МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное учреждение высшего образования
«Тверской государственный университет»
(ТВГУ)
Кафедра «Компьютерной безопасности и математических методов управления»

Лабораторная работа №2

по дисциплине «Параллельное программирование»
Тема: «Определение количества потоков CPU»

Выполнил студент группы
М-45
Талачева Р. Р.

Проверил
Желтов С. А.

Тверь, 2024

Оглавление

1 Цель работы

Целью лабораторной работы является определить оптимальное количество потоков CPU в вычислительной системе при организации параллельных вычислений.

2 Задачи

1. Создать многопоточное приложение C.
2. Реализовать последовательный алгоритм вычисления суммы элементов массива целых чисел.
3. Описать параллельный алгоритм, построить граф зависимостей.
4. Реализовать параллельный пирамидальный алгоритм вычисления суммы элементов массива целых чисел.
5. Определить время фактического выполнения программы для разных наборов входных данных и различного количества потоков:

   1 поток 100 000 элементов;

   2 потока 2 00 000 элементов;

   3 потока 2 00 000 элементов;

   И т.д.
6. Провести анализ зависимости времени фактического выполнения программы от количества потоков и размерности массива.

3 Создание приложения

3.1 Операционная система и вычислительная система

Для выполнения расчетов использовалась операционная система **Manjaro Linux** с ядром версии 6.10.11. Основные характеристики вычислительной системы:

- **Архитектура**: x86_64
- **Режим работы процессора**: 32-битный и 64-битный
- **Размер адресации**: 48 бит физический, 48 бит виртуальный
- **Порядок байт**: Little Endian
- **Процессор**: 12 логических ядер (6 физических с поддержкой Hyper-Threading)
  - **Модель процессора**: AMD Ryzen 5 5500U с встроенной графикой Radeon
  - **Частота процессора**:
    - **Максимальная частота**: 4.056 ГГц
    - **Минимальная частота**: 400 МГц
- **L1 кэш**: 192 КБ (данные) и 192 КБ (инструкции) на каждый физический процессор
- **L2 кэш**: 3 МБ
- **L3 кэш**: 8 МБ
- **Виртуализация**: Поддержка AMD-V

Система защищена от известных уязвимостей, таких как **Meltdown**, **Spectre** и других, при помощи аппаратных и программных методов защиты.

3.2 Программные средства

Для выполнения задач использовался компилятор **GCC** с поддержкой **POSIX-потоков** (pthreads). Сборка программы выполняется с использованием **Makefile**. Основные флаги компиляции включают в себя:

- `-Wall` — для включения всех предупреждений
- `-g` — для генерации отладочной информации

Программа компилируется с использованием динамических макросов для управления размером массива, количеством потоков и другими параметрами. Makefile позволяет динамически изменять параметры программы, такие как размер массива, количество потоков и другие, что удобно при проведении экспериментов с разными конфигурациями.

3.3 Подготовка программы

Объявим макросы, которые будем задавать во время компялиции программы:

```
#ifndef OPTION_NUMBER
#define OPTION_NUMBER 1
#endif

#ifndef RAND_RANGE
#define RAND_RANGE 100
#endif

#ifndef ARRAY_SIZE
#define ARRAY_SIZE 100000
#endif

#ifndef THREADS_NUM
#define THREADS_NUM 5
#endif

#ifndef LOOP_NUM
#define LOOP_NUM 100000
#endif
```

Напишем функцию для заполнения массива случайными числами:

```
void random_fill(long long *arr, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        arr[i] = rand() % RAND_RANGE;
    }
}
```

В функции main выделим память для динамического массива, вызовем один раз функцию srand, чтобы массив каждый раз заполнялся разными случайными значениями:

```
 int main() {

    long long *arr = malloc(ARRAY_SIZE * sizeof(long long));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    srand(time(NULL));
    random_fill(arr, ARRAY_SIZE);

    long long sum = 0;
    double elapsed_time = 0.0;

    /*обработка однопоточного или многопоточного режима*/

    printf("Sum of array elements: %lld\n", sum);
    printf("The elapsed time is %f seconds\n", elapsed_time);
    free(arr);
    return 0;
}
```

После выполнения однопоточного или многопоточного алгоритма выводим подчситанную сумму и время выполнения программы.

