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.

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

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 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 recursivly, 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 efficiant 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.

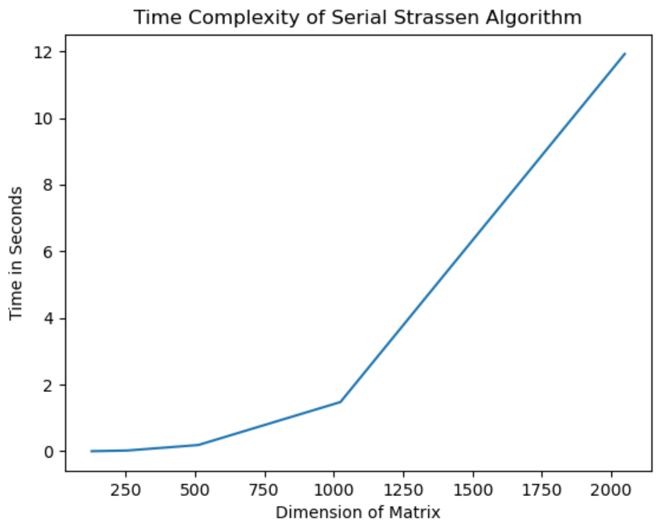
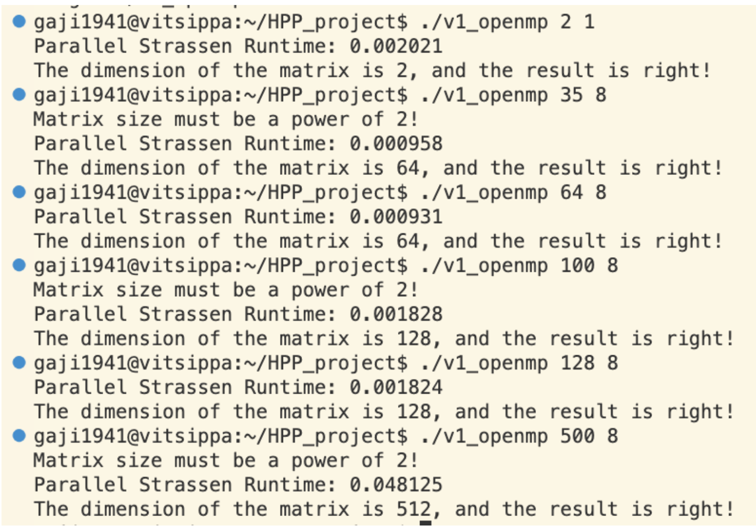


