

Московский Авиационный Институт  
(Национальный Исследовательский Университет)  
Институт №8 “Компьютерные науки и прикладная математика”  
Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №2 по курсу**  
**«Операционные системы»**

Группа: М8О-210БВ-24

Студент: Белков А.Д.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_

Дата: 16.10.25

Москва, 2025

# Постановка задачи

## Вариант 6.

Произвести перемножение 2-ух матриц, содержащих комплексные числа.

## Общий метод и алгоритм решения

Использованные системные вызовы:

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine)(void*), void* arg);` - создаёт поток с заданными атрибутами, который начинает выполнение функции `start_routine`.
- `int pthread_join(pthread_t thread, void **retval);` - ожидает завершения указанного потока
- `int clock_gettime(clockid_t clk_id, struct timespec* tp);` - получает текущее монотонное время системы  
`struct timespec`  
    `time_t tv_sec;` - секунды  
    `long tv_nsec;` - наносекунды  
};

Я реализовал программу, которая использует многопоточность для перемножения двух квадратных матриц, элементы которых являются комплексными числами. Для каждой матрицы задаётся порядок матрицы, после чего её элементы заполняются случайными числами (вещественная и мнимая части от 1 до 9).

Количество потоков передаётся в программу в виде ключа запуска `-t <число потоков>`.

Каждому потоку назначается своя часть строк первой матрицы, которые он должен обработать - таким образом выполняется равномерное распределение нагрузки: общее количество строк матрицы делится на количество потоков, и каждый поток вычисляет результат для своих строк.

Выполнение сложения производится по следующей формуле:

$(a + bi) + (c + di) = (a + c) + (b + d)i$ , где  $(a+c)$  - действительная часть,  $(b+d)$  - мнимая часть.

Выполнение умножения производится по следующей формуле:

$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$ , где  $(ac-bd)$  - действительная часть,  $(ad+bc)$  - мнимая часть.

После завершения всех потоков программа вычисляет также результат в последовательном режиме, чтобы сравнить время выполнения и определить ускорение параллельной версии.

Для измерения времени работы используется системный вызов `clock_gettime()`, а результат ускорения рассчитывается как отношение времени последовательного выполнения к времени параллельного.

Поскольку каждый поток обрабатывает строго определённую часть матрицы и не взаимодействует с общими разделяемыми переменными, между потоками отсутствуют пересечения в доступе к данным. В связи с этим использование механизмов синхронизации (мьютексов) не требуется.

## Код программы

### main.c:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#include <string.h>


typedef struct {

    double re, im;

} Complex;


typedef struct {

    Complex **A;

    Complex **B;

    Complex **C;

    int n;

    int start_row;

    int end_row;

} ThreadArgs;


static Complex add_complex(Complex a, Complex b) {

    Complex res;

    res.re = a.re + b.re;

    res.im = a.im + b.im;

    return res;

}
```

```

static Complex mul_complex(Complex a, Complex b) {

    Complex res;

    res.re = a.re * b.re - a.im * b.im;

    res.im = a.re * b.im + a.im * b.re;

    return res;

}

// Многопоточное умножение части матрицы

static void *multiply_part(void *_args) {

    ThreadArgs *args = (ThreadArgs*)_args;

    for (int i = args->start_row; i < args->end_row; i++) {

        for (int j = 0; j < args->n; j++) {

            Complex sum = {0, 0};

            for (int k = 0; k < args->n; k++)

                sum = add_complex(sum, mul_complex(args->A[i][k], args->B[k][j]));

            args->C[i][j] = sum;

        }

    }

    return NULL;

}

// Вывод матрицы

static void print_matrix(Complex **Matrix, int n, const char *name) {

    printf("\nМатрица %s:\n", name);

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++)

            printf("(%5.1f + %5.1fi) ", Matrix[i][j].re, Matrix[i][j].im);

        printf("\n");

    }

}

```

```

int main(int argc, char **argv) {

    if (argc != 3) {

        printf("Использование: %s -t <число_потоков>\n", argv[0]);

        return 1;

    }

    int num_threads = atoi(argv[2]);

    if (num_threads <= 0) {

        printf("Ошибка: количество потоков должно быть > 0\n");

        return 1;

    }

    int n;

    printf("Введите порядок квадратных матриц: ");

    if (scanf("%d", &n) != 1 || n <= 0) {

        printf("Ошибка: введите корректное число\n");

        return 1;