4 Реализация алгоритма

Описать алгоритм и программную реализацию, которую будете делать.

Программная реализация алгоритма:

- выполняется на любом языке программирования;
- должна иметь интуитивно понятный интерфейс, наглядно представляющий результат;
- входные данные загружаются из файлов, но могут быть отредактированы пользователем;
- результат представляется на экране и сохраняется в файл.

4.1 Последовательный алготитм

Реализуем однопоточный вариант вычисления суммы элементов массива:

```
/*******************************************
*             ОДНОПОТОЧНЫЙ АЛГОРИТМ          *
*******************************************/

long long count_sum(long long *arr, size_t size) {
    long long sum = 0;
    for (size_t i = 0; i < size; ++i) {
        sum += arr[i];
    }
    return sum;
}
```

Время выполнеия будем замерять в отдельной функции.

4.2 Паралелльный алготитм [2]

В моем варианте алгоритма для параллельного подсчёта суммы элементов массива используются потоки, которые обрабатывают разные части массива параллельно.

Структура предназначена для хранения информации, необходимой каждому потоку. Будем переддавать данные потоку в виде указателя на структуру. Здесь происходит преобразование типов, чтобы правильно работать с данными.

Каждому потоку выделяется свой уникальный участок массива для обработки.Каждому потоку присваивается участок массива в зависимости от его thread_id. Начало участка вычисляется как произведение thread_id на размер обрабатываемого блока. Это обеспечивает равномерную нагрузку, особенно когда количество элементов делится на потоки без остатка.

Чтобы избежать переполнения или выхода за пределы массива, последний поток обрабатывает остаточные элементы массива, если их количество меньше, чем стандартный блок.

```
/*******************************************
*             ПАРАЛЛЕЛЬНЫЙ  АЛГОРИТМ          *
*******************************************/
typedef struct {
    long long *arr;
    size_t size;
    long long result;
    int thread_id;
} ThreadData;

void* parallel_count(void *arg) {
   ThreadData *data = (ThreadData*) arg;
   size_t start = data->thread_id * data->size;
   size_t end = (data->thread_id +1) * data->size;
   data->result = 0;

   for(size_t i = start; i < end && i < ARRAY_SIZE; ++i) {
       data->result += data->arr[i];
```

```
    }

    if (data->thread_id == THREADS_NUM - 1) {
        end = ARRAY_SIZE;
    }

    pthread_exit (NULL);
}
```

Использование потоков позволяет разделить обработку большого массива между несколькими ядрами процессора. Это повышает производительность программы за счёт одновременного выполнения задач.

В моей реализации алгоритма зависимости между потоками минимальны, так как каждый поток работает над отдельной частью массива и ни один из них не ждет завершения другого. Каждый поток делает вычисления независимо, поэтому они могут работать одновременно. И только в конце программы в функции pthread_join , где главный поток ждет завершения всех потоков происходит их слияние.

