



# UPPSALA UNIVERSITET

High Performance Programming

Individual Project  
Matrix-matrix multiplication with Strassen algorithm

Jinglin Gao

August 3, 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  $2^n$ . In practice usually just cut the matrix into uneven blocks and process Strassen algorithm. But here 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 for matrix sizes above a certain crossover point, and change back to the naive multiplication method. The crossover point varies a lot since it depends on specific implementation and hardware.

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 10.

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 submatrices recursively. In order to make the calculation much more simple we defined the *sum* as helper functions. Since this 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 Strassen algorithm, 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 submatrix can be executed independently, making it a perfect candidate for parallelization.

Since its a recursive algorithm we have to consider how to parallel the program. It is inefficient to create threads recursively, but we can choose create tasks in tasks. We use *pragma omp task [clause]* to acheive this. By adding this directive each encountering thread/task creates a new task. In our program I use *shared* as a data scoping clause, so the data is visible to every thread/task. Also we need to make sure that all tasks are completed when we calculate and form the final result matrix, so we use *pragma omp taskwait* to wait until child tasks complete. In our main function we first call *pragma omp parallel* and *pragma omp single* so that one tasks starts the execution of the algorithm. And then we use *pragma omp task* to indicate that this piece of codes should be worked in parallel. This directive let other threads to help out by executing the tasks generated by the first thread.

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.

In addition to parallelizing the *strassen* algorithm, I attempted to optimize the *sum* and *naive\_mul* functions by parallelizing it with the *# pragma omp parallel for collapse(2)* directive. It turns out that if we also parallelize the *sum* function it will slow down the program. So in the end I abandoned this parallel method.

## 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 order:  
In the functions "*naive\_mul*", I implement a more efficient loop order to calculate the result. Since in cache the data are stored row by row, if we frequently access the data already in cache will significantly improve the performance. Here *jik* is the best loop order.

