ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ "МИФИ"»

Институт Интеллектуальных Кибернетических Систем
Кафедра №42 "Криптология и кибербезопасность"
Дисциплина «Параллельное программирование»

Отчет к лабораторной работе № 2
«Выделение ресурса параллелизма. Технология OpenMP»

Выполнили:
студенты группы Б22-505
Титов Дмитрий Иванович

Принял:
Куприяшин Михаил Андреевич

Москва
2024 год

Цель работы

Приобрести навыки разработки параллельной программы путём обнаружения ресурса параллелизма в имеющейся последовательной реализации.

1. Рабочая среда

Процессор - amd Ryzen 7 7840HS (2023) 8 ядер и 16 потоков
Оперативная память - 16GB DDR5
ОС - Ubuntu 24.04.1 LTS 64-бит
Среда разработки - VS Code, CMake 3.28.3, GCC 13.2.0
Версия OpenMP - 4.5 201511
Ноутбук был на зарядке

2. Оценка временной сложности алгоритма

В данном алгоритме происходит последовательный поиск элемента в массиве. Оценим его сложность:

Лучший случай (Best Case):

- Лучший случай происходит, если искомый элемент target находится в первом элементе массива, то есть на позиции array[0].
- Время выполнения составит O(1), так как алгоритм завершит работу после первой же итерации.

Худший случай (Worst Case):

- Худший случай — когда элемент target отсутствует в массиве или находится в последнем элементе массива.
- В этом случае алгоритм должен пройти по всему массиву, что означает count итераций.
- Время выполнения составит O(n), где n — это количество элементов в массиве (в данном случае count = 10,000,000).

Средний случай (Average Case):

- В среднем случае, элемент будет найден в среднем на позиции, которая находится в середине массива. То есть, алгоритм выполнит примерно n/2 операций.
- Ожидаемая временная сложность будет составлять O(n) в среднем случае, то есть относится к линейному классу временной сложности. Однако важно понимать, что различия в числе n / 2 и n могут существенно сказаться на времени работы программы, то есть может доходить до различий в разы.

Вывод:

- В лучшем случае — O(1)
- В худшем и среднем случае — O(n)

Расчет среднего случая

Средний случай означает, что элемент target может быть найден в среднем на любой позиции массива. Таким образом, можно рассматривать среднее количество операций, которые необходимо выполнить для нахождения элемента.

Важные предположения:

1. Элемент target есть в массиве.
2. Элемент может быть найден на любой позиции с вероятностью **1/n**.
3. Ожидаемая позиция нахождения элемента — это среднее значение позиций всех возможных мест, где элемент может быть найден. Это предположение следует из того, что среднее значение позиции вычисляется как мат. ожидание, то есть как сумма произведений позиции на ее вероятность ((1 / n) * i), дальнейшие расчеты будут схожи с расчетом числа операций, то есть приведут к значению (n + 1) / 2

Расчет среднего количества операций:

Пусть **i** — это индекс, на котором будет найден элемент. Если элемент найден на позиции **i**, то алгоритм выполнит **i** сравнений. Позиция **i** может быть любой от **1** до **n** с вероятностью **1/n**.

Ожидаемое количество операций (сравнений) **E(n)** для нахождения элемента в среднем можно найти как математическое ожидание:

$$\mathbf{E(n) = (1/n) \cdot (1+2+3+\cdots+n)}$$

Сумма $\mathbf{1+2+3+\cdots+n}$— это сумма первых **n** натуральных чисел, которая равна:

$$\mathbf{(n(n+1))/2.}$$

Тогда среднее количество операций **E(n)** будет равно:

$$\mathbf{E(n) = (1/n) \cdot (n(n+1))/2 = (n+1)/2}$$

Таким образом, среднее количество операций для нахождения элемента в массиве из **n** элементов равно **(n+1)/2**, что в asymptotic (асимптотическом) выражении даёт **O(n)**.

Итог:

- В среднем случае количество операций для нахождения элемента в массиве из **n** элементов — это **O(n)**.
- Это означает, что алгоритм требует линейное количество сравнений в среднем, даже если элемент присутствует в массиве.

Дополнительно

Если элемент отсутствует в массиве, алгоритм выполняет nnn операций. Это также соответствует случаю **O(n)**. Однако для анализа среднего случая чаще всего предполагается, что элемент присутствует в массиве, поскольку вероятность его отсутствия зависит от случайности.

