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-211БВ-24

Студент: ПРОХОРОВ В.И.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 16.02.26

Москва, 2026

Постановка задачи

Вариант 13.

Наложить K раз фильтр, использующий матрицу свертки, на матрицу, состоящую из вещественных чисел. Размер окна задается пользователем

Общий метод и алгоритм решения

Использованные системные вызовы:

- time_t time(time_t *tloc) – получение текущего времени.
- int clock_gettime(clockid_t clockid, struct timespec *tp) – получение точного времени для замера производительности
- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg) - создание нового потока.
- int pthread_join(pthread_t thread, void **retval) – ожидание завершения потоков.

Идея свёртки: для каждой ячейки матрицы считаем среднее значение в окне 3×3 учитывая границы. Используем два буфера: читаем из одного, пишем в другой, потом меняем их местами чтобы избежать копирование.

В последовательной версии один поток K раз проходит всю матрицу и считает среднее по окну.

В параллельной версии матрица делится по строкам между потоками. Каждый поток независимо делает все K итераций только над своим куском строк, локально меняет буферы после каждой итерации. Поскольку потоки не пересекаются по записи, синхронизация не нужна. Главный поток ждёт завершения всех и получает общее время.

Для оценки работы программы использовались метрики Ускорения (S) и Эффективности (X), которые считаются по формулам (Ts, Tp – время в мс):

Ускорение: $S = T_s / T_p$, где

- T_s – время последовательной реализации,
- T_p – время параллельной реализации,
- $1 <= S <= p$,
- p – количество ядер

Эффективность: $X = S / p$, где $X < 1$

У меня получилось Ts = 462.39 мс, и максимальная эффективность <= 1 (из за overhead)

Число потоков	Время (мс)	Ускорение S	Эффективность X
1	461.65	1.00	1.00
2	240.34	1.92	0.96
4	122.63	3.77	0.94
6	100.69	4.59	0.77
8	95.08	4.86	0.61
16	96.17	4.81	0.30
64	84.51	5.47	0.09
128	83.95	5.51	0.04
512	87.51	5.28	0.01

Код программы

sequential.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv) {
    if (argc != 3) {
        fprintf(stderr, "Usage: <filename> <N> <K>\n");
        return 1;
    }

    int N = atoi(argv[1]);
    int K = atoi(argv[2]);
    if (N < 3 || K < 1) {
        fprintf(stderr, "N >= 3 and K >= 1 required\n");
        return 1;
    }

    // Выделяем две матрицы (ping-pong)
    float **A = malloc(N * sizeof(float *));
    float **B = malloc(N * sizeof(float *));
    for (int i = 0; i < N; i++) {
        A[i] = malloc(N * sizeof(float));
        B[i] = malloc(N * sizeof(float));
    }

    // Заполняем случайными значениями [0, 1]
    srand(time(NULL));
```

```
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = (float)rand() / RAND_MAX;

    // Замеряем время
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Если K нечетное то out, иначе in
    float **in = A;
    float **out = B;

    for (int k = 0; k < K; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {

                float sum = 0.0;
                int count = 0;

                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {

                        int ni = i + di;
                        int nj = j + dj;

                        // Считаем сумму
                        if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                            sum += in[ni][nj];
                            count++;
                        }
                    }
                }
                out[i][j] = sum / count; // Считаем среднее
            }
        }
        // Меняем указатели
        float **tmp = in;
        in = out;
        out = tmp;
    }

    clock_gettime(CLOCK_MONOTONIC, &end); // Конец подсчета времени
    double time_ms = (end.tv_sec - start.tv_sec) * 1000.0 +
                     (end.tv_nsec - start.tv_nsec) / 1e6;

    printf("=== Sequential version ===\n");
    printf("N = %d, K = %d\n", N, K);
    printf("Time: %.2f ms\n", time_ms);
```

```
    for (int i = 0; i < N; i++) {
        free(A[i]);
        free(B[i]);
    }
    free(A);
    free(B);

    return 0;
}
```

parallel.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef struct {
    float **in;
    float **out;
    int start_row;
    int end_row;
    int N;
    int K;
} ThreadData;

static void *thread_work(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    float **in = data->in;
    float **out = data->out;
    int N = data->N;

    for (int k = 0; k < data->K; k++) {
        for (int i = data->start_row; i < data->end_row; i++) {
            for (int j = 0; j < N; j++) {

                float sum = 0.0f;
                int count = 0;
                // Квадрат 3x3
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {

                        int ni = i + di;
                        int nj = j + dj;
```