## 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.

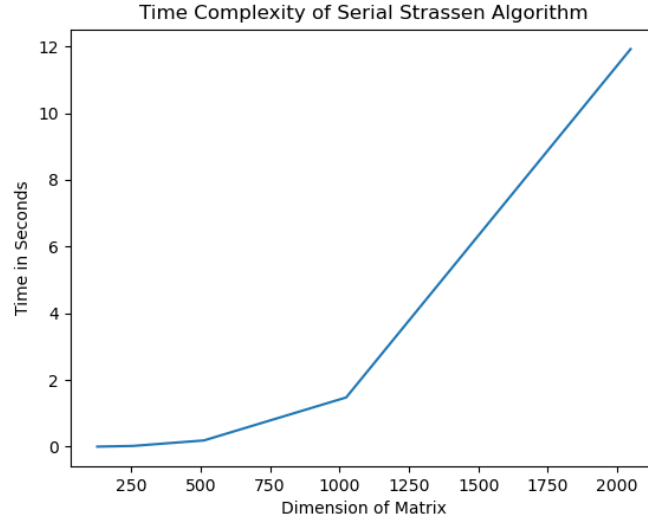


Figure 1: Measured execution time for Strassen algorithm

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.

```

● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 2 1
Parallel Strassen Runtime: 0.002021
The dimension of the matrix is 2, and the result is right!
● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 35 8
Matrix size must be a power of 2!
Parallel Strassen Runtime: 0.000958
The dimension of the matrix is 64, and the result is right!
● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 64 8
Parallel Strassen Runtime: 0.000931
The dimension of the matrix is 64, and the result is right!
● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 100 8
Matrix size must be a power of 2!
Parallel Strassen Runtime: 0.001828
The dimension of the matrix is 128, and the result is right!
● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 128 8
Parallel Strassen Runtime: 0.001824
The dimension of the matrix is 128, and the result is right!
● gaj1941@vitsippa:~/HPP_project$ ./v1_omp 500 8
Matrix size must be a power of 2!
Parallel Strassen Runtime: 0.048125
The dimension of the matrix is 512, 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.

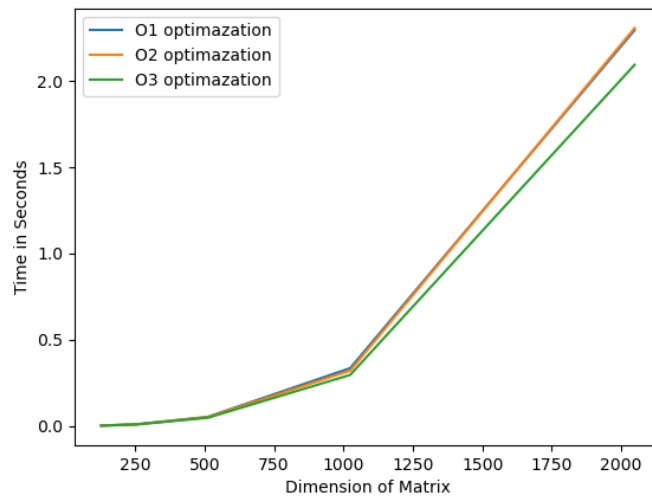


Figure 3: Different compiler optimization techniques performance

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 -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.

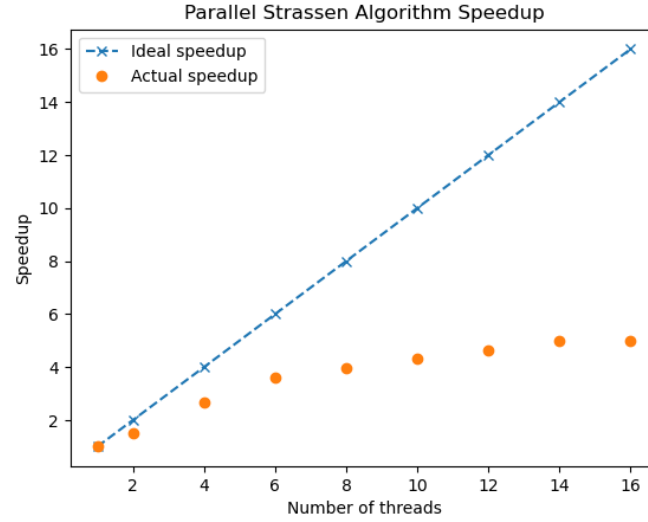


Figure 4: Speed up using different number of threads (n=512)

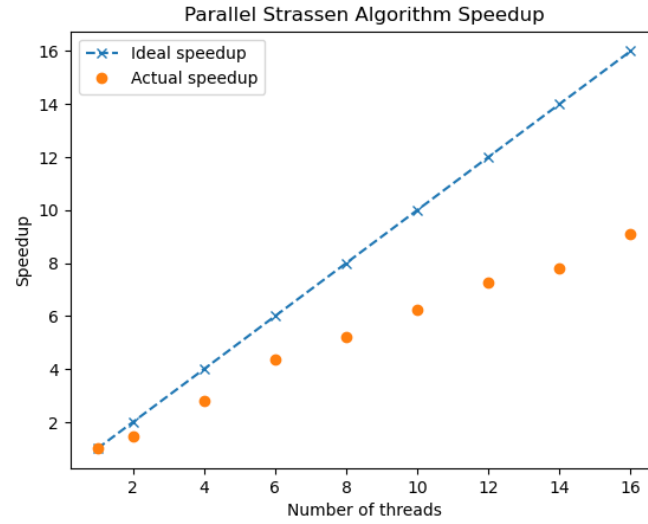


Figure 5: Speed up using different number of threads (n=2048)

It is clear that with the increase in the matrix dimension, we can reach a more ideal speed-up. The maximum speedup I reach is around 9 times when I compute a matrix with 2048 dimensions.

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. Also since all threads use the same memory all mallocs need to synchronize between threads. 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 Strassen algorithm performs better with larger matrices, there is a study that says that usually Strassen works better with  $n > 1000$  where  $n$  is the dimension of the matrix.[1]

## 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 the Strassen algorithm performs better when the size of the matrix is large. The best speedup I reached is around 9 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. One way to figure out this problem is instead of using a lot of mallocs in the program, we can just malloc a huge working area and point into that area each time.

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 9 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] Tuan Nguyen, Alex Adamson, and Andreas Santucci. Matrix multiplication: Strassen's algorithm.
- [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  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <omp.h>
5
6  void allocate_mem(int*** arr, int n);
7  void free_mem(int** arr, int n);
8  int rand_int(int N);
9  int ** naive_mul(int n, int **a, int **b);
10 bool check(int n, int **mat1, int **mat2);
11 void print_matrix(int N, int **a);
12 int ** sum(int n, int **a, int **b, bool sum);
13 int ** divide_matrix(int n, int **a, int i, int j);
14 int ** strassen(int n, int **a, int **b);
15 int ** combine_matrix(int n, int **a, int **b, int **c, int **d);
16
17
18 int main(int argc, char *argv[])
19 {
20     if (argc != 2) {
21         printf("Usage: %s <N>\n", argv[0]);
22         return -1;
23     }
24
25     int i, j, n;
26     int **a;
27     int **b;
28     int **c;
29     int **d;
30     int Nmax = 10; // random numbers in [0, N]
31
32     n = atoi(argv[1]); // get matrix size from command line
33
34     printf("I am here\n");
35
36     // create matrix A
37     allocate_mem(&a, n);
38     for ( i = 0 ; i < n ; i++ )
39         for ( j = 0 ; j < n ; j++ )
40             a[i][j] = rand_int(Nmax);
41
42     // print matrix A
43     // printf("Matrix A is:\n");
44     // print_matrix(n, a);
45
46     // create matrix B
47     allocate_mem(&b, n);
48     for ( i = 0 ; i < n ; i++ )
49         for ( j = 0 ; j < n ; j++ )
50             b[i][j] = rand_int(Nmax);
51
52     double start_strassen = omp_get_wtime();
53
54     c = strassen(n, a, b);
55
56     double end_strassen = omp_get_wtime();
57     d = naive_mul(n, a, b);
58
59     printf("Naive Strassen Runtime: %lf\n", end_strassen - start_strassen);
60
61     // check the accuracy of the Strassen algorithm
62     bool res = check(n, c, d);
63     if (res == true) {
64         printf("The dimension of the matrix is %d, and the result is right!\n", n);
65     }
66
67     free_mem(a, n);
68     free_mem(b, n);
69     free_mem(c, n);
70     free_mem(d, n);
```

```

71     return 0;
72 }
73
74
75
76 // allocate memory for a 2D array
77 void allocate_mem(int*** arr, int n)
78 {
79     int i;
80     *arr = (int**)malloc(n*sizeof(int*));
81     for(i=0; i<n; i++)
82         (*arr)[i] = (int*)malloc(n*sizeof(int));
83 }
84
85 // free memory for a 2D array
86 void free_mem(int** arr, int n)
87 {
88     int i;
89     for(i=0; i<n; i++)
90         free(arr[i]);
91     free(arr);
92 }
93
94 // generate a random interger in [0, N - 1]
95 int rand_int(int N)
96 {
97     int val = -1;
98     while( val < 0 || val >= N )
99     {
100         val = (int)(N * (double)rand()/RAND_MAX);
101     }
102     return val;
103 }
104
105 // naive matrix multiplication function, jik has the best loop order
106 int ** naive_mul(int n, int **a, int **b)
107 {
108     int **c;
109     allocate_mem(&c, n);
110     int i, j, k;;
111     for (j = 0; j < n; j++) {
112         for (i = 0; i < n; i++) {
113             int sum = 0;
114             for (k = 0; k < n; k++) {
115                 sum += a[i][k] * b[k][j];
116             }
117             c[i][j] = sum;
118         }
119     }
120     return c;
121 }
122
123 // check the accuracy of the Strassen algorithm
124 bool check(int n, int **mat1, int **mat2)
125 {
126     for (int i = 0; i < n; i++) {
127         for (int j = 0; j < n; j++) {
128             if (mat1[i][j] != mat2[i][j]) {
129                 return false;
130             }
131         }
132     }
133     return true;
134 }
135
136 // strassen function
137 int ** strassen(int n, int **a, int **b)
138 {
139
140     if (n <= 32) {
141         return naive_mul(n, a, b);
142     }
143
144     else {
145         int new_size = n / 2;
146         // create the submatrix