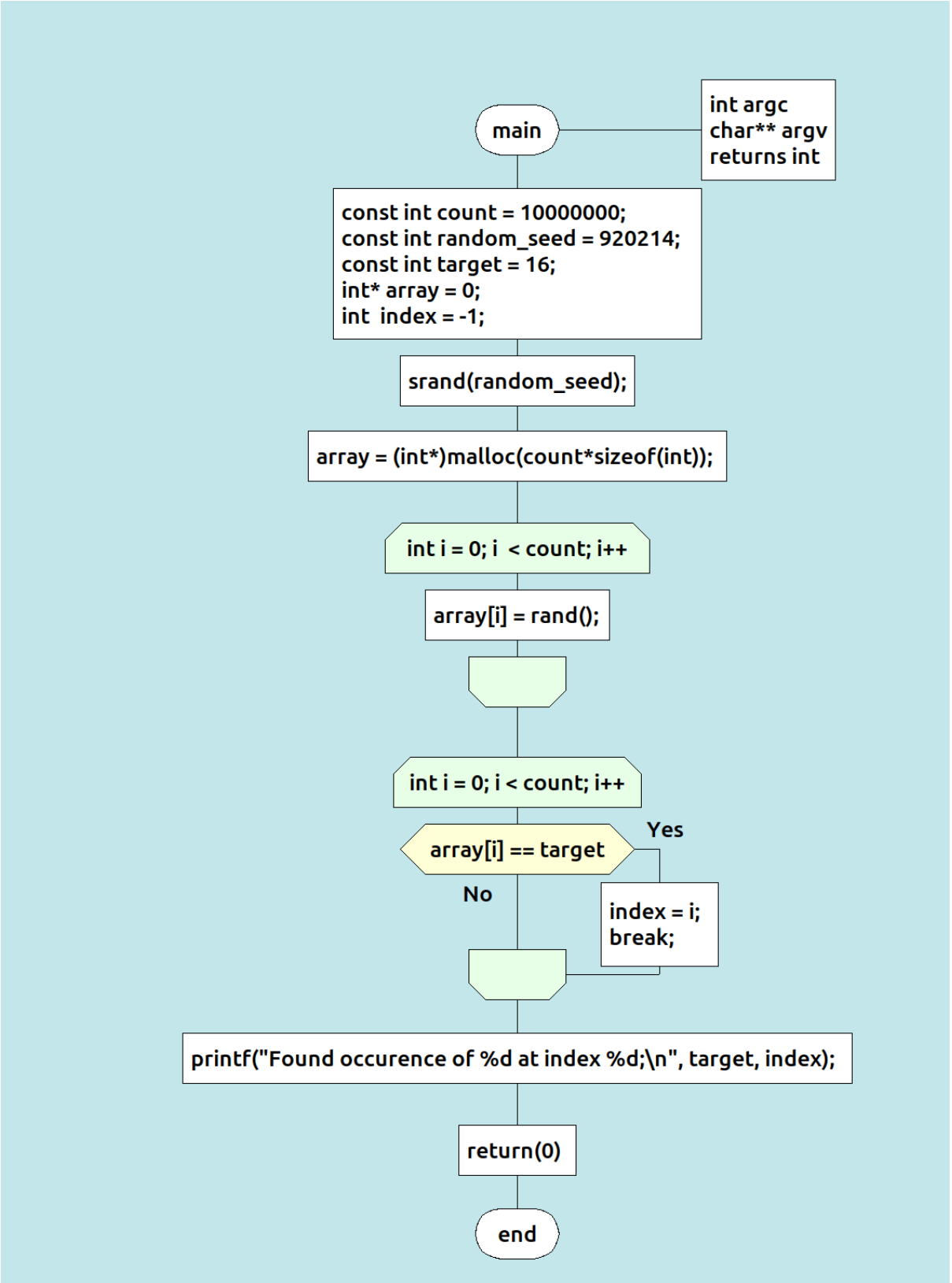

Рис. 2.1. ДРАКОН-схема предложенного алгоритма

3. Параллельный алгоритм

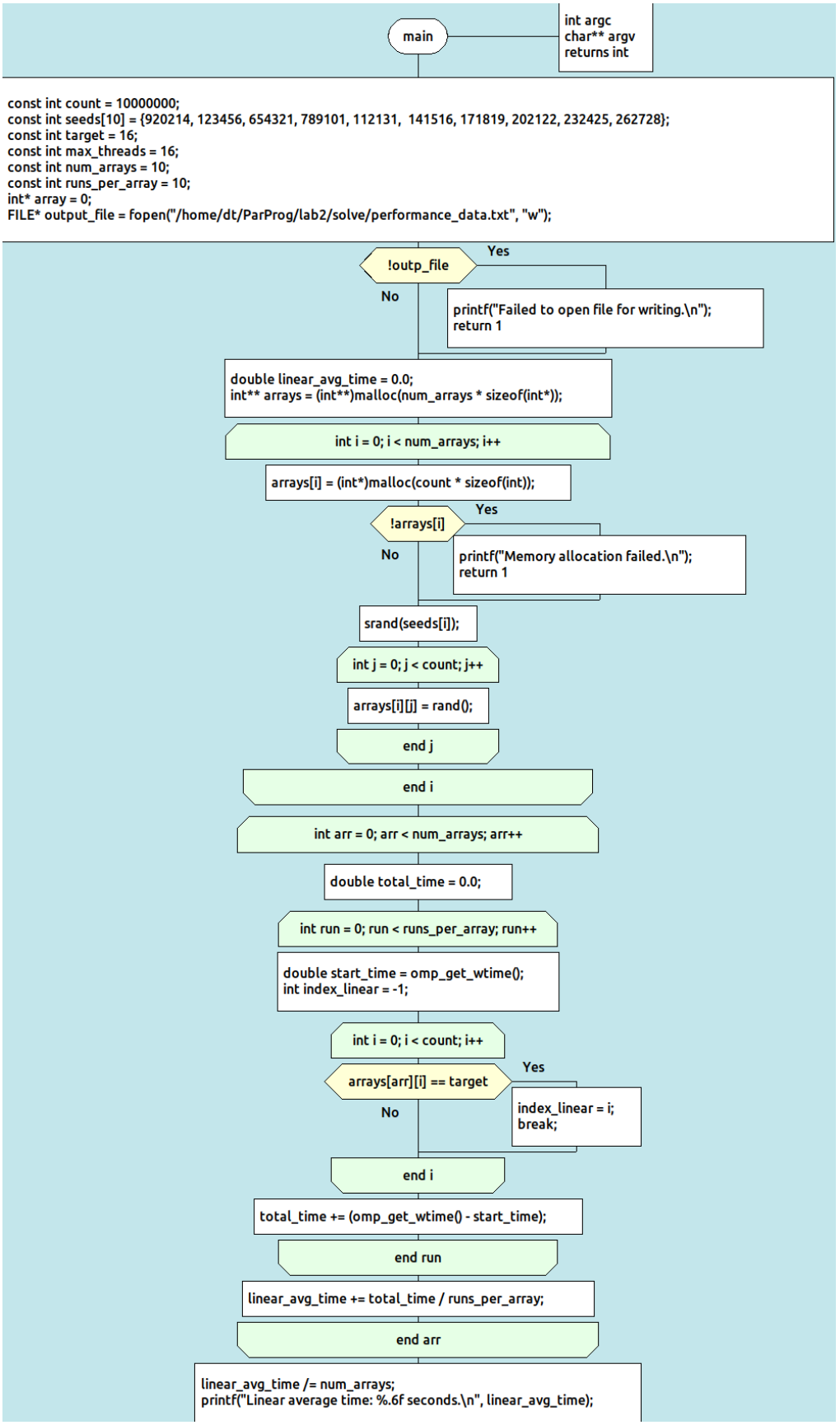

Рис. 3.1. ДРАКОН-схема параллельной программы, ввод данных, последоват. проход

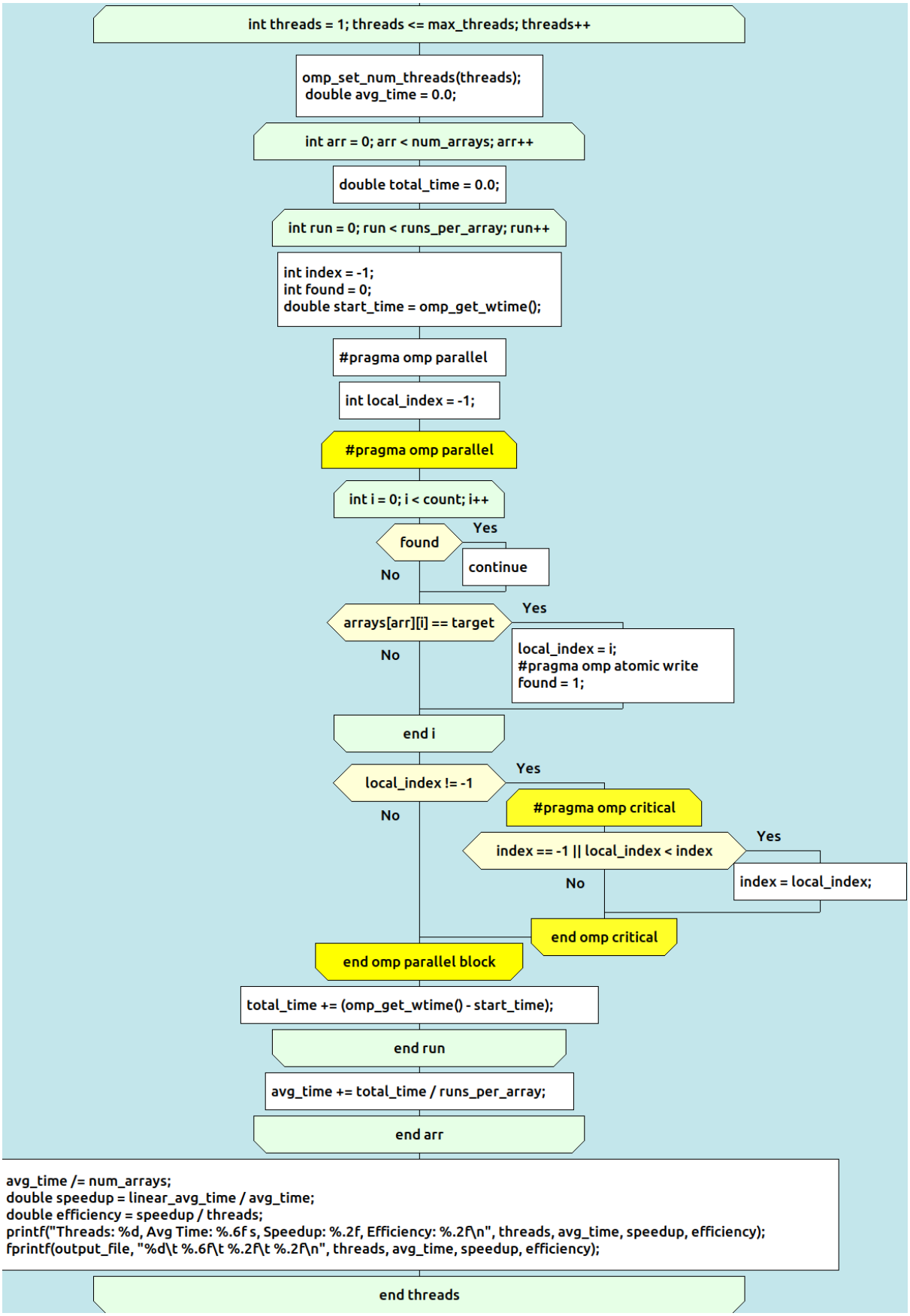

Рис. 3.2. ДРАКОН-схема параллельной программы, сам параллельный алгоритм

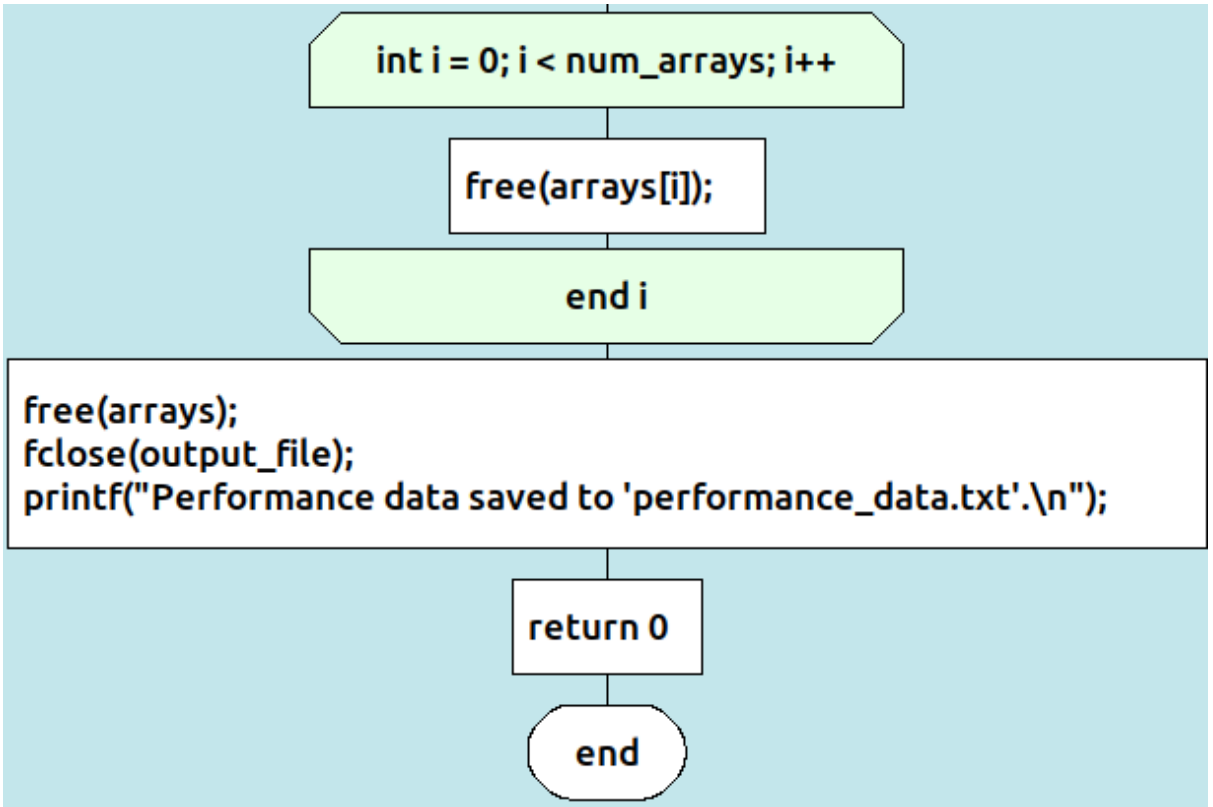

Рис. 3.3. ДРАКОН-схема параллельной программы. окончание

В данной программе можно распараллелить поиск, однако потребуется прерывание алгоритма в тот момент, когда будет найден нужный элемент. Это можно сделать с помощью директивы omp cancel, но вместо этого можно omp atomic и omp critical, то есть вместо отмены потоков каждый поток может проверять глобальный индикатор успешного поиска. Если число