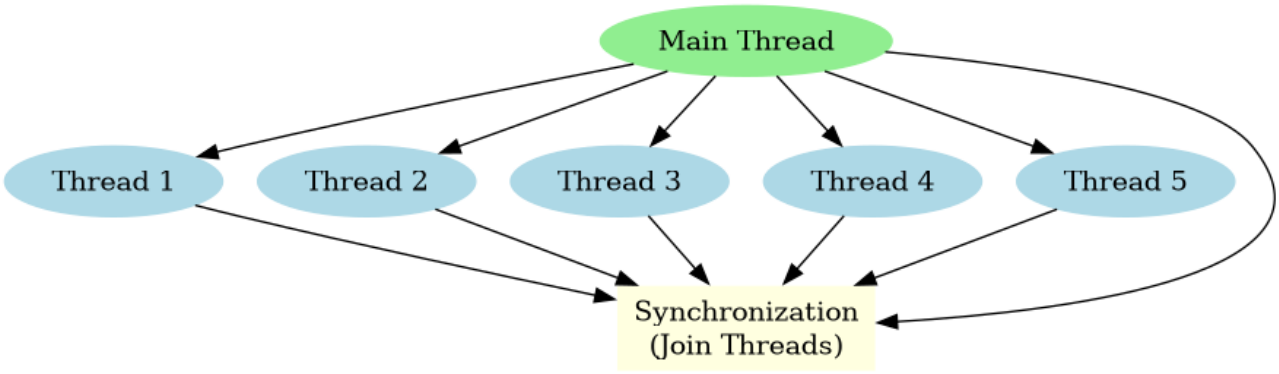

Рис. 4.1: Граф зависимостей

На рисунке [4.1] визуализирован граф зависимостей из пяти потоков при помощи программы Graphviz.

5 Оценка эффективности параллельной программы

Для оценки эфективности будем сравнивать показатели скорости выполнения при ее запуске на различных потоках. Так как измерения проводятся на одной высислительной системе, оптимальным варианом будет управление кольческом потоков выполнения. Анализируя измерения скорости работы для различного числа потоков, можно рассчитать показатели эффективности распараллеливания.
Будем рассматривать замеры времени работы программы с различными значениями параметров:

- **ARRAY_SIZE** — размер массива.
- **THREADS_NUM** — количество потоков.
- **LOOP_NUM** — количество итераций основного цикла (при этом программа вычисляет сумму массива несколько раз).

5.1 Анализ времени в зависимости от количес тва потоков (THREADS_NUM)

Исходные данные для анализа:

Таблица 5.1: Результаты измерений

THREADS_NUM	ARRAY_SIZE	Elapsed time (s)	LOOP_NUM
1	100000	0.00017	100000
1	200000	0.000235	100000
1	400000	0.000463	100000
1	600000	0.000725	100000
1	800000	0.001048	100000
2	100000	0.000036	100000
2	200000	0.000047	100000

Таблица 5.1: Результаты измерений

THREADS_NUM	ARRAY_SIZE	Elapsed time (s)	LOOP_NUM
2	400000	0.000352	100000
2	600000	0.000790	100000
2	800000	0.005380	100000
4	100000	0.000093	100000
4	200000	0.000205	100000
4	400000	0.000660	100000
4	600000	0.001462	100000
4	800000	0.002812	100000
6	100000	0.000132	100000
6	200000	0.000260	100000
6	400000	0.000382	100000
6	600000	0.001482	100000
6	800000	0.002608	100000
8	100000	0.000312	100000
8	200000	0.0003569	100000
8	400000	0.000523	100000
8	600000	0.002501	100000
8	800000	0.002753	100000

Для одного потока [5.1] наблюдается линейный рост времени выполнения по мере увеличения размера массива. Расходы на обработку данных ожидаемо увеличиваются с увеличением размера массива, что характерно для последовательной обработки данных. Использование одного потока может быть оправдано для небольших массивов (например, до 200000 элементов), поскольку прирост времени незначителен, но для больших массивов становится менее оптимально.

Для 2 потоков [5.1] время выполнения заметно снижается на малых размерах массива по сравнению с 1 потоком, достигая значительного ускорения. Время резко возрастает при размере массива 800000 элементов, что может свидетельствовать о повышении накладных расходов на распределение задач между потоками. На небольших массивах (100000-200000 элементов) эффек-

тивность 2 потоков является оптимальной, но с увеличением размера массива эффективность падает.

Для 4 потоков [5.1] эффективность возрастает при размерах массива до 400000 элементов, где достигается более низкое время выполнения по сравнению с 1 и 2 потоками. Начиная с 600000 элементов, время выполнения также увеличивается, но не столь критично, как для 2 потоков. Можно предположить, что 4 потока – хороший баланс для умеренно больших массивов (от 200000 до 600000 элементов), но для предельно больших массивов (800000 элементов) накладные расходы опять начинают влиять на производительность.

При использовании 6 [5.1] потоков наблюдается увеличение времени для небольших массивов (100000-200000 элементов) по сравнению с 4 потоками. Это указывает на накладные расходы, которые становятся ощутимыми при малых объемах данных. Для массивов размером 400000-800000 элементов производительность остается стабильной и сравнительно эффективной. 6 потоков оптимально подходят для массивов среднего размера (400000 элементов), но увеличиваются накладные расходы для малых массивов, что снижает эффективность.