```

```

147 int **a11, **a12, **a21, **a22;
148 int **b11, **b12, **b21, **b22;
149 int **c11, **c12, **c21, **c22;
150 int **m1, **m2, **m3, **m4, **m5, **m6, **m7;
151
152
153 // divide the matrix into 4 sub-matrixs
154 a11 = divide_matrix(n, a, 0, 0);
155 a12 = divide_matrix(n, a, 0, new_size);
156 a21 = divide_matrix(n, a, new_size, 0);
157 a22 = divide_matrix(n, a, new_size, new_size);
158
159 b11 = divide_matrix(n, b, 0, 0);
160 b12 = divide_matrix(n, b, 0, new_size);
161 b21 = divide_matrix(n, b, new_size, 0);
162 b22 = divide_matrix(n, b, new_size, new_size);
163
164
165 int **minus_a = sum(new_size, a12, a22, false);
166 int **add_b = sum(new_size, b21, b22, true);
167 m7 = strassen(new_size, minus_a, add_b);
168 free_mem(minus_a, new_size);
169 free_mem(add_b, new_size);
170
171 int **minus_a1 = sum(new_size, a21, a11, false);
172 int **add_b1 = sum(new_size, b11, b12, true);
173 m6 = strassen(new_size, minus_a1, add_b1);
174 free_mem(minus_a1, new_size);
175 free_mem(add_b1, new_size);
176
177 int **add_a = sum(new_size, a11, a12, true);
178 m5 = strassen(new_size, add_a, b22);
179 free_mem(add_a, new_size);
180
181 int **minus_b = sum(new_size, b21, b11, false);
182 m4 = strassen(new_size, a22, minus_b);
183 free_mem(minus_b, new_size);
184
185 int **minus_b1 = sum(new_size, b12, b22, false);
186 m3 = strassen(new_size, a11, minus_b1);
187 free_mem(minus_b1, new_size);
188
189 int **add_a1 = sum(new_size, a21, a22, true);
190 m2 = strassen(new_size, add_a1, b11);
191 free_mem(add_a1, new_size);
192
193 int **add_a2 = sum(new_size, a11, a22, true);
194 int **add_b2 = sum(new_size, b11, b22, true);
195 m1 = strassen(new_size, add_a2, add_b2);
196 free_mem(add_a2, new_size);
197 free_mem(add_b2, new_size);
198
199 free_mem(a11, new_size);
200 free_mem(a12, new_size);
201 free_mem(a21, new_size);
202 free_mem(a22, new_size);
203 free_mem(b11, new_size);
204 free_mem(b12, new_size);
205 free_mem(b21, new_size);
206 free_mem(b22, new_size);
207
208 int **add_m17 = sum(new_size, m1, m7, true);
209 int **sub_m45 = sum(new_size, m4, m5, false);
210 c11 = sum(new_size, add_m17, sub_m45, true);
211 free_mem(add_m17, new_size);
212 free_mem(sub_m45, new_size);
213
214 c12 = sum(new_size, m3, m5, true);
215
216 c21 = sum(new_size, m2, m4, true);
217
218 int **sub_m12 = sum(new_size, m1, m2, false);
219 int **sum_m36 = sum(new_size, m3, m6, true);
220 c22 = sum(new_size, sub_m12, sum_m36, true);
221 free_mem(sub_m12, new_size);
222 free_mem(sum_m36, new_size);

