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-212БВ-24

Студент: Брежнев Г. О.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 31.10.25

Москва, 2025

Постановка задачи

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы. Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы. В отчете привести исследование зависимости ускорения и эффективности алгорит

Вариант 8.

Есть K массивов одинаковой длины. Необходимо сложить эти массивы. Необходимо предусмотреть стратегию, адаптирующуюся под количество массивов и их длину (по количеству операций)

Общий метод и алгоритм решения

Использованные функции:

- pthread_create() - запуск нового потока с заданной функцией и аргументами.
- pthread_join() - приостанавливает основной поток до завершения указанного рабочего потока.
- sysconf(_SC_NPROCESSORS_ONLN) - узнать количество свободных ядер.

Алгоритм (adaptive_sum.c):

1. Вычисляем общий объем работы: K × N (кол-во массивов × размер массивов)
2. Определяем стратегию:
   Если работа < 1000 операций: последовательная версия
   Иначе: параллельная версия
3. Для параллельной версии:

   Вычисляем оптимальное количество потоков: min(работа/1000, ядра процессора, размер массивов). Каждый поток получает свой диапазон индексов результирующего массива.

4. Каждый поток:
   Для каждого своего индекса i вычисляет сумму arrays[0][i] + arrays[1][i] + ... + arrays[K-1][i]
   Записывает результат в result[i]
5. Основной поток ждет завершения всех рабочих потоков
6. Выводим результаты: выбранную стратегию, количество потоков, время выполнения/

Код программы

utils.c

```
#include <pthread.h>
```

```
#include <unistd.h>
#include <string.h>
#include "array_sum.h"



void *thread_func(void *arguments) {
    ThreadArgs *args = (ThreadArgs*)arguments;

    for (int i = args->start_index; i < args->end_index; i++) {
        double sum = 0.0;
        for (int j = 0; j < args->array_count; j++) {
            sum += args->arrays[j][i];
        }
        args->result[i] = sum;
    }

    return NULL;
}

void sequential_func(double **arrays, double *result, int array_count, int array_size)
{
    for (int i = 0; i < array_size; i++) {
        result[i] = 0.0;
        for (int j = 0; j < array_count; j++) {
            result[i] += arrays[j][i];
        }
    }
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include "array_sum.h"

#define PER_THREAD 1000
#define MAX_THREADS 16



int main(int argc, char **argv) {
    if (argc != 4) {
        printf("Использование: ./array_sum <количество_массивов> <размер_массивов>
<количество_потоков>\n");
        return 1;
```

```
    }

    int array_count = atoi(argv[1]);
    int array_size = atoi(argv[2]);
    int threads_count = atoi(argv[3]);

    double **arrays = (double**)malloc(sizeof(double*) * array_count);
    double *sequential_res = (double*)malloc(array_size * sizeof(double));
    double *parallel_res = (double*)malloc(array_size * sizeof(double));

    srand(time(NULL));
    for (int i = 0; i < array_count; i++) {
        arrays[i] = (double*)malloc(sizeof(double) * array_size);
        for (int j = 0; j < array_size; j++) {
            arrays[i][j] = rand() % 10000;
        }
    }

    struct timespec end, start;

    clock_gettime(CLOCK_MONOTONIC, &start);
    sequential_func(arrays, sequential_res, array_count, array_size);
    clock_gettime(CLOCK_MONOTONIC, &end);
    double sequential_time = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec -
start.tv_nsec) / 1000000.0;

    clock_gettime(CLOCK_MONOTONIC, &start);

    pthread_t* threads = (pthread_t*)malloc(threads_count * sizeof(pthread_t));
    ThreadArgs* thread_args = (ThreadArgs*)malloc(threads_count * sizeof(ThreadArgs));

    for (int i = 0; i < threads_count; i++) {
        int el_per_thread = array_size / threads_count;

        thread_args[i].start_index = i * el_per_thread;
        thread_args[i].end_index = (i == threads_count - 1) ? array_size : (i + 1) *
el_per_thread;
        thread_args[i].array_count = array_count;
        thread_args[i].array_size = array_size;
        thread_args[i].arrays = arrays;
        thread_args[i].result = parallel_res;

        pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
    }

    for (int i = 0; i < threads_count; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    clock_gettime(CLOCK_MONOTONIC, &end);
    double parallel_time = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec -
start.tv_nsec) / 1000000.0;

    int match = 1;
    for (int i = 0; i < array_size; i++) {
        if (sequential_res[i] != parallel_res[i]) {
            match = 0;
            break;
        }
    }
    /*
    printf("Количество массивов: %d\n", array_count);
    printf("Размер массивов: %d\n", array_size);
    printf("Количество потоков: %d\n", threads_count);
    */
    printf("Последовательная версия: %.2f мс\n", sequential_time);
    printf("Параллельная версия: %.2f мс\n", parallel_time);
    printf("Ускорение: %.2f раз\n", sequential_time / parallel_time);
    printf("Эффективность: %.5f\n", (sequential_time / parallel_time) /
 threads_count);
    // printf("Результаты: %s\n", match ? "совпадают" : "не совпадают");

    for (int i = 0; i < array_count; i++) {
        free(arrays[i]);
    }
    free(arrays);
    free(sequential_res);
    free(parallel_res);
    free(threads);
    free(thread_args);

    return 0;
}
```

