
65     }
66 }
67
68 // function to print the matrix
69 void print_matrix(int N, int **A) {
70     int i, j;
```

```

71     for (i = 0; i < N; i++) {
72         for (j = 0; j < N; j++) {
73             printf("%d\t", A[i][j]);
74         }
75         printf("\n");
76     }
77 }
78
79 int ** add_matrix(int n, int **A, int **B) {
80     int i, j;
81     int **result;
82     result = create_matrix(n);
83     for (i = 0; i < n; i++) {
84         for (j = 0; j < n; j = j + 2) {
85             result[i][j] = A[i][j] + B[i][j];
86             result[i][j+1] = A[i][j+1] + B[i][j+1];
87         }
88     }
89     return result;
90 }
91
92 int ** sub_matrix(int n, int **A, int **B) {
93     int i, j;
94     int **result;
95     result = create_matrix(n);
96     for (i = 0; i < n; i++) {
97         for (j = 0; j < n; j = j + 2) {
98             result[i][j] = A[i][j] - B[i][j];
99             result[i][j+1] = A[i][j+1] - B[i][j+1];
100        }
101    }
102    return result;
103 }
104
105 // divide the matrix into 4 sub-matrixs
106 int ** divide_matrix(int n, int **A, int i, int j) {
107     int **result;
108     result = create_matrix(n / 2);
109     int x, y;
110     for (x = 0; x < n / 2; x++) {
111         for (y = 0; y < n / 2; y++) {
112             result[x][y] = A[x + i][y + j];
113         }
114     }
115     return result;
116 }
117
118 // a naive function to calculate the result
119 int **naive_calculate(int n, int **A, int **B) {
120     int **result;
121     result = create_matrix(n);
122     for (int i = 0; i < n; i++) {
123         for (int j=0; j < n; j++) {
124             result[i][j] = 0;
125             for (int k = 0; k < n; k++) {
126                 result[i][j] += A[i][k] * B[k][j];
127             }
128         }
129     }
130     return result;
131 }
132
133 // check accuracy of the algorithm
134 bool check_accuracy(int n, int **mat1, int **mat2) {
135     for (int i = 0; i < n; i++) {
136         for (int j = 0; j < n; j++) {
137             if (mat1[i][j] != mat2[i][j]) {
138                 return false;
139             }
140         }
141     }
142     return true;
143 }
144
145 int ** strassen(int n, int **A, int **B) {
146     int **C;

```

```

147     C = create_matrix(n);
148     if (n == 1) {
149         C[0][0] = A[0][0] * B[0][0];
150         return C;
151     }
152
153     // decline the matrix demension
154     int m = n / 2;
155     // printf("m = %d\n", m);
156     int **A11 = divide_matrix(n, A, 0, 0);
157     int **A12 = divide_matrix(n, A, 0, m);
158     int **A21 = divide_matrix(n, A, m, 0);
159     int **A22 = divide_matrix(n, A, m, m);
160     int **B11 = divide_matrix(n, B, 0, 0);
161     int **B12 = divide_matrix(n, B, 0, m);
162     int **B21 = divide_matrix(n, B, m, 0);
163     int **B22 = divide_matrix(n, B, m, m);
164
165     // calculate the 7 sub-matrixs
166     int **add1 = add_matrix(m, A11, A22);
167     int **add2 = add_matrix(m, B11, B22);
168     int **M1 = strassen(m, add1, add2);
169     free(add1);
170     free(add2);
171
172     int **add3 = add_matrix(m, A21, A22);
173     int **M2 = strassen(m, add3, B11);
174     free(add3);
175
176     int **sub1 = sub_matrix(m, B12, B22);
177     int **M3 = strassen(m, A11, sub1);
178     free(sub1);
179
180     int **sub2 = sub_matrix(m, B21, B11);
181     int **M4 = strassen(m, A22, sub2);
182     free(sub2);
183
184     int **add4 = add_matrix(m, A11, A12);
185     int **M5 = strassen(m, add4, B22);
186     free(add4);
187
188     int **sub3 = sub_matrix(m, A21, A11);
189     int **add5 = add_matrix(m, B11, B12);
190     int **M6 = strassen(m, sub3, add5);
191     free(sub3);
192     free(add5);
193
194     int **sub4 = sub_matrix(m, A12, A22);
195     int **add6 = add_matrix(m, B21, B22);
196     int **M7 = strassen(m, sub4, add6);
197     free(sub4);
198     free(add6);
199
200     // calculate the 4 sub-matrixs of the result matrix
201     int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
202     int **C12 = add_matrix(m, M3, M5);
203     int **C21 = add_matrix(m, M2, M4);
204     int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);
205
206     // combine the 4 sub-matrixs into a matrix
207     int i, j;
208     for (i = 0; i < m; i++) {
209         for (j = 0; j < m; j++) {
210             C[i][j] = C11[i][j];
211             C[i][j + m] = C12[i][j];
212             C[i + m][j] = C21[i][j];
213             C[i + m][j + m] = C22[i][j];
214         }
215     }
216
217     free(A11);
218     free(A12);
219     free(A21);
220     free(A22);
221     free(B11);
222     free(B12);

```