```

```

223     free_mem(m1, new_size);
224     free_mem(m2, new_size);
225     free_mem(m3, new_size);
226     free_mem(m4, new_size);
227     free_mem(m5, new_size);
228     free_mem(m6, new_size);
229     free_mem(m7, new_size);
230
231
232     // combine the submatrix
233     int **c;
234     c = combine_matrix(new_size, c11, c12, c21, c22);
235
236     free_mem(c11, new_size);
237     free_mem(c12, new_size);
238     free_mem(c21, new_size);
239     free_mem(c22, new_size);
240
241     return c;
242 }
243 }
244
245 void print_matrix(int N, int **a)
246 {
247     int i, j;
248     for (i = 0; i < N; i++) {
249         for (j = 0; j < N; j++) {
250             printf("%d\t", a[i][j]);
251         }
252         printf("\n");
253     }
254 }
255
256
257 int ** sum(int n, int **a, int **b, bool sum)
258 {
259     int i, j;
260     int **result;
261     allocate_mem(&result, n);
262     for (i=0; i<n; i++) {
263         for (j=0; j<n; j++) {
264             if (sum) {
265                 result[i][j] = a[i][j] + b[i][j];
266             }
267             else {
268                 result[i][j] = a[i][j] - b[i][j];
269             }
270         }
271     }
272     return result;
273 }
274
275 // divide the matrix into 4 sub-matrixs
276 int ** divide_matrix(int n, int **a, int i, int j)
277 {
278     int **result;
279     allocate_mem(&result, n / 2);
280     int x, y;
281     for (x = 0; x < n / 2; x++) {
282         for (y = 0; y < n / 2; y++) {
283             result[x][y] = a[x + i][y + j];
284         }
285     }
286     return result;
287 }
288
289 int ** combine_matrix(int n, int **a, int **b, int **c, int **d)
290 {
291     int size = n * 2;
292     int **result;
293     allocate_mem(&result, size);
294     for (int i = 0; i < size; i++) {
295         for (int j = 0; j < size; j++) {
296             if (i < n && j < n) {
297                 result[i][j] = a[i][j];
298             }