Время выполнения значительно возрастает для всех размеров массива, особенно для небольших массивов, что указывает на значительные накладные расходы при увеличении количества потоков ([5.1] и [5.2]). 8 потоков не являются оптимальным вариантом для данного алгоритма, так как не дают значительного прироста производительности. Накладные расходы на управление потоками перевешивают преимущество параллелизации при 8 потоках.

5.2 Анализ параллельного ускорения [1]

Под параллельным ускорением будем понимать величину, отражающую прирост скорости выполнения параллельной программы на заданном числе потоков по сравнению с однопоточным режимом, т.е.

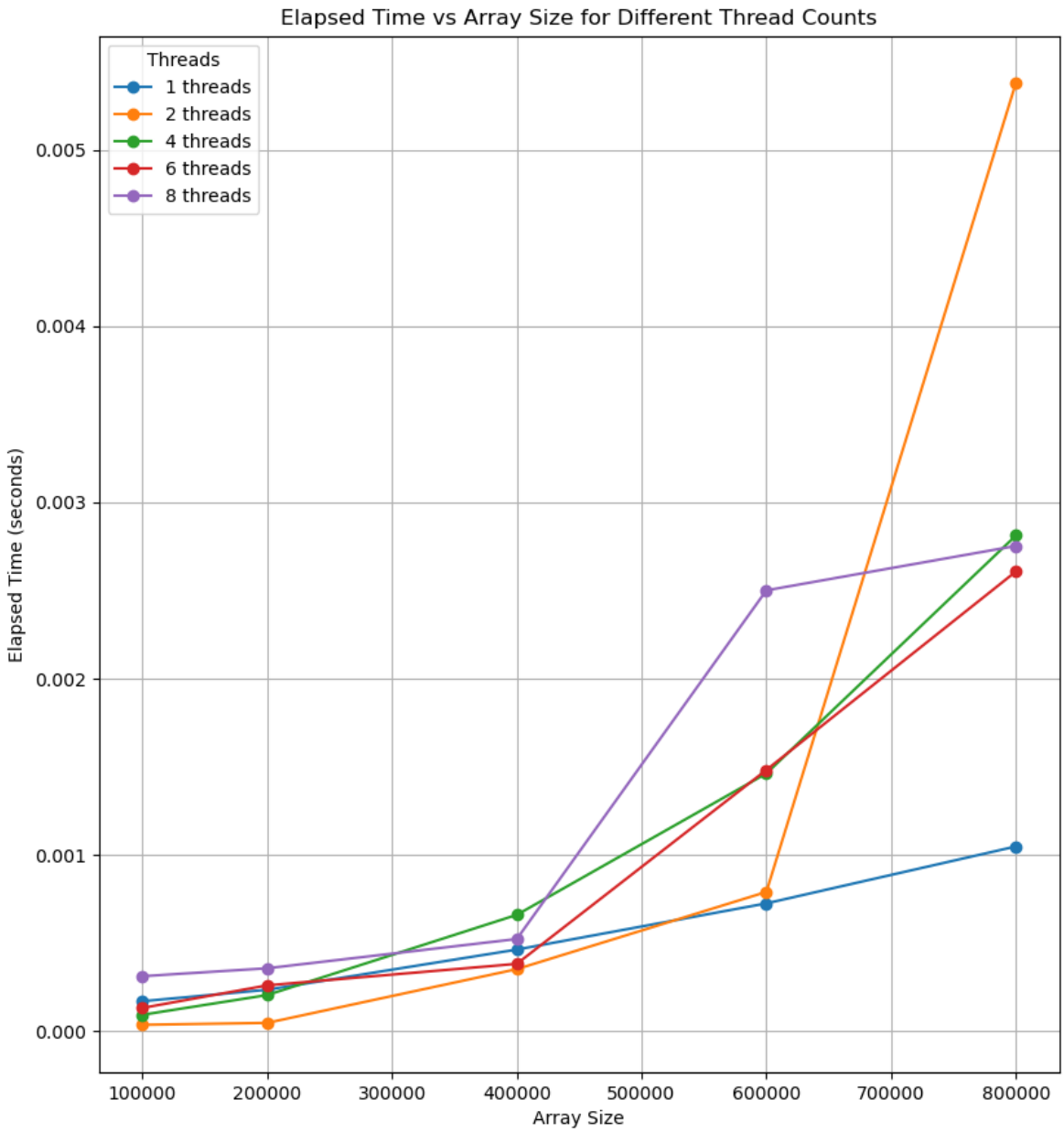

Рис. 5.1: График зависимости скорости выполнения от числа потоков

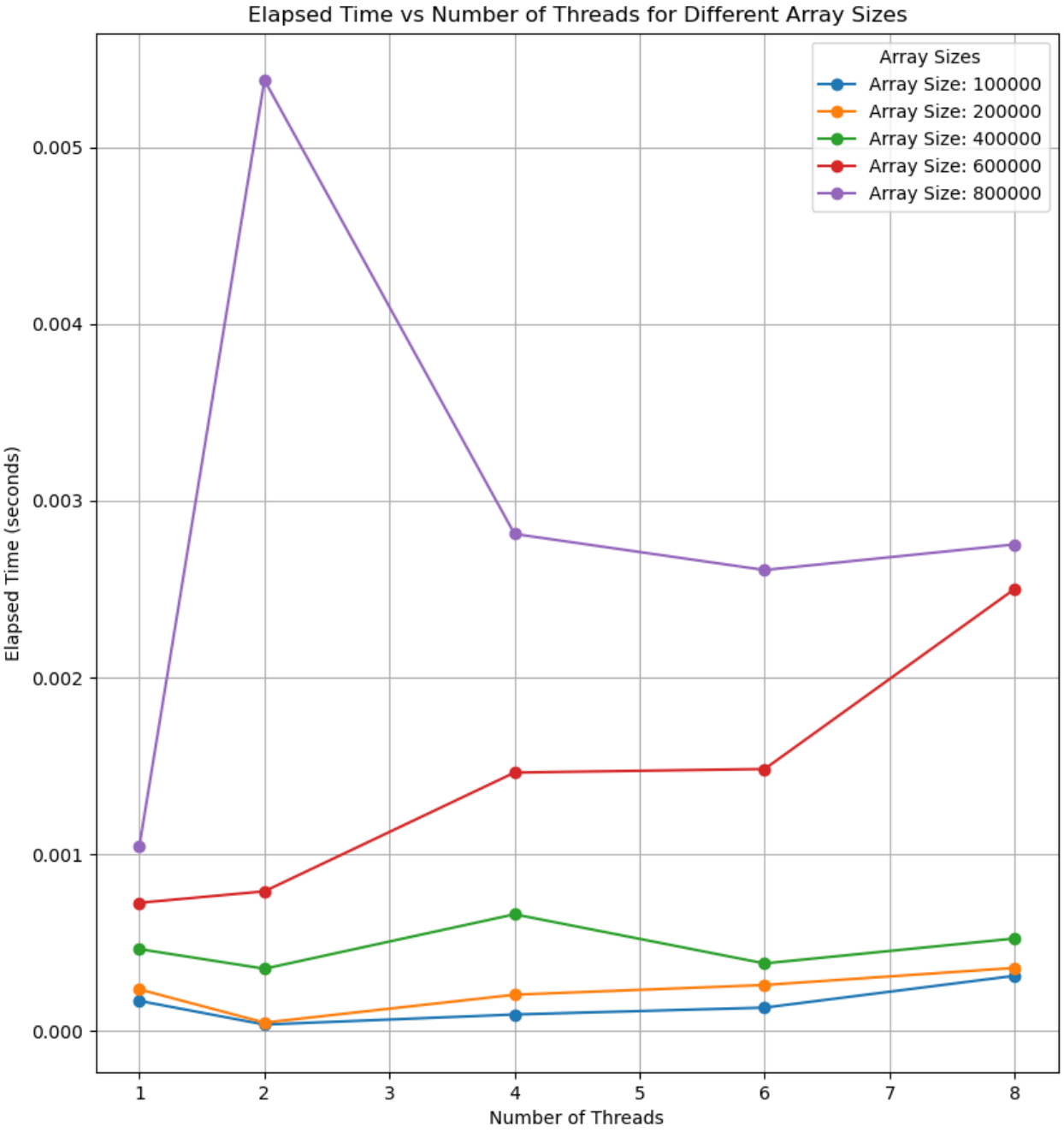

Рис. 5.2: График зависимости скорости выполнения от размера массива

$$S(p) = \frac{V(p)}{V(1)} \tag{5.1}$$

где $V(p)$ - средняя скорость выполнения программы, на p потоках, выраженная в условных еденицах работы в секунду УЕР/с. В данном случает УЕР - количество просуммированных элементов массива.

Таблица 5.2: Расчет ускорения для всех потоков

0	100,000	200,000	400,000	600,000	800,000
2	0.2118	0.2	0.7603	1.0897	5.1336
4	0.5471	0.8723	1.4255	2.0166	2.6832
6	0.7765	1.1064	0.8251	2.0441	2.4885
8	1.8353	1.5187	1.1296	3.4497	2.6269

Заметим, что ускорение для двух потоков на 800000 элементах ([5.2]) превышает значение p, что теоретически неверно. Однако сверх-линейное параллельное ускорение допустимо в экспериментах, так как такие ситуации можно списать на увеличение вычислительного ресурса процессоров и объема кеш-памяти первого уровня. И иногда от этого повышается процент кеш-попаданий.

5.3 Анализ параллельной эффективности [1]

Формула для расчета параллельной эффективности

$$E(p) = \frac{S(p)}{p} = \frac{V(p)}{p \cdot V(1)} \tag{5.2}$$

Будем вычислять среднюю скорость выполнения программы **методом Амдала** - рассчитать $V(p)$, зафиксировав объем выполняемой работы, когда для разлычных p время разлячное.

В таком случае выражение для параллельного ускорения примет вид:

$$S(p) = \frac{V(p)}{V(1)} = \frac{t(1)}{t(p)} \tag{5.3}$$

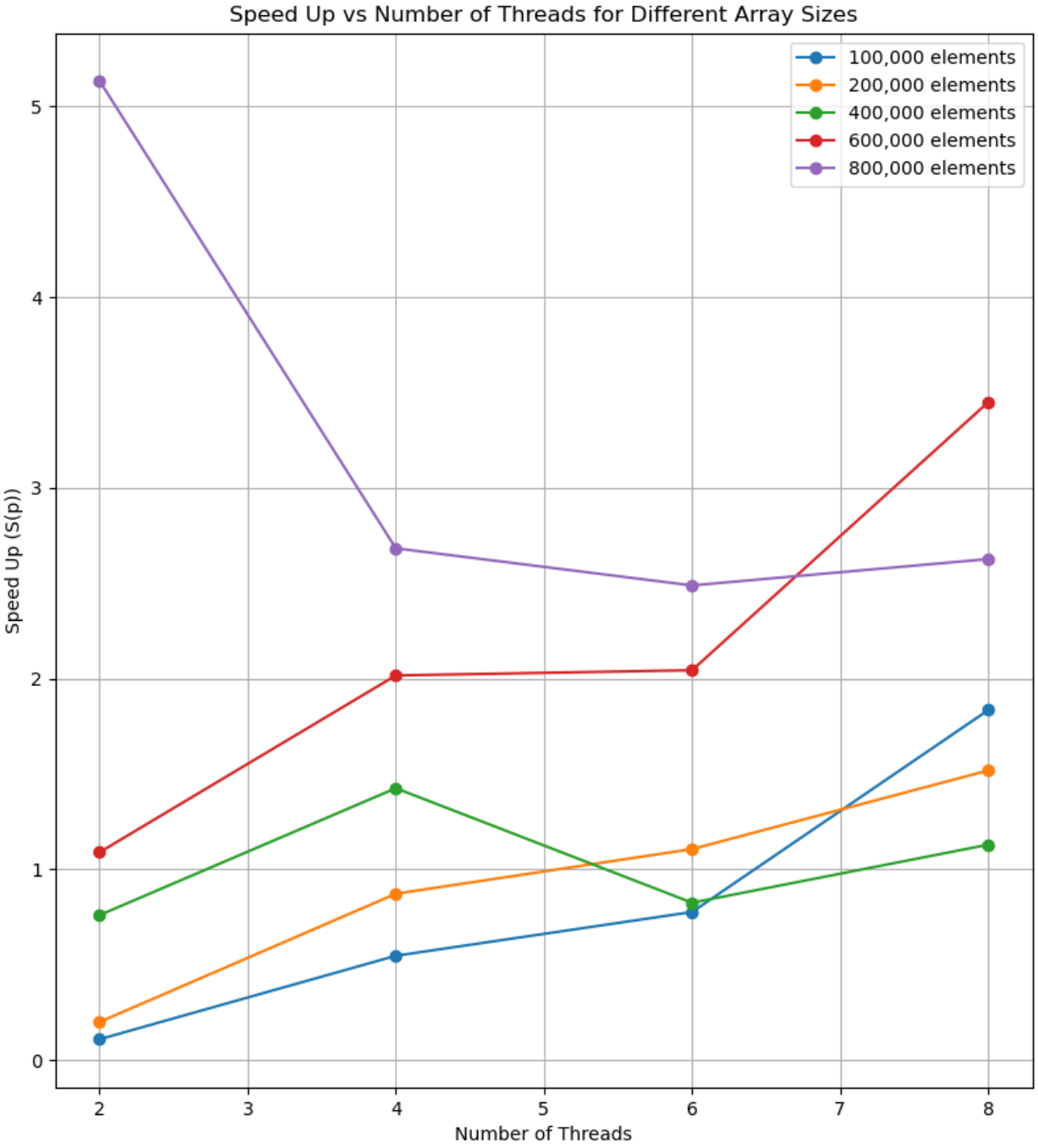

Рис. 5.3: График зависомости парралельного ускорения от числа потоков

Модифицировав этот закон по Бухановскому, имеем:

$$S(p) = \frac{T_1}{T_p} \tag{5.4}$$

, где T_1 — время выполнения на 1 потоке, а T_p — время выполнения на pp потоках.

Таблица 5.3: Расчет параллельной эффективности для всех потоков

0	100,000	200,000	400,000	600,000	800,000
2	0.1059	0.1	0.3802	0.5449	2.5668
4	0.1368	0.2181	0.3564	0.5042	0.6708
6	0.1294	0.1844	0.1375	0.3407	0.4148
8	0.2294	0.1898	0.1412	0.4312	0.3284

Таким образом, чем более ближе значения ускорения к p, тем более эффективен алгоритм, то есть эффективность $\to 1$. (См. [5.3])

5.4 Оптимальное количество потоков

Анализ таблицы эффективности:

- **Для массива в 100,000 элементов:** эффективность заметно снижается при увеличении количества потоков, и наибольшая эффективность достигается при использовании 2 потоков.
- **Для 200,000 и 400,000 элементов:** эффективность лучше всего на 2 и 4 потоках, но начинает падать при увеличении потоков.
- **Для 600,000 элементов:** эффективность также выше на 2 и 4 потоках.
- **Для 800,000 элементов:** эффективность выше на 2 потоках, но всё ещё приемлема на 4 потоках, затем начинает снижаться.

5.5 Выводы

Оптимальное количество потоков для каждого размера массива варьируется в зависимости от требуемой производительности и эффективности. На основе таблицы можно сделать следующие рекомендации:

- **100,000 - 400,000 элементов:** 2-4 потока.
- **600,000 - 800,000 элементов:** 2-4 потока также остаются оптимальными, но можно использовать до 6 потоков, если требуется немного больше производительности, хотя это снижает эффективность.