```

223     free(B21);
224     free(B22);
225     free(M1);
226     free(M2);
227     free(M3);
228     free(M4);
229     free(M5);
230     free(M6);
231     free(M7);
232     free(C11);
233     free(C12);
234     free(C21);
235     free(C22);
236
237     return C;
238 }
239
240 int main(int argc, char *argv[]) {
241     // check the number of arguments
242     if (argc != 2) {
243         printf("Usage: %s <matrix size>", argv[0]);
244         return -1;
245     }
246
247     // create the matrixs to be multiplied
248     int N = atoi(argv[1]);
249     int n = N;
250
251     // check the matrix size
252     if (check_matrix(N) == -1) {
253         n = next_power_of_2(N);
254     }
255
256     int **A = create_matrix(n); // should return a array of pointers
257     random_matrix(n, N, A);
258
259     int **B = create_matrix(n);
260     random_matrix(n, N, B);
261
262     double starttime = omp_get_wtime();
263
264     int **C = strassen(n, A, B);
265
266     double endtime = omp_get_wtime();
267
268     printf("Time taken: %lf\n", endtime - starttime);
269
270     int **result = naive_calculate(n, A, B);
271
272     // check the accuracy of the Strassen algorithm
273     if (check_accuracy) {
274         printf("The dimension of the matrix is %d, and the result is right!\n", N);
275     }
276
277
278     free(A);
279     free(B);
280     free(C);
281     return 0;
282 }

```

Openmp Version 1

```
1  # include <stdio.h>
2  # include <stdlib.h>
3  # include <stdbool.h>
4  # include <omp.h>
5
6  int check_matrix(int N) {
7      if (N <= 0) {
8          printf("Matrix size must be larger than 0!");
9          return 0;
10     }
11
12     // check if the matrix size is a power of 2
13     while( N != 1) {
14         if (N % 2 != 0) {
15             printf("Matrix size must be a power of 2!\n");
16             return -1;
17         }
18         N = N / 2;
19     }
20     return 1;
21 }
22
23 // find the next power of 2
24 int next_power_of_2(int N) {
25     int i = 1;
26     while (i < N) {
27         i <<= 1;
28     }
29     return i;
30 }
31
32 // allocate memory for a matrix
33 int ** create_matrix(int n) {
34     int *data = (int *)malloc(n * n * sizeof(int));
35     int **array = (int **)malloc(n * sizeof(int *));
36     // check if the memory is allocated
37     if (data == NULL || array == NULL) {
38         printf("Matrix memory allocation failed!");
39         exit(-1);
40     }
41
42     for (int i = 0; i < n; i++) {
43         array[i] = &(data[i * n]);
44     }
45     return array;
46 }
47
48 // randomly generate a matrix
49 void random_matrix(int n, int N, int **A) {
50     int i, j;
51     // printf("%d %d\n", n, N);
52     #pragma omp parallel for collapse(2)
53     for (i = 0; i < N; i++) {
54         for (j = 0; j < N; j++) {
55             A[i][j] = rand() % 10; // generate a random number between 0 and 9
56         }
57     }
58
59     if (n != N) {
60         for (i = N; i < n; i++) {
61             for (j = N; j < n; j++) {
62                 A[i][j] = 0;
63             }
64         }
65     }
66 }
67
68 // function to print the matrix
69 void print_matrix(int N, int **A) {
70     int i, j;
71     for (i = 0; i < N; i++) {
72         for (j = 0; j < N; j++) {
73             printf("%d\t", A[i][j]);
74         }
75     }
76 }
```