Figure 1: Measured execution time for Strassen algorithm

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.



```
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 2 1
Parallel Strassen Runtime: 0.002021
The dimension of the matrix is 2, and the result is right!
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 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!
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 64 8
Parallel Strassen Runtime: 0.000931
The dimension of the matrix is 64, and the result is right!
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 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!
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 128 8
Parallel Strassen Runtime: 0.001824
The dimension of the matrix is 128, and the result is right!
gaji1941@vitsippa:~/HPP_project$ ./v1_openmp 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.

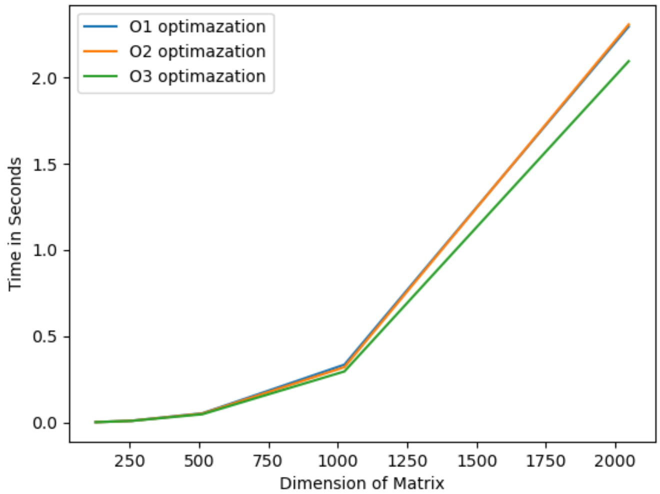


Figure 3: Different compiler optimization techniques performance

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.

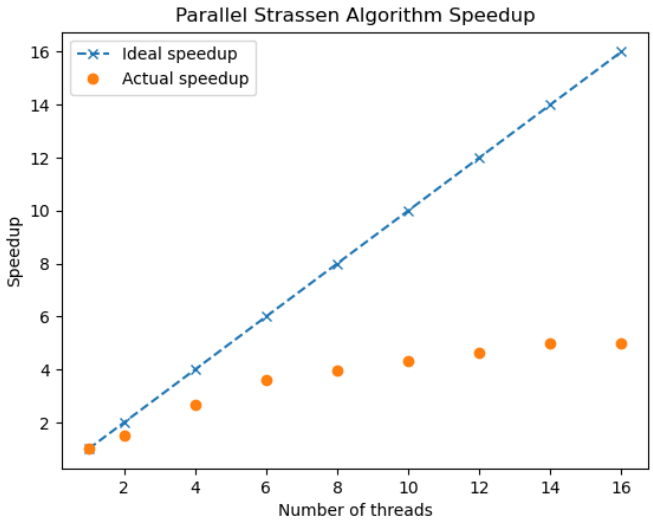


Figure 4: Speed up using different number of threads (n=512)

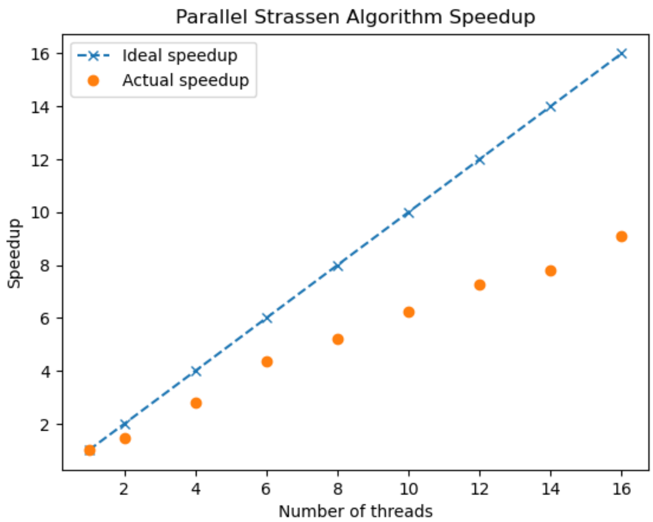


Figure 5: Speed up using different number of threads (n=2048)

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 Strasssen 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

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

void allocate_mem(int*** arr, int n);
void free_mem(int** arr, int n);
int rand_int(int N);
int ** naive_mul(int n, int **a, int **b);
bool check(int n, int **mat1, int **mat2);
void print_matrix(int N, int **a);
int ** sum(int n, int **a, int **b, bool sum);
int ** divide_matrix(int n, int **a, int i, int j);
int ** strassen(int n, int **a, int **b);
int ** combine_matrix(int n, int **a, int **b, int **c, int **d);


int main(int argc, char *argv[])
{
  if (argc != 2) {
    printf("Usage: %s <N>\n", argv[0]);
    return -1;
  }

  int i, j, n;
  int **a;
  int **b;
  int **c;
  int **d;
  int Nmax = 10; // random numbers in [0, N]

  n = atoi(argv[1]); // get matrix size from command line

  printf("I am here\n");

  // create matrix A
  allocate_mem(&a, n);
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      a[i][j] = rand_int(Nmax);

  // print matrix A
  // printf("Matrix A is:\n");
  // print_matrix(n, a);

  // create matrix B
  allocate_mem(&b, n);
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      b[i][j] = rand_int(Nmax);

  double start_strassen = omp_get_wtime();

  c = strassen(n, a, b);

  double end_strassen = omp_get_wtime();
  d = naive_mul(n, a, b);

  printf("Naive Strassen Runtime: %lf\n", end_strassen - start_strassen);

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

  free_mem(a, n);
  free_mem(b, n);
  free_mem(c, n);
  free_mem(d, n);
```

```c
  return 0;
}



// allocate memory for a 2D array
void allocate_mem(int*** arr, int n)
{
  int i;
  *arr = (int**)malloc(n*sizeof(int*));
  for(i=0; i<n; i++)
    (*arr)[i] = (int*)malloc(n*sizeof(int));
}

// free memory for a 2D array
void free_mem(int** arr, int n)
{
  int i;
  for(i=0; i<n; i++)
    free(arr[i]);
  free(arr);
}

// generate a random interger in [0, N - 1]
int rand_int(int N)
{
  int val = -1;
  while( val < 0 || val >= N )
    {
      val = (int)(N * (double)rand()/RAND_MAX);
    }
  return val;
}

// naive matrix multiplication function, jik has the best loop order
int ** naive_mul(int n, int **a, int **b)
{
  int **c;
  allocate_mem(&c, n);
  int i, j, k;;
  for (j = 0; j < n; j++) {
    for (i = 0; i < n; i++) {
      int sum = 0;
      for (k = 0; k < n; k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] = sum;
    }
  }
  return c;
}

// check the accuracy of the Strassen algorithm
bool check(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;
}

// strassen function
int ** strassen(int n, int **a, int **b)
{