найдено, остальные потоки автоматически пропускают оставшиеся итерации. Такой подход минимизирует издержки для отмены потоков через omp cancel, так как заменяется крайне простой проверкой флагов, к тому же программе не предоставляется возможности завершить потоки в неопределенном состоянии.

Директивы OpenMP

#pragma omp parallel

Эта директива указывает, что следующий блок кода должен выполняться параллельно с использованием нескольких потоков. Она используется, чтобы указать, что код внутри блока должен быть выполнен разными потоками параллельно. В данном случае она применяется в коде поиска элемента в массиве, позволяя распараллелить цикл поиска.

#pragma omp for

Эта директива используется внутри параллельного блока для разделения итераций цикла между потоками. Каждый поток получает часть работы, которая будет выполнена в рамках данного цикла. В коде она используется для параллельного выполнения цикла, который ищет целевой элемент в массиве.

#pragma omp atomic write

Эта директива используется для атомарной операции, обеспечивающей, что запись переменной found будет выполнена атомарно, то есть без прерываний. Это предотвращает состояния гонки, когда несколько потоков одновременно пытаются изменить значение переменной.

#pragma omp critical

Директива #pragma omp critical указывает, что следующий блок кода должен быть выполнен исключительно одним потоком одновременно. В данном случае она используется для обновления переменной index, чтобы избежать гонки за доступ к этой переменной несколькими потоками одновременно.

4. Анализ

4.1 Среднее время от числа потоков

Время выполнения программы в зависимости от числа потоков можно оценить по изменению среднего времени с увеличением числа потоков.

Формула:

$$T_{avg}(threads) = \frac{Total\ Time}{Number\ of\ Arrays \times Runs\ per\ Array}$$

Где:

- T_{avg}(threads) — среднее время для заданного числа потоков.

Видно, что с увеличением числа потоков среднее время для выполнения программы значительно уменьшается, что подтверждается следующими наблюдениями:

- При одном потоке T_{avg}=0.010343 с ростом числа потоков среднее время резко падает до 0.001426 при 16 потоках.
- С увеличением числа потоков время сначала уменьшается, но на 8 потоке оно начинает немного увеличиваться. Это может быть связано с наложением накладных расходов на многозадачность или неэффективным использованием потоков на определенных уровнях.