```

75     printf("\n");
76 }
77 }
78
79 int ** add_matrix(int n, int **A, int **B) {
80     int i, j;
81     int **result;
82     result = create_matrix(n);
83     // #pragma omp parallel for collapse(2)
84     for (i = 0; i < n; i++) {
85         for (j = 0; j < n; j++) {
86             result[i][j] = A[i][j] + B[i][j];
87         }
88     }
89     return result;
90 }
91
92 int ** sub_matrix(int n, int **A, int **B) {
93     int i, j;
94     int **result;
95     result = create_matrix(n);
96     // #pragma omp parallel for collapse(2)
97     for (i = 0; i < n; i++) {
98         for (j = 0; j < n; j++) {
99             result[i][j] = A[i][j] - B[i][j];
100         }
101     }
102     return result;
103 }
104
105 // divide the matrix into 4 sub-matrixs
106 int ** divide_matrix(int n, int **A, int i, int j) {
107     int **result;
108     result = create_matrix(n / 2);
109     int x, y;
110     // #pragma omp parallel for collapse(2)
111     for (x = 0; x < n / 2; x++) {
112         for (y = 0; y < n / 2; y++) {
113             result[x][y] = A[x + i][y + j];
114         }
115     }
116     return result;
117 }
118
119 // a naive function to calculate the result
120 int **naive_calculate(int n, int **A, int **B) {
121     int **result;
122     result = create_matrix(n);
123     for (int i = 0; i < n; i++) {
124         for (int j = 0; j < n; j++) {
125             result[i][j] = 0;
126             for (int k = 0; k < n; k++) {
127                 result[i][j] += A[i][k] * B[k][j];
128             }
129         }
130     }
131     return result;
132 }
133
134 // check accuracy of the algorithm
135 bool check_accuracy(int n, int **mat1, int **mat2) {
136     for (int i = 0; i < n; i++) {
137         for (int j = 0; j < n; j++) {
138             if (mat1[i][j] != mat2[i][j]) {
139                 return false;
140             }
141         }
142     }
143     return true;
144 }
145
146 int ** strassen(int n, int **A, int **B) {
147     int **C;
148     C = create_matrix(n);
149     if (n == 1) {
150         C[0][0] = A[0][0] * B[0][0];

```

```

151     return C;
152 }
153
154 // decline the matrix demension
155 int m = n / 2;
156 // printf("m = %d\n", m);
157 int **A11 = divide_matrix(n, A, 0, 0);
158 int **A12 = divide_matrix(n, A, 0, m);
159 int **A21 = divide_matrix(n, A, m, 0);
160 int **A22 = divide_matrix(n, A, m, m);
161 int **B11 = divide_matrix(n, B, 0, 0);
162 int **B12 = divide_matrix(n, B, 0, m);
163 int **B21 = divide_matrix(n, B, m, 0);
164 int **B22 = divide_matrix(n, B, m, m);
165
166 int **M1;
167 int **M2;
168 int **M3;
169 int **M4;
170 int **M5;
171 int **M6;
172 int **M7;
173
174 // calculate the 7 sub-matrixs
175 #pragma omp parallel num_threads(8)
176 {
177     #pragma omp single
178     {
179         // int **M1;
180         #pragma omp task
181         {
182             int **add1 = add_matrix(m, A11, A22);
183             int **add2 = add_matrix(m, B11, B22);
184             M1 = strassen(m, add1, add2);
185             free(add1);
186             free(add2);
187         }
188         // int **M2;
189         #pragma omp task
190         {
191             int **add3 = add_matrix(m, A21, A22);
192             M2 = strassen(m, add3, B11);
193             free(add3);
194         }
195         // int **M3;
196         #pragma omp task
197         {
198             int **sub1 = sub_matrix(m, B12, B22);
199             M3 = strassen(m, A11, sub_matrix(m, B12, B22));
200             free(sub1);
201         }
202         // int **M4;
203         #pragma omp task
204         {
205             int **sub2 = sub_matrix(m, B21, B11);
206             M4 = strassen(m, A22, sub2);
207             free(sub2);
208         }
209         // int **M5;
210         #pragma omp task
211         {
212             int **add4 = add_matrix(m, A11, A12);
213             M5 = strassen(m, add4, B22);
214             free(add4);
215         }
216         // int **M6;
217         #pragma omp task
218         {
219             int **sub3 = sub_matrix(m, A21, A11);
220             int **add5 = add_matrix(m, B11, B12);
221             M6 = strassen(m, sub3, add5);
222             free(sub3);
223             free(add5);
224         }
225         // int **M7;
226         #pragma omp task

```