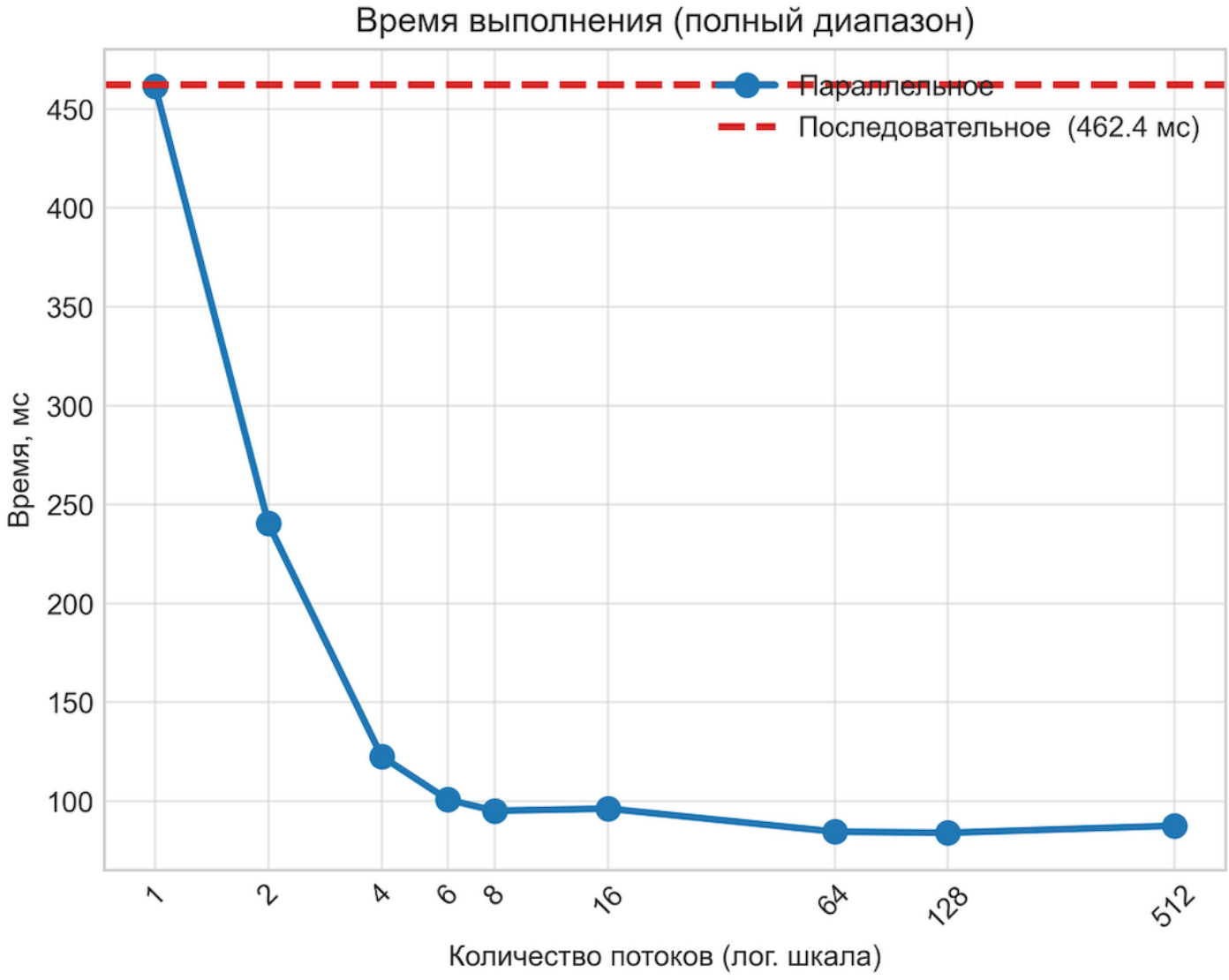
```
                        if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                            sum += in[ni][nj];
                            count++;
                        }
                    }
                }
                out[i][j] = sum / count;
            }
        }
        // Меняем указатели
        float **tmp = in;
        in = out;
        out = tmp;
    }
    return NULL;
}

int main(int argc, char **argv) {
    if (argc != 4) {
        fprintf(stderr, "Usage: <filename> <N> <K> <max_threads>\n");
        return 1;
    }

    int N = atoi(argv[1]);
    int K = atoi(argv[2]);
    int max_threads = atoi(argv[3]);

    if (N < 3 || K < 1 || max_threads < 1) {
        fprintf(stderr, "N >= 3, K >= 1, max_threads >= 1 required\n");
        return 1;
    }

    // Ограничиваем количество потоков
    int num_threads = max_threads;
    if (num_threads > N)
        num_threads = N; // не больше чем количество строк

    printf("=== Parallel version ===\n");
    printf("N = %d, K = %d, threads = %d\n", N, K, num_threads);

    // Выделяем матрицы
    float **A = malloc(N * sizeof(float *));
    float **B = malloc(N * sizeof(float *));
    for (int i = 0; i < N; i++) {
        A[i] = malloc(N * sizeof(float));
        B[i] = malloc(N * sizeof(float));
    }
```