adaptive_sum.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include "array_sum.h"

#define MIN_WORK_PER_THREAD 1000

void adaptive_sum(double **arrays, double *result, int array_count, int array_size, int
```

```
 *chosen_threads) {
    int total_work = array_count * array_size;
    int available_cores = sysconf(_SC_NPROCESSORS_ONLN);

    if (total_work < MIN_WORK_PER_THREAD) {
        *chosen_threads = 1;
        sequential_func(arrays, result, array_count, array_size);
    } else {
        int threads_to_use = total_work / MIN_WORK_PER_THREAD;

        if (threads_to_use < 1) { threads_to_use = 1; }
        if (threads_to_use > available_cores) { threads_to_use = available_cores; }
        if (threads_to_use > array_size) { threads_to_use = array_size; }

        *chosen_threads = threads_to_use;

        pthread_t* threads = (pthread_t*)malloc(threads_to_use * sizeof(pthread_t));
        ThreadArgs* thread_args = (ThreadArgs*)malloc(threads_to_use *
sizeof(ThreadArgs));

        int el_per_thread = array_size / threads_to_use;

        for (int i = 0; i < threads_to_use; i++) {
            thread_args[i].start_index = i * el_per_thread;
            thread_args[i].end_index = (i == threads_to_use - 1) ? array_size : (i + 1)
* el_per_thread;
            thread_args[i].array_count = array_count;
            thread_args[i].array_size = array_size;
            thread_args[i].arrays = arrays;
            thread_args[i].result = result;

            pthread_create(&threads[i], NULL, thread_func, &thread_args[i]);
        }

        for (int i = 0; i < threads_to_use; i++) {
            pthread_join(threads[i], NULL);
        }

        free(threads);
        free(thread_args);
    }
}

int main(int argc, char **argv) {
    if (argc != 3) {
        printf("Использование: ./adaptive_sum <количество_массивов>
<размер_массивов>\n");
        return 1;
    }
```

```
    int array_count = atoi(argv[1]);
    int array_size = atoi(argv[2]);

    double **arrays = (double**)malloc(sizeof(double*) * array_count);
    double *result = (double*)malloc(array_size * sizeof(double));

    srand(time(NULL));
    for (int i = 0; i < array_count; i++) {
        arrays[i] = (double*)malloc(sizeof(double) * array_size);
        for (int j = 0; j < array_size; j++) {
            arrays[i][j] = rand() % 10000;
        }
    }

    struct timespec end, start;
    int chosen_threads;

    clock_gettime(CLOCK_MONOTONIC, &start);
    adaptive_sum(arrays, result, array_count, array_size, &chosen_threads);
    clock_gettime(CLOCK_MONOTONIC, &end);
    double adaptive_time = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec -
 start.tv_nsec) / 1000000.0;

    printf("Адаптивная стратегия:\n");
    printf("Количество массивов: %d\n", array_count);
    printf("Размер массивов: %d\n", array_size);
    printf("Выбранная стратегия: %s\n", chosen_threads == 1 ? "последовательная" :
"параллельная");
    printf("Количество потоков: %d\n", chosen_threads);
    printf("Время выполнения: %.2f мс\n", adaptive_time);

    for (int i = 0; i < array_count; i++) {
        free(arrays[i]);
    }
    free(arrays);
    free(result);

    return 0;
}
```

array_sum.h

```
#ifndef ARRAY_SUM_H
#define ARRAY_SUM_H

typedef struct {
    int array_count;
```

```
    int array_size;
    double **arrays;
    double *result;
    int start_index;
    int end_index;
} ThreadArgs;

void *thread_func(void *arguments);
void sequential_func(double **arrays, double *result, int array_count, int array_size);