```

227         {
228             int **sub4 = sub_matrix(m, A12, A22);
229             int **add6 = add_matrix(m, B21, B22);
230             M7 = strassen(m, sub4, add6);
231             free(sub4);
232             free(add6);
233         }
234     }
235 }
236
237 // calculate the 4 sub-matrixs of the result matrix
238 #pragma omp taskwait
239 int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
240 int **C12 = add_matrix(m, M3, M5);
241 int **C21 = add_matrix(m, M2, M4);
242 int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);
243
244 // combine the 4 sub-matrixs to get the final result matrix
245 #pragma omp taskwait
246 int i, j;
247 for (i = 0; i < m; i++) {
248     for (j = 0; j < m; j++) {
249         C[i][j] = C11[i][j];
250         C[i][j + m] = C12[i][j];
251         C[i + m][j] = C21[i][j];
252         C[i + m][j + m] = C22[i][j];
253     }
254 }
255
256 }
257
258 // #pragma omp taskwait
259
260 free(A11);
261 free(A12);
262 free(A21);
263 free(A22);
264 free(B11);
265 free(B12);
266 free(B21);
267 free(B22);
268 free(M1);
269 free(M2);
270 free(M3);
271 free(M4);
272 free(M5);
273 free(M6);
274 free(M7);
275
276 return C;
277 }
278
279 int main(int argc, char *argv[]) {
280     // check the number of arguments
281     if (argc != 2) {
282         printf("Usage: %s <matrix size>", argv[0]);
283         return -1;
284     }
285
286     // create the matrixs to be multiplied
287     int N = atoi(argv[1]);
288     int n = N;
289
290     // check the matrix size
291     if (check_matrix(N) == -1) {
292         n = next_power_of_2(N);
293     }
294
295     int **A = create_matrix(n); // should return a array of pointers
296     random_matrix(n, N, A);
297
298     int **B = create_matrix(n);
299     random_matrix(n, N, B);
300
301     int **C;
302

```