Причины уменьшения времени:

- При увеличении числа потоков система может эффективно разделять задачи между ядрами, что ускоряет выполнение.
- На 8 и более потоках происходит небольшое увеличение времени, что может быть связано с:
    - Перегрузкой системы, когда количество потоков превышает количество ядер.
    - Потерей производительности из-за накладных расходов на синхронизацию и управление потоками.

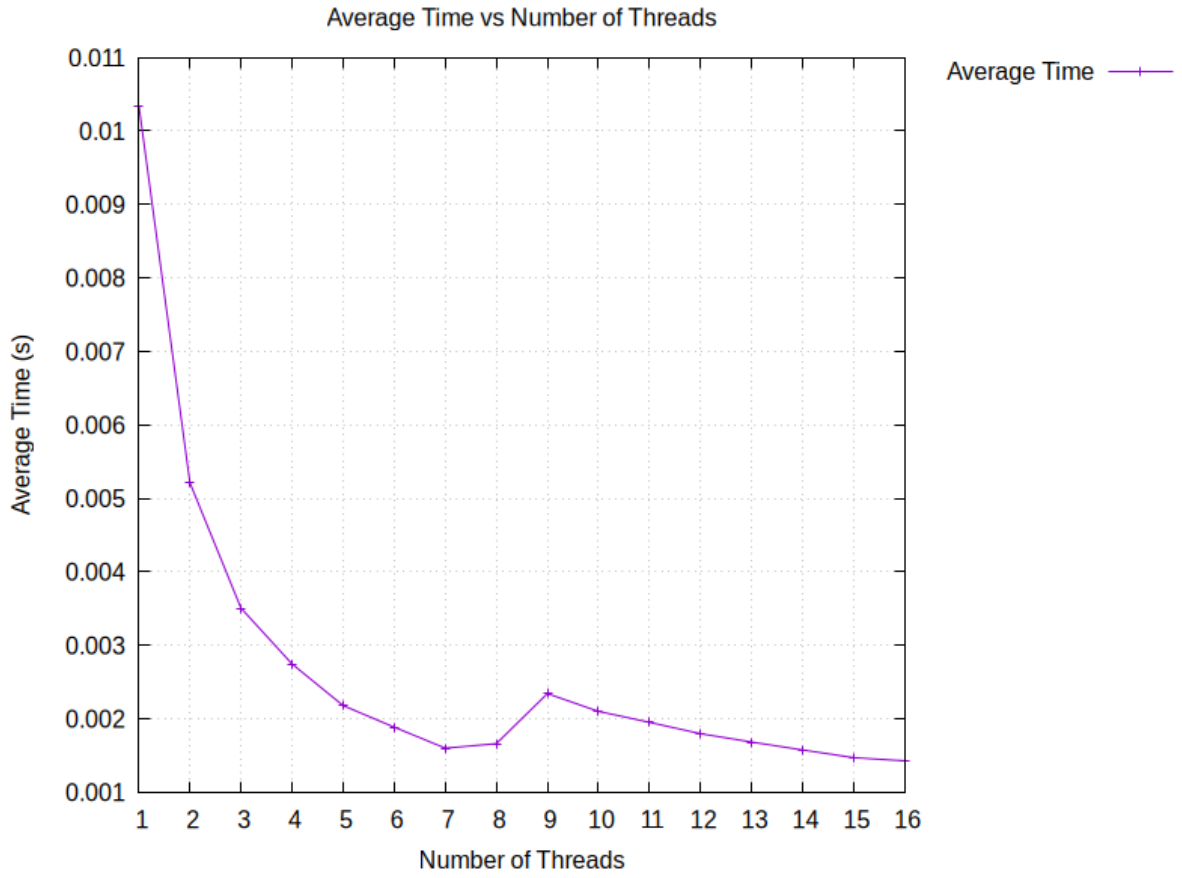

Рис. 4.1. График зависимости среднего времени от числа потоков

4.2 Ускорение от числа потоков

$$S(threads) = \frac{T_{linear}}{T_{avg}(threads)}$$

Где:

- T_{linear} — время для линейного поиска (с использованием 1 потока).
- T_{avg}(threads) — среднее время при использовании nnn потоков.

Ускорение показывает, насколько быстрее выполняется программа с увеличением числа потоков по сравнению с линейным методом.