```

```
299     else if (i < n) {
300         result[i][j] = b[i][j-n];
301     }
302     else if (j < n) {
303         result[i][j] = c[i-n][j];
304     }
305     else {
306         result[i][j] = d[i-n][j-n];
307     }
308 }
309 }
310 return result;
311 }
```

---

## Parallel with Openmp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <omp.h>
5
6  void allocate_mem(int*** arr, int n);
7  void free_mem(int** arr, int n);
8  int rand_int(int N);
9  int ** naive_mul(int n, int **a, int **b);
10 bool check(int n, int **mat1, int **mat2);
11 void print_matrix(int N, int **a);
12 int ** sum(int n, int **a, int **b, bool sum);
13 int ** divide_matrix(int n, int **a, int i, int j);
14 int ** strassen(int n, int **a, int **b);
15 int ** combine_matrix(int n, int **a, int **b, int **c, int **d);
16 int check_matrix(int N);
17 int next_power_of_2(int N);
18
19
20 int main(int argc, char *argv[])
21 {
22     if (argc != 3) {
23         printf("Usage: %s <N> <number of threads>\n", argv[0]);
24         return -1;
25     }
26
27     int i, j, n, N;
28     int **a;
29     int **b;
30     int **c;
31     int **d;
32     int Nmax = 10; // random numbers in [0, N]
33
34     n = atoi(argv[1]); // get matrix size from command line
35     N = atoi(argv[2]);
36
37     // check the matrix size
38     if (check_matrix(n) == -1) {
39         n = next_power_of_2(n);
40     }
41
42     // create matrix A
43     allocate_mem(&a, n);
44     for ( i = 0 ; i < n ; i++ )
45         for ( j = 0 ; j < n ; j++ )
46             a[i][j] = rand_int(Nmax);
47
48     // print matrix A
49     // printf("Matrix A is:\n");
50     // print_matrix(n, a);
51
52     // create matrix B
53     allocate_mem(&b, n);
54     for ( i = 0 ; i < n ; i++ )
55         for ( j = 0 ; j < n ; j++ )
56             b[i][j] = rand_int(Nmax);
57
58     double start_strassen = omp_get_wtime();
59
60     #pragma omp parallel num_threads(N)
61     {
62         #pragma omp single
63         {
64             c = strassen(n, a, b);
65         }
66     }
67
68     double end_strassen = omp_get_wtime();
69     d = naive_mul(n, a, b);
70
71     printf("Parallel Strassen Runtime: %lf\n", end_strassen - start_strassen);
72
73     // check the accuracy of the Strassen algorithm
74     bool res = check(n, c, d);
```