  if (n <= 32) {
    return naive_mul(n, a, b);
  }

  else {
    int new_size = n / 2;
    // create the submatrix
```

```
    int **a11, **a12, **a21, **a22;
    int **b11, **b12, **b21, **b22;
    int **c11, **c12, **c21, **c22;
    int **m1, **m2, **m3, **m4, **m5, **m6, **m7;


    // divide the matrix into 4 sub-matrixs
    a11 = divide_matrix(n, a, 0, 0);
    a12 = divide_matrix(n, a, 0, new_size);
    a21 = divide_matrix(n, a, new_size, 0);
    a22 = divide_matrix(n, a, new_size, new_size);

    b11 = divide_matrix(n, b, 0, 0);
    b12 = divide_matrix(n, b, 0, new_size);
    b21 = divide_matrix(n, b, new_size, 0);
    b22 = divide_matrix(n, b, new_size, new_size);


    int **minus_a = sum(new_size, a12, a22, false);
    int **add_b = sum(new_size, b21, b22, true);
    m7 = strassen(new_size, minus_a, add_b);
    free_mem(minus_a, new_size);
    free_mem(add_b, new_size);

    int **minus_a1 = sum(new_size, a21, a11, false);
    int **add_b1 = sum(new_size, b11, b12, true);
    m6 = strassen(new_size, minus_a1, add_b1);
    free_mem(minus_a1, new_size);
    free_mem(add_b1, new_size);

    int **add_a = sum(new_size, a11, a12, true);
    m5 = strassen(new_size, add_a, b22);
    free_mem(add_a, new_size);

    int **minus_b = sum(new_size, b21, b11, false);
    m4 = strassen(new_size, a22, minus_b);
    free_mem(minus_b, new_size);

    int **minus_b1 = sum(new_size, b12, b22, false);
    m3 = strassen(new_size, a11, minus_b1);
    free_mem(minus_b1, new_size);

    int **add_a1 = sum(new_size, a21, a22, true);
    m2 = strassen(new_size, add_a1, b11);
    free_mem(add_a1, new_size);

    int **add_a2 = sum(new_size, a11, a22, true);
    int **add_b2 = sum(new_size, b11, b22, true);
    m1 = strassen(new_size, add_a2, add_b2);
    free_mem(add_a2, new_size);
    free_mem(add_b2, new_size);

    free_mem(a11, new_size);
    free_mem(a12, new_size);
    free_mem(a21, new_size);
    free_mem(a22, new_size);
    free_mem(b11, new_size);
    free_mem(b12, new_size);
    free_mem(b21, new_size);
    free_mem(b22, new_size);

    int **add_m17 = sum(new_size, m1, m7, true);
    int **sub_m45 = sum(new_size, m4, m5, false);
    c11 = sum(new_size, add_m17, sub_m45, true);
    free_mem(add_m17, new_size);
    free_mem(sub_m45, new_size);

    c12 = sum(new_size, m3, m5, true);

    c21 = sum(new_size, m2, m4, true);

    int **sub_m12 = sum(new_size, m1, m2, false);
    int **sum_m36 = sum(new_size, m3, m6, true);
    c22 = sum(new_size, sub_m12, sum_m36, true);
    free_mem(sub_m12, new_size);
    free_mem(sum_m36, new_size);
```

```c

    free_mem(m1, new_size);
    free_mem(m2, new_size);
    free_mem(m3, new_size);
    free_mem(m4, new_size);
    free_mem(m5, new_size);
    free_mem(m6, new_size);
    free_mem(m7, new_size);

    // combine the submatrix
    int **c;
    c = combine_matrix(new_size, c11, c12, c21, c22);

    free_mem(c11, new_size);
    free_mem(c12, new_size);
    free_mem(c21, new_size);
    free_mem(c22, new_size);

    return c;
  }
}

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 ** sum(int n, int **a, int **b, bool sum)
{
  int i, j;
  int **result;
  allocate_mem(&result, n);
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      if (sum) {
        result[i][j] = a[i][j] + b[i][j];
      }
      else {
        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;
    allocate_mem(&result, 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;
}

int ** combine_matrix(int n, int **a, int **b, int **c, int **d)
{
  int size = n * 2;
  int **result;
  allocate_mem(&result, size);
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      if (i < n && j < n) {
        result[i][j] = a[i][j];
      }
```

```
      else if (i < n) {
        result[i][j] = b[i][j-n];
      }
      else if (j < n) {
        result[i][j] = c[i-n][j];
      }
      else {
        result[i][j] = d[i-n][j-n];
      }
    }
  }
  return result;
}
```