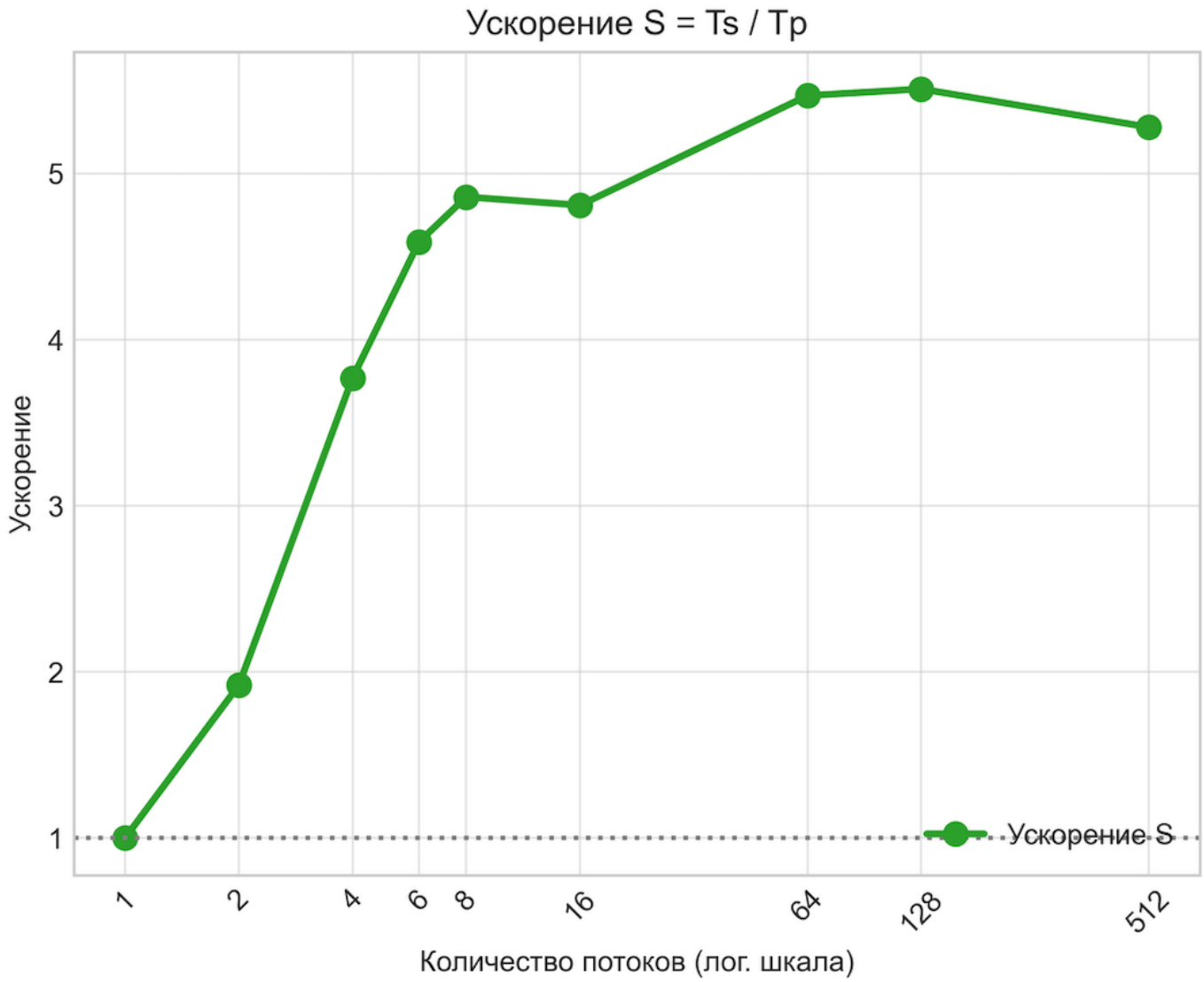
```
    srand(time(NULL));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = (float)rand() / RAND_MAX;

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Создание потоков
    pthread_t *threads = malloc(num_threads * sizeof(pthread_t));
    ThreadData *tdata = malloc(num_threads * sizeof(ThreadData));

    float **in = A;
    float **out = B;

    int rows_per_thread = N / num_threads;
    int remainder = N % num_threads;

    for (int t = 0; t < num_threads; t++) {
        int start_row = t * rows_per_thread + (t < remainder ? t : remainder);
        int end_row = start_row + rows_per_thread + (t < remainder ? 1 : 0);

        tdata[t] = (ThreadData){.in = in,
                                .out = out,
                                .start_row = start_row,
                                .end_row = end_row,
                                .N = N,
                                .K = K};

        // Создание потока
        pthread_create(&threads[t], NULL, thread_work, &tdata[t]);
    }

    // Ожидание завершения потоков
    for (int t = 0; t < num_threads; t++)
        pthread_join(threads[t], NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);
    double time_ms = (end.tv_sec - start.tv_sec) * 1000.0 +
                     (end.tv_nsec - start.tv_nsec) / 1e6;

    printf("Time: %.2f ms\n", time_ms);

    for (int i = 0; i < N; i++) {
        free(A[i]);
        free(B[i]);
    }
    free(A);
```

```
    free(B);
    free(threads);
    free(tdata);

    return 0;
}
```