```
303     double starttime = omp_get_wtime();
304     omp_set_num_threads(8);
305
306     C = strassen(n, A, B);
307
308     double endtime = omp_get_wtime();
309     printf("Time taken: %lf\n", endtime - starttime);
310     // print_matrix(N, C);
311
312     // int **result = naive_calculate(n, A, B);
313
314     // // check the accuracy of the Strassen algorithm
315     // if (check_accuracy) {
316     //     printf("The dimension of the matrix is %d, and the result is right!\n", N);
317     // }
318
319     free(A);
320     free(B);
321     free(C);
322     return 0;
323 }
```

OpenMP Version 2

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <omp.h>
5
6 int check_matrix(int N) {
7     if (N <= 0) {
8         printf("Matrix size must be larger than 0!");
9         return 0;
10    }
11
12    // check if the matrix size is a power of 2
13    while( N != 1) {
14        if (N % 2 != 0) {
15            printf("Matrix size must be a power of 2!\n");
16            return -1;
17        }
18        N = N / 2;
19    }
20    return 1;
21 }
22
23 // find the next power of 2
24 int next_power_of_2(int N) {
25     int i = 1;
26     while (i < N) {
27         i <<= 1;
28     }
29     return i;
30 }
31
32 // allocate memory for a matrix
33 int ** create_matrix(int n) {
34     int *data = (int *)malloc(n * n * sizeof(int));
35     int **array = (int **)malloc(n * sizeof(int *));
36     // check if the memory is allocated
37     if (data == NULL || array == NULL) {
38         printf("Matrix memory allocation failed!");
39         exit(-1);
40     }
41
42     for (int i = 0; i < n; i++) {
43         array[i] = &(data[i * n]);
44     }
45     return array;
46 }
47
48 // randomly generate a matrix
49 void random_matrix(int n, int N, int **A) {
50     int i, j;
51     // printf("%d %d\n", n, N);
52     #pragma omp parallel for collapse(2)
53     for (i = 0; i < N; i++) {
54         for (j = 0; j < N; j++) {
55             A[i][j] = rand() % 10; // generate a random number between 0 and 9
56         }
57     }
58
59     if (n != N) {
60         for (i = N; i < n; i++) {
61             for (j = N; j < n; j++) {
62                 A[i][j] = 0;
63             }
64         }
65     }
66 }
67
68 // function to print the matrix
69 void print_matrix(int N, int **A) {
70     int i, j;
71     for (i = 0; i < N; i++) {
72         for (j = 0; j < N; j++) {
73             printf("%d\t", A[i][j]);
74         }
75     }
76 }
```

```

75     printf("\n");
76 }
77 }
78
79 int ** add_matrix(int n, int **A, int **B) {
80     int i, j;
81     int **result;
82     result = create_matrix(n);
83     // #pragma omp parallel for collapse(2)
84     for (i = 0; i < n; i++) {
85         for (j = 0; j < n; j++) {
86             result[i][j] = A[i][j] + B[i][j];
87         }
88     }
89     return result;
90 }
91
92 int ** sub_matrix(int n, int **A, int **B) {
93     int i, j;
94     int **result;
95     result = create_matrix(n);
96     // #pragma omp parallel for collapse(2)
97     for (i = 0; i < n; i++) {
98         for (j = 0; j < n; j++) {
99             result[i][j] = A[i][j] - B[i][j];
100         }
101     }
102     return result;
103 }
104
105 // divide the matrix into 4 sub-matrixs
106 int ** divide_matrix(int n, int **A, int i, int j) {
107     int **result;
108     result = create_matrix(n / 2);
109     int x, y;
110     // #pragma omp parallel for collapse(2)
111     for (x = 0; x < n / 2; x++) {
112         for (y = 0; y < n / 2; y++) {
113             result[x][y] = A[x + i][y + j];
114         }
115     }
116     return result;
117 }
118
119 // a naive function to calculate the result
120 int **naive_calculate(int n, int **A, int **B) {
121     int **result;
122     result = create_matrix(n);
123     for (int i = 0; i < n; i++) {
124         for (int j = 0; j < n; j++) {
125             result[i][j] = 0;
126             for (int k = 0; k < n; k++) {
127                 result[i][j] += A[i][k] * B[k][j];
128             }
129         }
130     }
131     return result;
132 }
133
134 // check accuracy of the algorithm
135 bool check_accuracy(int n, int **mat1, int **mat2) {
136     for (int i = 0; i < n; i++) {
137         for (int j = 0; j < n; j++) {
138             if (mat1[i][j] != mat2[i][j]) {
139                 return false;
140             }
141         }
142     }
143     return true;
144 }
145
146 int ** strassen(int n, int **A, int **B) {
147     int **C;
148     C = create_matrix(n);
149     if (n == 1) {
150         C[0][0] = A[0][0] * B[0][0];