Parallel with Openmp

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

void allocate_mem(int*** arr, int n);
void free_mem(int** arr, int n);
int rand_int(int N);
int ** naive_mul(int n, int **a, int **b);
bool check(int n, int **mat1, int **mat2);
void print_matrix(int N, int **a);
int ** sum(int n, int **a, int **b, bool sum);
int ** divide_matrix(int n, int **a, int i, int j);
int ** strassen(int n, int **a, int **b);
int ** combine_matrix(int n, int **a, int **b, int **c, int **d);
int check_matrix(int N);
int next_power_of_2(int N);


int main(int argc, char *argv[])
{
  if (argc != 3) {
    printf("Usage: %s <N> <number of threads>\n", argv[0]);
    return -1;
  }

  int i, j, n, N;
  int **a;
  int **b;
  int **c;
  int **d;
  int Nmax = 10; // random numbers in [0, N]

  n = atoi(argv[1]); // get matrix size from command line
  N = atoi(argv[2]);

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

  // create matrix A
  allocate_mem(&a, n);
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      a[i][j] = rand_int(Nmax);

  // print matrix A
  // printf("Matrix A is:\n");
  // print_matrix(n, a);

  // create matrix B
  allocate_mem(&b, n);
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < n ; j++ )
      b[i][j] = rand_int(Nmax);

  double start_strassen = omp_get_wtime();

  #pragma omp parallel num_threads(N)
  {
    #pragma omp single
    {
      c = strassen(n, a, b);
    }
  }

  double end_strassen = omp_get_wtime();
  d = naive_mul(n, a, b);

  printf("Parallel Strassen Runtime: %lf\n", end_strassen - start_strassen);

  // check the accuracy of the Strassen algorithm
  bool res = check(n, c, d);
```

```c
  if (res == true) {
      printf("The dimension of the matrix is %d, and the result is right!\n", n);
  }

  free_mem(a, n);
  free_mem(b, n);
  free_mem(c, n);
  free_mem(d, n);
  return 0;
}



// allocate memory for a 2D array
void allocate_mem(int*** arr, int n)
{
  int i;
  *arr = (int**)malloc(n*sizeof(int*));
  for(i=0; i<n; i++)
    (*arr)[i] = (int*)malloc(n*sizeof(int));
}

// free memory for a 2D array
void free_mem(int** arr, int n)
{
  int i;
  for(i=0; i<n; i++)
    free(arr[i]);
  free(arr);
}

// generate a random interger in [0, N - 1]
int rand_int(int N)
{
  int val = -1;
  while( val < 0 || val >= N )
    {
      val = (int)(N * (double)rand()/RAND_MAX);
    }
  return val;
}

// naive matrix multiplication function, jik has the best loop order
int ** naive_mul(int n, int **a, int **b)
{
  int **c;
  allocate_mem(&c, n);
  int i, j, k;

  #pragma omp parallel for collapse(2)
    for (j = 0; j < n; j++) {
      for (i = 0; i < n; i++) {
        int sum = 0;
        for (k = 0; k < n; k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
      }
    }

  return c;
}

// check the accuracy of the Strassen algorithm
bool check(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;
}

```

```
// strassen function
int ** strassen(int n, int **a, int **b)
{

  if (n <= 32) {
    return naive_mul(n, a, b);
  }

  else {
    int new_size = n / 2;
    // create the submatrix
    int **a11, **a12, **a21, **a22;
    int **b11, **b12, **b21, **b22;
    int **c11, **c12, **c21, **c22;
    int **m1, **m2, **m3, **m4, **m5, **m6, **m7;


    // divide the matrix into 4 sub-matrixs
    a11 = divide_matrix(n, a, 0, 0);
    a12 = divide_matrix(n, a, 0, new_size);
    a21 = divide_matrix(n, a, new_size, 0);
    a22 = divide_matrix(n, a, new_size, new_size);

    b11 = divide_matrix(n, b, 0, 0);
    b12 = divide_matrix(n, b, 0, new_size);
    b21 = divide_matrix(n, b, new_size, 0);
    b22 = divide_matrix(n, b, new_size, new_size);