    }

    // Выделение памяти для матриц

    Complex **A = malloc(n * sizeof(Complex));

    Complex **B = malloc(n * sizeof(Complex));

    Complex **C = malloc(n * sizeof(Complex)); // Для результатов параллельного
умножения

    Complex **D = malloc(n * sizeof(Complex)); // Для результатов последовательного
умножения

    for (int i = 0; i < n; i++) {

        A[i] = malloc(n * sizeof(Complex));

        B[i] = malloc(n * sizeof(Complex));

        C[i] = malloc(n * sizeof(Complex));

        D[i] = malloc(n * sizeof(Complex));

    }

```

```

srand((unsigned)time(NULL));

for (int i = 0; i < n; i++)

    for (int j = 0; j < n; j++) {

        A[i][j].re = rand() % 10;

        A[i][j].im = rand() % 10;

        B[i][j].re = rand() % 10;

        B[i][j].im = rand() % 10;

    }

print_matrix(A, n, "A");

print_matrix(B, n, "B");

printf("Количество потоков: %d\n", num_threads);

pthread_t *threads = malloc(num_threads * sizeof(pthread_t));

ThreadArgs *thread_args = malloc(num_threads * sizeof(ThreadArgs));

int rows_per_thread = n / num_threads;

struct timespec start, end;

// Параллельное умножение

clock_gettime(CLOCK_MONOTONIC, &start);

for (int t = 0; t < num_threads; t++) {

    thread_args[t] = (ThreadArgs){

        .A = A,

        .B = B,

        .C = C,

        .n = n,

        .start_row = t * rows_per_thread,

        .end_row = (t == num_threads - 1) ? n : (t + 1) * rows_per_thread

    };

```

```

        pthread_create(&threads[t], NULL, multiply_part, &thread_args[t]);
    }

    for (int t = 0; t < num_threads; t++)

        pthread_join(threads[t], NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);

    double time_parallel = (end.tv_sec - start.tv_sec) + (end.tv_nsec -
start.tv_nsec) / 1e9;

    print_matrix(C, n, "Результат умножения (параллельно)");

    // Последовательное умножение

    clock_gettime(CLOCK_MONOTONIC, &start);

    for (int i = 0; i < n; i++)

        for (int j = 0; j < n; j++) {

            Complex sum = {0, 0};

            for (int k = 0; k < n; k++)

                sum = add_complex(sum, mul_complex(A[i][k], B[k][j]));

            D[i][j] = sum;

        }

    clock_gettime(CLOCK_MONOTONIC, &end);

    double seq_time_sec = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec)
/ 1e9;

    print_matrix(D, n, "Результат умножения (последовательно)");

    double speedup = seq_time_sec / time_parallel;

    double efficiency = speedup / num_threads;

    printf("\nВремя работы (параллельно): %.6f секунд\n", time_parallel);

    printf("Время работы (последовательно): %.6f секунд\n", seq_time_sec);

    printf("Ускорение = %.2fx\n", speedup);

    printf("Эффективность = %.4f\n", efficiency);

```

```

    for (int i = 0; i < n; i++) {

        free(A[i]); free(B[i]); free(C[i]); free(D[i]);

    }

    free(A);

    free(B);

    free(C);

    free(D);

    free(threads);

    free(thread_args);

    return 0;

}

```

## Протокол работы программы

### Тестирование:

artmlink@pop-os:~/2\_course\_MAI/MAI-OS-Labs-2025/lab-2\$ ./main -t 2

Введите порядок квадратных матриц: 2

Матрица A:

```

( 3.0 + 5.0i) ( 5.0 + 5.0i)
( 7.0 + 0.0i) ( 4.0 + 6.0i)

```

Матрица B:

```

( 6.0 + 8.0i) ( 5.0 + 0.0i)
( 9.0 + 6.0i) ( 8.0 + 9.0i)

```

Количество потоков: 2

Матрица Результат умножения (параллельно):

```

( -7.0 + 129.0i) ( 10.0 + 110.0i)
( 42.0 + 134.0i) ( 13.0 + 84.0i)

```

Матрица Результат умножения (последовательно):

```

( -7.0 + 129.0i) ( 10.0 + 110.0i)
( 42.0 + 134.0i) ( 13.0 + 84.0i)

```

Время работы (параллельно): 0.000278 секунд

Время работы (последовательно): 0.000001 секунд



Ускорение = 0.00x

Эффективность = 0.0015