#endif
```

Протокол работы программы

~/Projects/mai_os_lab/lab2/build main ./array_sum
✔

Использование: ./array_sum <количество_массивов> <размер_массивов> <количество_потоков>

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 2
1 ✘

Последовательная версия: 10.30 мс

Параллельная версия: 4.99 мс

Ускорение: 2.06 раз

Эффективность: 1.03214

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 4
✔

Последовательная версия: 10.73 мс

Параллельная версия: 3.22 мс

Ускорение: 3.33 раз

Эффективность: 0.83365

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 8
✔

Последовательная версия: 10.41 мс

Параллельная версия: 1.89 мс

Ускорение: 5.50 раз

Эффективность: 0.68725

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 12
✔

Последовательная версия: 10.57 мс

Параллельная версия: 2.15 мс

Ускорение: 4.92 раз

Эффективность: 0.41023

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 16
✔

Последовательная версия: 10.35 мс

Параллельная версия: 2.45 мс

Ускорение: 4.22 раз

Эффективность: 0.26377

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 32
✔

Последовательная версия: 11.43 мс

Параллельная версия: 2.38 мс

Ускорение: 4.80 раз

Эффективность: 0.15011

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 64
✔

Последовательная версия: 10.67 мс

Параллельная версия: 3.76 мс

Ускорение: 2.84 раз

Эффективность: 0.04437