Анализ:

- Для одного потока видим, что ускорение не равно 1, что говорит нам, о уже существующих расходах, хотя программа не начала еще синхронизации потоков.
- С увеличением числа потоков ускорение растет, достигая значений, близких к 6.18 при 16 потоках.
- Ускорение растет до 7 потоков, после чего проваливается на 8 и 9 потоках, после чего заново наблюдается рост

Вывод по ускорению:

- Ускорение возрастает с количеством потоков до определенного предела.
- После 7 потоков ускорение сначала чутка спадает, после чего проваливается на 9 потоков и затем заново растет до 16 потоков. Данное явление также можно объяснить тем, что система достигает физического количества ядер и на 8 и 7 потоках, должна быть более или менее сохраняющая тенденцию роста ситуация, однако наблюдается аномалия, скорее всего связанная с тем, что система слишком близко подошла к физической границе.

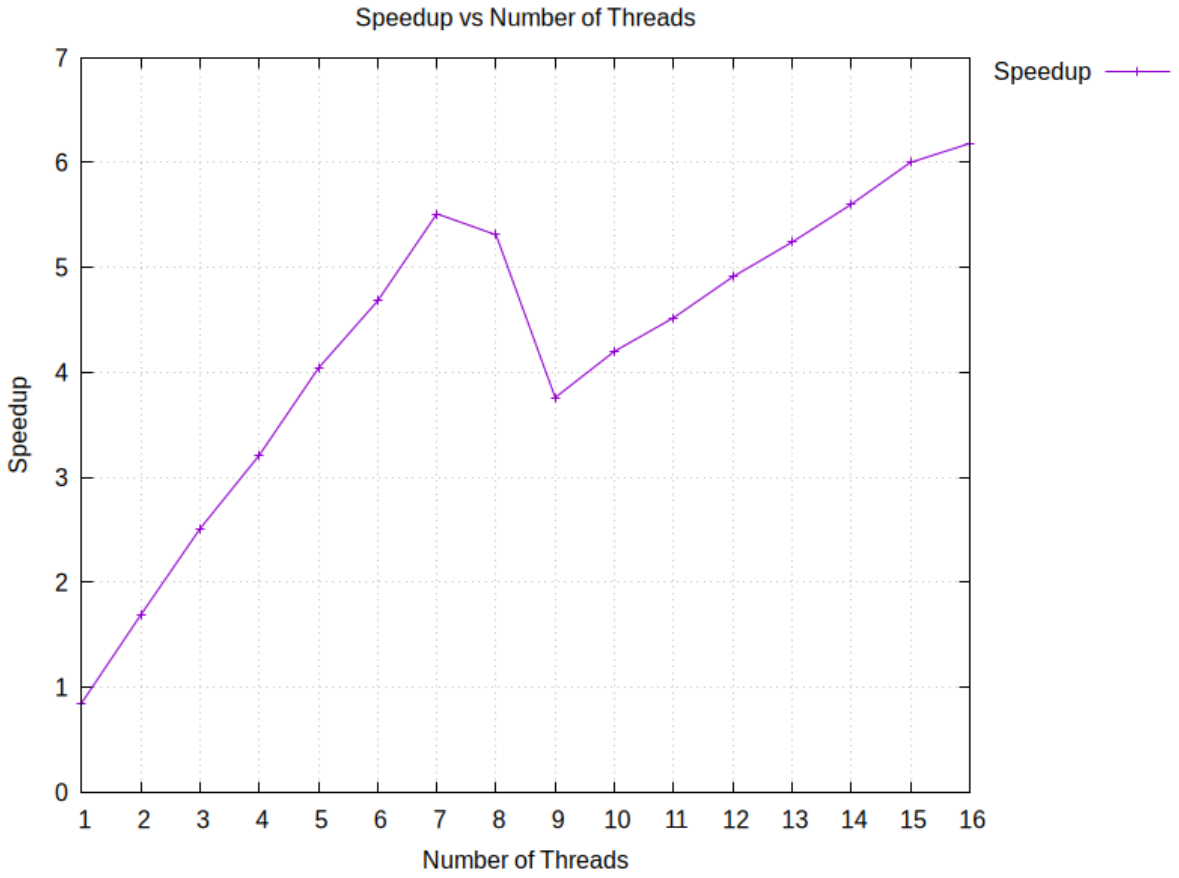

Рис. 4.2. График зависимости ускорения от числа потоков

4.3. Эффективность от числа потоков

$$E(threads) = \frac{S(threads)}{threads}$$

Где:

- S(threads) — ускорение для nnn потоков.
- threads — количество потоков.

Эффективность показывает, насколько эффективно используется каждый поток для ускорения работы программы.