Протокол работы программы

Число потоков	Время (мс)	Ускорение S	Эффективность X
1	461.65	1.00	1.00
2	240.34	1.92	0.96
4	122.63	3.77	0.94
6	100.69	4.59	0.77
8	95.08	4.86	0.61
16	96.17	4.81	0.30
64	84.51	5.47	0.09
128	83.95	5.51	0.04
512	87.51	5.28	0.01






```
(.venv) ➜  lab_2 git:(main) ✗ ./sequential 512 64
=== Sequential version ===
N = 512, K = 64
Time: 457.25 ms
```

```
(.venv) ➜  lab_2 git:(main) ✗ make run
./parallel 512 64 1
=== Parallel version ===
N = 512, K = 64, threads = 1
Time: 456.18 ms
./parallel 512 64 2
=== Parallel version ===
N = 512, K = 64, threads = 2
Time: 240.66 ms
./parallel 512 64 4
=== Parallel version ===
N = 512, K = 64, threads = 4
Time: 123.00 ms
```

```
./parallel 512 64 6
=== Parallel version ===
N = 512, K = 64, threads = 6
Time: 106.56 ms
./parallel 512 64 8
=== Parallel version ===
N = 512, K = 64, threads = 8
Time: 89.64 ms
./parallel 512 64 16
=== Parallel version ===
N = 512, K = 64, threads = 16
Time: 91.66 ms
./parallel 512 64 64
=== Parallel version ===
N = 512, K = 64, threads = 64
Time: 83.30 ms
./parallel 512 64 128
=== Parallel version ===
N = 512, K = 64, threads = 128
Time: 82.95 ms
./parallel 512 64 512
=== Parallel version ===
N = 512, K = 64, threads = 512
Time: 85.29 ms
```

Вывод

При выполнении данной лабораторной работы я научился работать с потоками в macos, а также их синхронизировать. Из тестов я понял что при увеличении количества потоков, программа работает быстрее: с 462 мс в последовательной версии время падает до 84-96 мс уже при 8-16 потоках и достигает минимума около 84 мс при 128 потоках. Ускорение растёт не строго пропорционально числу потоков при запуске программы с 1-4 потоками, эффективность остаётся высокой (0.9–0.96). После 8 потоков (числа логических ядер) прирост замедляется, эффективность стремительно падает, а начиная с 16 потоков добавление новых потоков уже почти не даёт выигрыша или даже слегка ухудшает результат. Максимальное ускорение (5.51 раза) наблюдается при 128 потоках, хотя эффективность крайне мала (0.04). Дальнейшее увеличение до 512 потоков уже приводит к небольшому ухудшению из-за накладных расходов (overhead). Лучшее соотношение Ускорения и Эффективности можно заметить при p = 6/8 потоков, из-за того, что на моем pc 8 логических ядер. Параллелизация даёт хороший прирост производительности, но до определённого значения, после которого выгода почти исчезает, а в некоторых случаях даже оборачивается потерями из-за накладных расходов (overhead).