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 256
✔

Последовательная версия: 10.85 мс

Параллельная версия: 11.73 мс

Ускорение: 0.93 раз

Эффективность: 0.00361

~/Projects/mai_os_lab/lab2/build main ./array_sum 1000 1000 1024
✔

Последовательная версия: 11.27 мс

Параллельная версия: 48.72 мс

Ускорение: 0.23 раз

Эффективность: 0.00023

~/Projects/mai_os_lab/lab2/build main ./adaptive_sum 10 10
✔

Адаптивная стратегия:

Количество массивов: 10

Размер массивов: 10

Выбранная стратегия: последовательная

Количество потоков: 1

Время выполнения: 0.04 мс

~/Projects/mai_os_lab/lab2/build main ./adaptive_sum 1000 1000
✔

Адаптивная стратегия:

Количество массивов: 1000

Размер массивов: 1000

Выбранная стратегия: параллельная

Количество потоков: 12

Время выполнения: 2.04 мс

~/Projects/mai_os_lab/lab2/build main ./adaptive_sum 1000 100
✔

Адаптивная стратегия:

Количество массивов: 1000

Размер массивов: 100

Выбранная стратегия: параллельная

Количество потоков: 12

Время выполнения: 1.10 мс

~/Projects/mai_os_lab/lab2/build main ./adaptive_sum 100 100
✔

Адаптивная стратегия:

Количество массивов: 100

Размер массивов: 100

Выбранная стратегия: параллельная

Количество потоков: 10

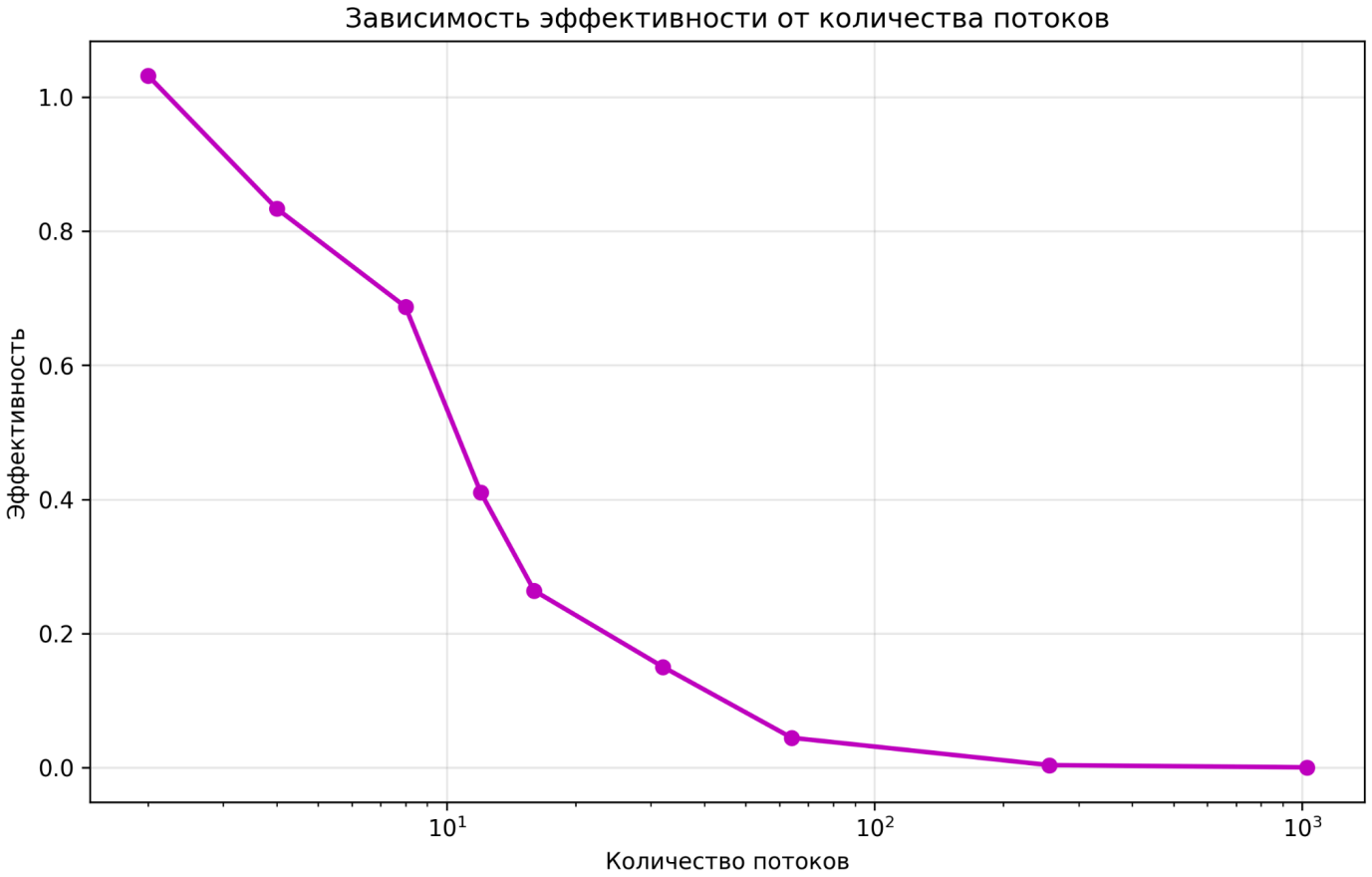
Время выполнения: 0.66 мс

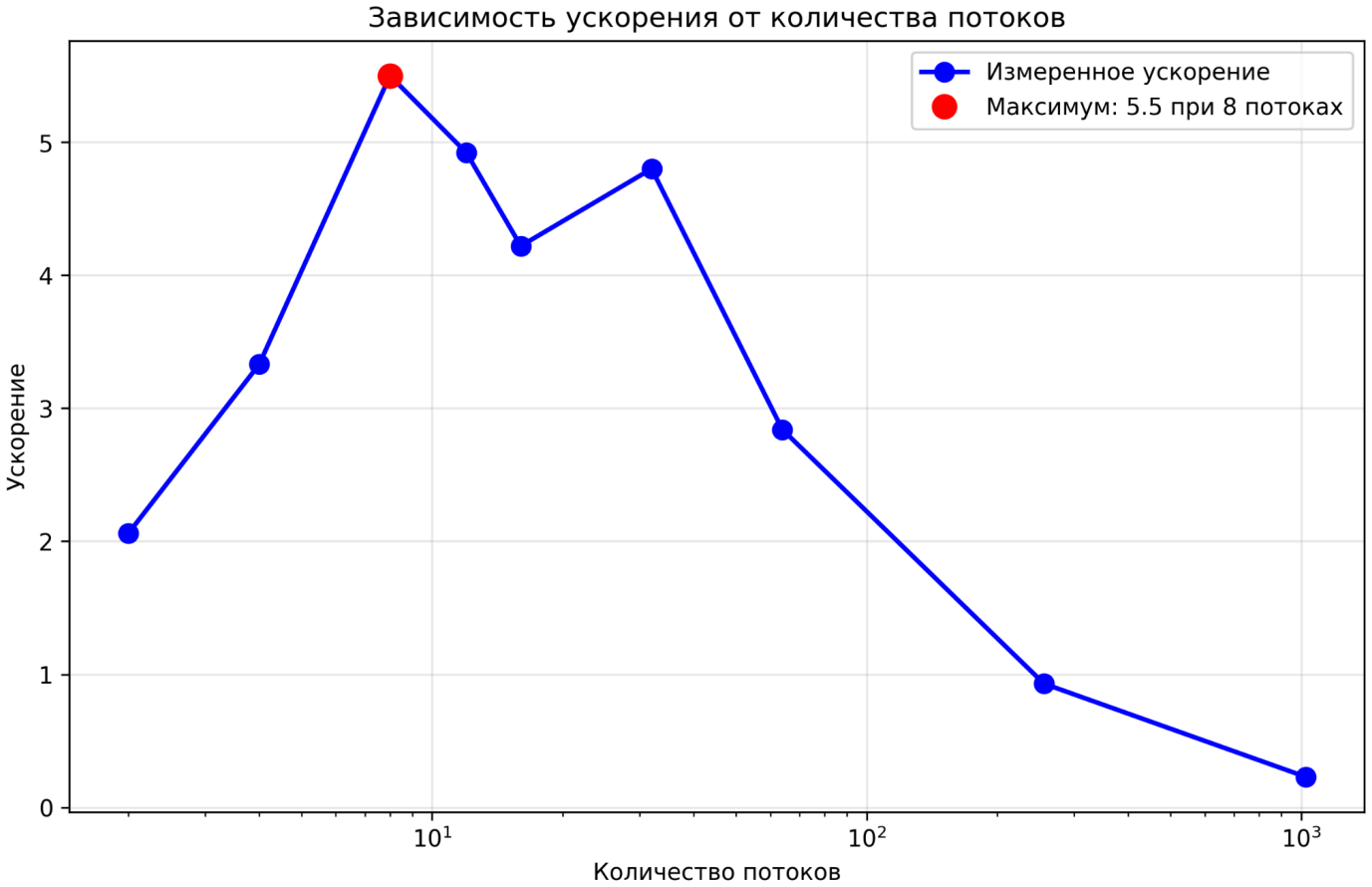
Число потоков	Время исполнения (мс)	Ускорение	Эффективность
2	4.99	2.06	1.03214
4	3.22	3.33	0.83365