```
artmlink@pop-os:~/2_course_MAI/MAI-OS-Labs-2025/lab-2$ ./main -t 2
Введите порядок квадратных матриц: 2

Матрица A:
( 3.0 + 5.0i) ( 5.0 + 5.0i)
( 7.0 + 0.0i) ( 4.0 + 6.0i)

Матрица B:
( 6.0 + 8.0i) ( 5.0 + 0.0i)
( 9.0 + 6.0i) ( 8.0 + 9.0i)
Количество потоков: 2

Матрица Результат умножения (параллельно):
( -7.0 + 129.0i) ( 10.0 + 110.0i)
( 42.0 + 134.0i) ( 13.0 + 84.0i)

Матрица Результат умножения (последовательно):
( -7.0 + 129.0i) ( 10.0 + 110.0i)
( 42.0 + 134.0i) ( 13.0 + 84.0i)

Время работы (параллельно): 0.000278 секунд
Время работы (последовательно): 0.000001 секунд
Ускорение = 0.00x
Эффективность = 0.0015
```

```
artmlink@pop-os:~/2_course_MAI/MAI-OS-Labs-2025/lab-2$ ./main -t 4
Введите порядок квадратных матриц: 3
```

Матрица A:

```
( 8.0 + 6.0i) ( 7.0 + 4.0i) ( 6.0 + 0.0i)
( 4.0 + 0.0i) ( 7.0 + 8.0i) ( 2.0 + 0.0i)
( 7.0 + 4.0i) ( 0.0 + 4.0i) ( 2.0 + 3.0i)
```

Матрица B:

```
( 3.0 + 6.0i) ( 4.0 + 4.0i) ( 0.0 + 3.0i)
( 5.0 + 6.0i) ( 5.0 + 3.0i) ( 5.0 + 3.0i)
( 1.0 + 0.0i) ( 7.0 + 9.0i) ( 7.0 + 0.0i)
Количество потоков: 4
```

Матрица Результат умножения (параллельно):

```
( 5.0 + 128.0i) ( 73.0 + 151.0i) ( 47.0 + 65.0i)
( 1.0 + 106.0i) ( 41.0 + 95.0i) ( 25.0 + 73.0i)
(-25.0 + 77.0i) (-13.0 + 103.0i) (-10.0 + 62.0i)
```

Матрица Результат умножения (последовательно):

```
( 5.0 + 128.0i) ( 73.0 + 151.0i) ( 47.0 + 65.0i)
( 1.0 + 106.0i) ( 41.0 + 95.0i) ( 25.0 + 73.0i)
(-25.0 + 77.0i) (-13.0 + 103.0i) (-10.0 + 62.0i)
```

Время работы (параллельно): 0.000358 секунд

Время работы (последовательно): 0.000002 секунд

Ускорение = 0.00x

Эффективность = 0.0011

```
artmlink@pop-os:~/2_course_MAI/MAI-OS-Labs-2025/lab-2$ ./main -t 4
```

Введите порядок квадратных матриц: 3

Матрица A:

```
( 8.0 + 6.0i) ( 7.0 + 4.0i) ( 6.0 + 0.0i)
( 4.0 + 0.0i) ( 7.0 + 8.0i) ( 2.0 + 0.0i)
( 7.0 + 4.0i) ( 0.0 + 4.0i) ( 2.0 + 3.0i)
```

Матрица B:

```
( 3.0 + 6.0i) ( 4.0 + 4.0i) ( 0.0 + 3.0i)
( 5.0 + 6.0i) ( 5.0 + 3.0i) ( 5.0 + 3.0i)
( 1.0 + 0.0i) ( 7.0 + 9.0i) ( 7.0 + 0.0i)
```

Количество потоков: 4

Матрица Результат умножения (параллельно):

```
( 5.0 + 128.0i) ( 73.0 + 151.0i) ( 47.0 + 65.0i)
( 1.0 + 106.0i) ( 41.0 + 95.0i) ( 25.0 + 73.0i)
(-25.0 + 77.0i) (-13.0 + 103.0i) (-10.0 + 62.0i)
```

Матрица Результат умножения (последовательно):