    #pragma omp task shared(m7)
    {
      int **minus_a = sum(new_size, a12, a22, false);
      int **add_b = sum(new_size, b21, b22, true);
      m7 = strassen(new_size, minus_a, add_b);
      free_mem(minus_a, new_size);
      free_mem(add_b, new_size);
    }

    #pragma omp task shared(m6)
    {
      int **minus_a1 = sum(new_size, a21, a11, false);
      int **add_b1 = sum(new_size, b11, b12, true);
      m6 = strassen(new_size, minus_a1, add_b1);
      free_mem(minus_a1, new_size);
      free_mem(add_b1, new_size);
    }

    #pragma omp task shared(m5)
    {
      int **add_a = sum(new_size, a11, a12, true);
      m5 = strassen(new_size, add_a, b22);
      free_mem(add_a, new_size);
    }

    #pragma omp task shared(m4)
    {
      int **minus_b = sum(new_size, b21, b11, false);
      m4 = strassen(new_size, a22, minus_b);
      free_mem(minus_b, new_size);
    }

    #pragma omp task shared(m3)
    {
      int **minus_b1 = sum(new_size, b12, b22, false);
      m3 = strassen(new_size, a11, minus_b1);
      free_mem(minus_b1, new_size);
    }

    #pragma omp task shared(m2)
    {
      int **add_a1 = sum(new_size, a21, a22, true);
      m2 = strassen(new_size, add_a1, b11);
      free_mem(add_a1, new_size);
    }

    #pragma omp task shared(m1)
    {
```

```c
      int **add_a2 = sum(new_size, a11, a22, true);
      int **add_b2 = sum(new_size, b11, b22, true);
      m1 = strassen(new_size, add_a2, add_b2);
      free_mem(add_a2, new_size);
      free_mem(add_b2, new_size);
    }

    #pragma omp taskwait

    free_mem(a11, new_size);
    free_mem(a12, new_size);
    free_mem(a21, new_size);
    free_mem(a22, new_size);
    free_mem(b11, new_size);
    free_mem(b12, new_size);
    free_mem(b21, new_size);
    free_mem(b22, new_size);

    #pragma omp task shared(c11)
    {
      int **add_m17 = sum(new_size, m1, m7, true);
      int **sub_m45 = sum(new_size, m4, m5, false);
      c11 = sum(new_size, add_m17, sub_m45, true);
      free_mem(add_m17, new_size);
      free_mem(sub_m45, new_size);
    }

    #pragma omp task shared(c12)
    {
      c12 = sum(new_size, m3, m5, true);
    }

    #pragma omp task shared(c21)
    {
      c21 = sum(new_size, m2, m4, true);
    }

    #pragma omp task shared(c22)
    {
      int **sub_m12 = sum(new_size, m1, m2, false);
      int **sum_m36 = sum(new_size, m3, m6, true);
      c22 = sum(new_size, sub_m12, sum_m36, true);
      free_mem(sub_m12, new_size);
      free_mem(sum_m36, new_size);
    }

    #pragma omp taskwait

    free_mem(m1, new_size);
    free_mem(m2, new_size);
    free_mem(m3, new_size);
    free_mem(m4, new_size);
    free_mem(m5, new_size);
    free_mem(m6, new_size);
    free_mem(m7, new_size);

    // combine the submatrix
    int **c;
    c = combine_matrix(new_size, c11, c12, c21, c22);

    free_mem(c11, new_size);
    free_mem(c12, new_size);
    free_mem(c21, new_size);
    free_mem(c22, new_size);

    return c;
  }
}

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 ** sum(int n, int **a, int **b, bool sum)
{
  int i, j;
  int **result;
  allocate_mem(&result, n);
  // #pragma omp parallel for collapse(2)
    for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
        if (sum) {
          result[i][j] = a[i][j] + b[i][j];
        }
        else {
          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;
    allocate_mem(&result, 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;
}

int ** combine_matrix(int n, int **a, int **b, int **c, int **d)
{
  int size = n * 2;
  int **result;
  allocate_mem(&result, size);
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      if (i < n && j < n) {
        result[i][j] = a[i][j];
      }
      else if (i < n) {
        result[i][j] = b[i][j-n];
      }
      else if (j < n) {
        result[i][j] = c[i-n][j];
      }
      else {
        result[i][j] = d[i-n][j-n];
      }
    }
  }
  return result;
}

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;
}
```