6 Приложение

6.1 Исходный под программы

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

#ifndef OPTION_NUMBER
#define OPTION_NUMBER 1
#endif

#ifndef RAND_RANGE
#define RAND_RANGE 100
#endif

#ifndef ARRAY_SIZE
#define ARRAY_SIZE 100000
#endif

#ifndef THREADS_NUM
#define THREADS_NUM 5
#endif

#ifndef LOOP_NUM
#define LOOP_NUM 100000
#endif
```

```
double measure_time(long long (*func)(long long*, size_t),
                    long long *arr, size_t size, long long *result) {
    clock_t begin = clock();
    *result = func(arr, size);
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    return time_spent;
}

void random_fill(long long *arr, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        arr[i] = rand() % RAND_RANGE;
    }
}

/*********************************************
 *            ОДНОПОТОЧНЫЙ АЛГОРИТМ          *
 *********************************************/

long long count_sum(long long *arr, size_t size) {
    long long sum = 0;
    for (size_t i = 0; i < size; ++i) {
        sum += arr[i];
    }
    return sum;
}

/*********************************************
 *            ПАРАЛЛЕЛЬНЫЙ  АЛГОРИТМ         *
 *********************************************/
typedef struct {
    long long *arr;
    size_t size;
    long long result;
    int thread_id;
```

```
} ThreadData;

void* parallel_count(void *arg) {
    ThreadData *data = (ThreadData*) arg;
    size_t start = data->thread_id * data->size;
    size_t end = (data->thread_id +1) * data->size;
    data->result = 0;

    for(size_t i = start; i < end && i < ARRAY_SIZE; ++i) {
        data->result += data->arr[i];
    }

    if (data->thread_id == THREADS_NUM - 1) {
        end = ARRAY_SIZE;
    }

    pthread_exit (NULL);
}




int main() {

    long long *arr = malloc(ARRAY_SIZE * sizeof(long long));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    srand(time(NULL));
    random_fill(arr, ARRAY_SIZE);

    long long sum = 0;
```

```
    double elapsed_time = 0.0;

   /* for (size_t i = 0; i < ARRAY_SIZE; ++i) {
        printf("arr[%ld] = %lld\n", i, arr[i]);
    }*/


#if OPTION_NUMBER == 1

    elapsed_time = measure_time(count_sum, arr, ARRAY_SIZE, &sum);

#elif OPTION_NUMBER == 2
    clock_t begin = clock();

    for (size_t i = 0; i < LOOP_NUM; ++i) {

       ThreadData thread_data[THREADS_NUM];
       pthread_t threadID[THREADS_NUM];
       size_t partition_size = ARRAY_SIZE / THREADS_NUM;

       for (int i = 0 ; i < THREADS_NUM ; i++) {
           thread_data[i].arr = arr;
           thread_data[i].size = partition_size;
           thread_data[i].thread_id = i;
           pthread_create (&threadID[i], NULL,parallel_count ,
                           (void*)&thread_data[i]);
       }

       for (int i = 0 ; i < THREADS_NUM ; i++) {
          pthread_join (threadID[i], NULL);
          sum += thread_data[i].result;
       }
    }

    clock_t end = clock();
```

```
    elapsed_time = (double)(end - begin) / CLOCKS_PER_SEC;

#endif

    printf("Sum of array elements: %lld\n", sum);
    printf("The elapsed time is %f seconds\n", elapsed_time);
    free(arr);
    return 0;
}
```

6.2 Makefile

```
CC = gcc

CFLAGS = -Wall -g

SRC = src/main.c
TARGET = array_sum

OPTION_NUMBER ?= 1
RAND_RANGE ?= 100
ARRAY_SIZE ?= 100000
THREADS_NUM ?= 5
LOOP_NUM ?= 100000


all:
        $(CC) $(CFLAGS) -DOPTION_NUMBER=$(OPTION_NUMBER)
        -DRAND_RANGE=$(RAND_RANGE) \
        -DARRAY_SIZE=$(ARRAY_SIZE) -DTHREADS_NUM=$(THREADS_NUM)
        -DLOOP_NUM=$(LOOP_NUM) \
        -o $(TARGET) $(SRC)

clean:
```

```
	rm -f $(TARGET)
```

Литература

[1] В.В. Соснин, П.В. Балакшин, Д.С. Шилко, Д.А. Пушкарев, А.В. Мишенёв,П.В. Кустарев, А.А. Тропченко, ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ, Санкт-Петербург, 2023.

[2] Flynn, M. J. (1972). "Some computer organizations and their effectiveness."IEEE Transactions on Computers.