```

```

151     return C;
152 }
153
154 // decline the matrix demension
155 int m = n / 2;
156 // printf("m = %d\n", m);
157 int **A11 = divide_matrix(n, A, 0, 0);
158 int **A12 = divide_matrix(n, A, 0, m);
159 int **A21 = divide_matrix(n, A, m, 0);
160 int **A22 = divide_matrix(n, A, m, m);
161 int **B11 = divide_matrix(n, B, 0, 0);
162 int **B12 = divide_matrix(n, B, 0, m);
163 int **B21 = divide_matrix(n, B, m, 0);
164 int **B22 = divide_matrix(n, B, m, m);
165
166 int **M1;
167 int **M2;
168 int **M3;
169 int **M4;
170 int **M5;
171 int **M6;
172 int **M7;
173
174 // calculate the 7 sub-matrixs
175 #pragma omp parallel num_threads(8)
176 {
177     #pragma omp sections
178     {
179         // int **M1;
180         #pragma omp section
181         {
182             int **add1 = add_matrix(m, A11, A22);
183             int **add2 = add_matrix(m, B11, B22);
184             M1 = strassen(m, add1, add2);
185             free(add1);
186             free(add2);
187         }
188         // int **M2;
189         #pragma omp section
190         {
191             int **add3 = add_matrix(m, A21, A22);
192             M2 = strassen(m, add3, B11);
193             free(add3);
194         }
195         // int **M3;
196         #pragma omp section
197         {
198             int **sub1 = sub_matrix(m, B12, B22);
199             M3 = strassen(m, A11, sub_matrix(m, B12, B22));
200             free(sub1);
201         }
202         // int **M4;
203         #pragma omp section
204         {
205             int **sub2 = sub_matrix(m, B21, B11);
206             M4 = strassen(m, A22, sub2);
207             free(sub2);
208         }
209         // int **M5;
210         #pragma omp section
211         {
212             int **add4 = add_matrix(m, A11, A12);
213             M5 = strassen(m, add4, B22);
214             free(add4);
215         }
216         // int **M6;
217         #pragma omp section
218         {
219             int **sub3 = sub_matrix(m, A21, A11);
220             int **add5 = add_matrix(m, B11, B12);
221             M6 = strassen(m, sub3, add5);
222             free(sub3);
223             free(add5);
224         }
225         // int **M7;
226         #pragma omp section

```

```

227         {
228             int **sub4 = sub_matrix(m, A12, A22);
229             int **add6 = add_matrix(m, B21, B22);
230             M7 = strassen(m, sub4, add6);
231             free(sub4);
232             free(add6);
233         }
234     }
235 }
236
237
238
239     int **C11 = add_matrix(m, sub_matrix(m, add_matrix(m, M1, M4), M5), M7);
240     int **C12 = add_matrix(m, M3, M5);
241     int **C21 = add_matrix(m, M2, M4);
242     int **C22 = add_matrix(m, add_matrix(m, sub_matrix(m, M1, M2), M3), M6);
243
244     // combine the 4 sub-matrixs to get the final result matrix
245     // #pragma omp taskwait
246     int i, j;
247     #pragma omp parallel for collapse(2)
248     for (i = 0; i < m; i++) {
249         for (j = 0; j < m; j++) {
250             C[i][j] = C11[i][j];
251             C[i][j + m] = C12[i][j];
252             C[i + m][j] = C21[i][j];
253             C[i + m][j + m] = C22[i][j];
254         }
255     }
256
257     // }
258
259     // #pragma omp taskwait
260
261     free(A11);
262     free(A12);
263     free(A21);
264     free(A22);
265     free(B11);
266     free(B12);
267     free(B21);
268     free(B22);
269     free(M1);
270     free(M2);
271     free(M3);
272     free(M4);
273     free(M5);
274     free(M6);
275     free(M7);
276
277     return C;
278 }
279
280 int main(int argc, char *argv[]) {
281     // check the number of arguments
282     if (argc != 2) {
283         printf("Usage: %s <matrix size>", argv[0]);
284         return -1;
285     }
286
287     // create the matrixs to be multiplied
288     int N = atoi(argv[1]);
289     int n = N;
290
291     // check the matrix size
292     if (check_matrix(N) == -1) {
293         n = next_power_of_2(N);
294     }
295
296     int **A = create_matrix(n); // should return a array of pointers
297     random_matrix(n, N, A);
298     // printf("Matrix A:\n");
299     // print_matrix(n, A);
300     // printf("\n");
301
302     int **B = create_matrix(n);

```

```

303     random_matrix(n, N, B);
304     // print_matrix(n, B);
305     // printf("\n");
306
307     int **C;
308
309     double starttime = omp_get_wtime();
310     omp_set_num_threads(8);
311
312     C = strassen(n, A, B);
313
314     double endtime = omp_get_wtime();
315     printf("Time taken: %lf\n", endtime - starttime);
316     // print_matrix(N, C);
317
318     int **result = naive_calculate(n, A, B);
319
320     // check the accuracy of the Strassen algorithm
321     if (check_accuracy) {
322         printf("The dimension of the matrix is %d, and the result is right!\n", N);
323     }
324
325     free(A);
326     free(B);
327     free(C);
328     return 0;
329 }

```