```
( 5.0 + 128.0i) ( 73.0 + 151.0i) ( 47.0 + 65.0i)
( 1.0 + 106.0i) ( 41.0 + 95.0i) ( 25.0 + 73.0i)
(-25.0 + 77.0i) (-13.0 + 103.0i) (-10.0 + 62.0i)
```

Время работы (параллельно): 0.000358 секунд

Время работы (последовательно): 0.000002 секунд

Ускорение = 0.00x

Эффективность = 0.0011

$$\begin{pmatrix} 8+6\cdot i & 7+4\cdot i & 6 \\ 4 & 7+8\cdot i & 2 \\ 7+4\cdot i & 0+4\cdot i & 2+3\cdot i \end{pmatrix} \cdot \begin{pmatrix} 3+6\cdot i & 4+4\cdot i & 0+3\cdot i \\ 5+6\cdot i & 5+3\cdot i & 5+3\cdot i \\ 1 & 7+9\cdot i & 7 \end{pmatrix} =$$
$$\begin{pmatrix} 5+128i & 73+151i & 47+65i \\ 1+106i & 41+95i & 25+73i \\ -25+77i & -13+103i & -10+62i \end{pmatrix}$$

## Вывод

В ходе выполнения данной лабораторной работы было изучено применение многопоточности для ускорения вычислений при умножении квадратных матриц, содержащих комплексные числа. Были проанализированы время работы параллельного и последовательного алгоритмов, а также вычислено ускорение и эффективность при различном числе потоков. Также были получены следующие выводы, которые удобно изобразить в виде таблицы, условимся что на ввод подаётся 512 потоков, чтобы явно продемонстрировать изменение ключевых значений.

Число потоков	Время исполнения параллельно(с)	Время исполнения последовательно (с)	Ускорение	Эффективность
1	1.084627	0.954912	0.88x	0.8804
2	0.573130	0.926837	1.62x	0.8086
4	0.355643	0.913884	2.57x	0.6424
8	0.302910	0.937856	3.10x	0.3870
16	0.213701	0.910988	4.26x	0.2664
512	0.178692	0.923636	5.17x	0.0101
1024	1.109201	1.057515	0.95x	0.0009
8192	1.168172	1.208539	1.03x	0.0001
16384	1.236852	1.080107	0.87x	0.0001

Расчёты:

- Ускорение (speedup):  $S = T_{seq}/T_{par}$ , где  $T_{seq}$  - время выполнения последовательного алгоритма, а  $T_{par}$  - время выполнения параллельного алгоритма с N-ым количеством потоков.
- Эффективность (efficiency):  $E = S/N$ , где  $a$  ускорение,  $N$  - число потоков.

Анализ результатов:

1. Количество потоков **меньше** числа логических ядер (1-8 потоков):
  - Ускорение растёт с увеличением числа потоков: 0.88x при 1 потоке и 3.10x при 8 потоках.
  - Эффективность постепенно снижается, что объясняется накладными расходами на создание и завершение потоков, но остаётся высокой для небольшого числа потоков (0.387-0.8804).
  - Потоки работают независимо, так как каждый обрабатывает свою часть строк матрицы, поэтому race conditions отсутствуют, мьютексы не требуются.

**Вывод по 1-му блоку:** В этом диапазоне многопоточность даёт наилучший прирост производительности с минимальными накладными расходами.

2. Количество потоков **равно** числу логических ядер (16 потоков):
  - Ускорение увеличилось до 4.26x, но эффективность упала до 0.2664.
  - Начинают проявляться эффекты конкуренции за ресурсы и накладные расходы на переключение контекста.

**Вывод по 2-му блоку:** Достигнут практический предел эффективности использования CPU, дальнейшее увеличение потоков даёт незначительное улучшение или даже падение производительности.

3. Количество потоков **значительно больше** числа логических ядер (16+ потоков):
  - Эффективность резко падает почти до нуля (0.0101 при 512 потоках, 0.0001 при 8192 и 16384 потоках)
  - Ускорение перестаёт расти и падает ниже 1x при 1024+ потоках
  - Причины деградации:
    - накладные ресурсы на создание и управление потоками **превышают вычислительную пользу**;
    - снижение эффективности использования памяти;
    - планировщик ОС перегружен, переключения контекста занимают много времени;

**Вывод по 3-му блоку:** При чрезмерно большом количестве потоков производительность деградирует - накладные расходы полностью перевешивают выигрыш от параллелизма.