```

75     if (res == true) {
76         printf("The dimension of the matrix is %d, and the result is right!\n", n);
77     }
78
79     free_mem(a, n);
80     free_mem(b, n);
81     free_mem(c, n);
82     free_mem(d, n);
83     return 0;
84 }
85
86
87
88 // allocate memory for a 2D array
89 void allocate_mem(int*** arr, int n)
90 {
91     int i;
92     *arr = (int**)malloc(n*sizeof(int*));
93     for(i=0; i<n; i++)
94         (*arr)[i] = (int*)malloc(n*sizeof(int));
95 }
96
97 // free memory for a 2D array
98 void free_mem(int** arr, int n)
99 {
100     int i;
101     for(i=0; i<n; i++)
102         free(arr[i]);
103     free(arr);
104 }
105
106 // generate a random interger in [0, N - 1]
107 int rand_int(int N)
108 {
109     int val = -1;
110     while( val < 0 || val >= N )
111     {
112         val = (int)(N * (double)rand()/RAND_MAX);
113     }
114     return val;
115 }
116
117 // naive matrix multiplication function, jik has the best loop order
118 int ** naive_mul(int n, int **a, int **b)
119 {
120     int **c;
121     allocate_mem(&c, n);
122     int i, j, k;
123
124     #pragma omp parallel for collapse(2)
125     for (j = 0; j < n; j++) {
126         for (i = 0; i < n; i++) {
127             int sum = 0;
128             for (k = 0; k < n; k++) {
129                 sum += a[i][k] * b[k][j];
130             }
131             c[i][j] = sum;
132         }
133     }
134
135     return c;
136 }
137
138 // check the accuracy of the Strassen algorithm
139 bool check(int n, int **mat1, int **mat2)
140 {
141     for (int i = 0; i < n; i++) {
142         for (int j = 0; j < n; j++) {
143             if (mat1[i][j] != mat2[i][j]) {
144                 return false;
145             }
146         }
147     }
148     return true;
149 }
150

```

```

151 // strassen function
152 int ** strassen(int n, int **a, int **b)
153 {
154
155     if (n <= 32) {
156         return naive_mul(n, a, b);
157     }
158
159     else {
160         int new_size = n / 2;
161         // create the submatrix
162         int **a11, **a12, **a21, **a22;
163         int **b11, **b12, **b21, **b22;
164         int **c11, **c12, **c21, **c22;
165         int **m1, **m2, **m3, **m4, **m5, **m6, **m7;
166
167
168         // divide the matrix into 4 sub-matrixs
169         a11 = divide_matrix(n, a, 0, 0);
170         a12 = divide_matrix(n, a, 0, new_size);
171         a21 = divide_matrix(n, a, new_size, 0);
172         a22 = divide_matrix(n, a, new_size, new_size);
173
174         b11 = divide_matrix(n, b, 0, 0);
175         b12 = divide_matrix(n, b, 0, new_size);
176         b21 = divide_matrix(n, b, new_size, 0);
177         b22 = divide_matrix(n, b, new_size, new_size);
178
179         #pragma omp task shared(m7)
180         {
181             int **minus_a = sum(new_size, a12, a22, false);
182             int **add_b = sum(new_size, b21, b22, true);
183             m7 = strassen(new_size, minus_a, add_b);
184             free_mem(minus_a, new_size);
185             free_mem(add_b, new_size);
186         }
187
188         #pragma omp task shared(m6)
189         {
190             int **minus_a1 = sum(new_size, a21, a11, false);
191             int **add_b1 = sum(new_size, b11, b12, true);
192             m6 = strassen(new_size, minus_a1, add_b1);
193             free_mem(minus_a1, new_size);
194             free_mem(add_b1, new_size);
195         }
196
197         #pragma omp task shared(m5)
198         {
199             int **add_a = sum(new_size, a11, a12, true);
200             m5 = strassen(new_size, add_a, b22);
201             free_mem(add_a, new_size);
202         }
203
204         #pragma omp task shared(m4)
205         {
206             int **minus_b = sum(new_size, b21, b11, false);
207             m4 = strassen(new_size, a22, minus_b);
208             free_mem(minus_b, new_size);
209         }
210
211         #pragma omp task shared(m3)
212         {
213             int **minus_b1 = sum(new_size, b12, b22, false);
214             m3 = strassen(new_size, a11, minus_b1);
215             free_mem(minus_b1, new_size);
216         }
217
218         #pragma omp task shared(m2)
219         {
220             int **add_a1 = sum(new_size, a21, a22, true);
221             m2 = strassen(new_size, add_a1, b11);
222             free_mem(add_a1, new_size);
223         }
224
225         #pragma omp task shared(m1)
226         {

```