8	1.89	5.5	0.68725
12	2.15	4.92	0.41023
16	2.45	4.22	0.26377
32	2.38	4.80	0.15011
64	3.76	2.84	0.04437
256	11.73	0.93	0.00361
1024	48.72	0.23	0.00023

Вывод

В ходе выполнения лабораторной работы была разработана программа с адаптивной стратегией параллельного суммирования массивов. На практике подтвердилось, что использование многопоточности позволяет существенно сократить время вычислений за счет распределения нагрузки между несколькими ядрами процессора. Однако максимальное ускорение достигается только при оптимальном количестве потоков, соответствующем числу физических ядер процессора. При превышении этого количества наблюдается снижение эффективности из-за возрастания накладных расходов на переключение контекста и синхронизацию потоков. Разработанная адаптивная стратегия автоматически выбирает оптимальный режим работы, используя последовательные вычисления для малых объемов данных и параллельные - для больших, что обеспечивает баланс между производительностью и эффективным использованием ресурсов системы.

Ниже приведены графики зависимостей эффективности и ускорения от количества потоков.


Зависимость эффективности от количества потоков
Эффективность
1.0
0.8
0.6
0.4
0.2
0.0
10^1
10^2
10^3
Количество потоков


Зависимость ускорения от количества потоков
Измеренное ускорение
Максимум: 5.5 при 8 потоках
Ускорение
5
4
3
2
1
0
10^1
10^2
10^3
Количество потоков