```

227     int **add_a2 = sum(new_size, a11, a22, true);
228     int **add_b2 = sum(new_size, b11, b22, true);
229     m1 = strassen(new_size, add_a2, add_b2);
230     free_mem(add_a2, new_size);
231     free_mem(add_b2, new_size);
232 }
233
234 #pragma omp taskwait
235
236 free_mem(a11, new_size);
237 free_mem(a12, new_size);
238 free_mem(a21, new_size);
239 free_mem(a22, new_size);
240 free_mem(b11, new_size);
241 free_mem(b12, new_size);
242 free_mem(b21, new_size);
243 free_mem(b22, new_size);
244
245 #pragma omp task shared(c11)
246 {
247     int **add_m17 = sum(new_size, m1, m7, true);
248     int **sub_m45 = sum(new_size, m4, m5, false);
249     c11 = sum(new_size, add_m17, sub_m45, true);
250     free_mem(add_m17, new_size);
251     free_mem(sub_m45, new_size);
252 }
253
254 #pragma omp task shared(c12)
255 {
256     c12 = sum(new_size, m3, m5, true);
257 }
258
259 #pragma omp task shared(c21)
260 {
261     c21 = sum(new_size, m2, m4, true);
262 }
263
264 #pragma omp task shared(c22)
265 {
266     int **sub_m12 = sum(new_size, m1, m2, false);
267     int **sum_m36 = sum(new_size, m3, m6, true);
268     c22 = sum(new_size, sub_m12, sum_m36, true);
269     free_mem(sub_m12, new_size);
270     free_mem(sum_m36, new_size);
271 }
272
273 #pragma omp taskwait
274
275 free_mem(m1, new_size);
276 free_mem(m2, new_size);
277 free_mem(m3, new_size);
278 free_mem(m4, new_size);
279 free_mem(m5, new_size);
280 free_mem(m6, new_size);
281 free_mem(m7, new_size);
282
283 // combine the submatrix
284 int **c;
285 c = combine_matrix(new_size, c11, c12, c21, c22);
286
287 free_mem(c11, new_size);
288 free_mem(c12, new_size);
289 free_mem(c21, new_size);
290 free_mem(c22, new_size);
291
292 return c;
293 }
294 }
295
296 void print_matrix(int N, int **a)
297 {
298     int i, j;
299     for (i = 0; i < N; i++) {
300         for (j = 0; j < N; j++) {
301             printf("%d\t", a[i][j]);
302         }

```

```

303     printf("\n");
304 }
305 }
306
307
308 int ** sum(int n, int **a, int **b, bool sum)
309 {
310     int i, j;
311     int **result;
312     allocate_mem(&result, n);
313     // #pragma omp parallel for collapse(2)
314     for (i=0; i<n; i++) {
315         for (j=0; j<n; j++) {
316             if (sum) {
317                 result[i][j] = a[i][j] + b[i][j];
318             }
319             else {
320                 result[i][j] = a[i][j] - b[i][j];
321             }
322         }
323     }
324     return result;
325 }
326
327 // divide the matrix into 4 sub-matrixs
328 int ** divide_matrix(int n, int **a, int i, int j)
329 {
330     int **result;
331     allocate_mem(&result, n / 2);
332     int x, y;
333     for (x = 0; x < n / 2; x++) {
334         for (y = 0; y < n / 2; y++) {
335             result[x][y] = a[x + i][y + j];
336         }
337     }
338     return result;
339 }
340
341 int ** combine_matrix(int n, int **a, int **b, int **c, int **d)
342 {
343     int size = n * 2;
344     int **result;
345     allocate_mem(&result, size);
346     for (int i = 0; i < size; i++) {
347         for (int j = 0; j < size; j++) {
348             if (i < n && j < n) {
349                 result[i][j] = a[i][j];
350             }
351             else if (i < n) {
352                 result[i][j] = b[i][j-n];
353             }
354             else if (j < n) {
355                 result[i][j] = c[i-n][j];
356             }
357             else {
358                 result[i][j] = d[i-n][j-n];
359             }
360         }
361     }
362     return result;
363 }
364
365 int check_matrix(int N)
366 {
367     if (N <= 0) {
368         printf("Matrix size must be larger than 0!");
369         return 0;
370     }
371     // check if the matrix size is a power of 2
372     while( N != 1) {
373         if (N % 2 != 0) {
374             printf("Matrix size must be a power of 2!\n");
375             return -1;
376         }
377         N = N / 2;
378     }

```

```
379     return 1;
380 }
381
382 // find the next power of 2
383 int next_power_of_2(int N)
384 {
385     int i = 1;
386     while (i < N) {
387         i <<= 1;
388     }
389     return i;
390 }
```

---