Анализ:

- Эффективность для одного потока сразу же не равна 1 (100%), что свидетельствует о достаточно больших накладных расходах, однако тенденция около 0.8 сохраняется вплоть до 7 потоков, что говорит о достаточно большое стабильности такого алгоритма.
- На 9 и более потоках эффективность резко падает до значений около 0.4, что указывает на то, что увеличение числа потоков не дает значительного ускорения и приводит к потере производительности из-за накладных расходов на управление потоками и синхронизацию.

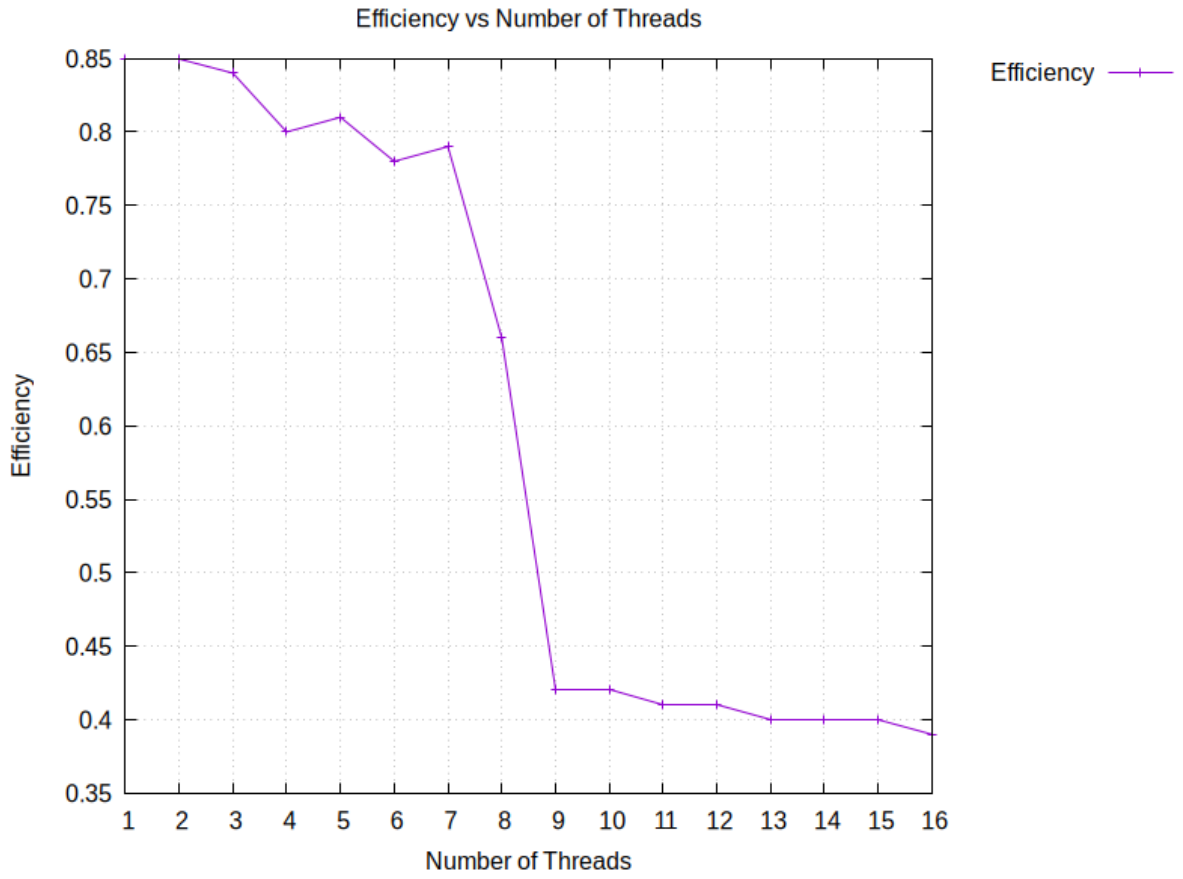

Рис. 4.3. График зависимости эффективности от числа потоков

Заключение

Анализ производительности программы на многопоточной системе выявил, что при увеличении числа потоков от 1 до 7 наблюдается значительное сокращение времени выполнения, рост ускорения (до 5.5) и сохранение эффективности на уровне от 0.78 до 0.85.

Максимальная эффективность в 0.85 достигается при использовании 1 и 2 потоков, что связано с минимальными накладными расходами и равномерным распределением нагрузки между ядрами. Однако дальнейшее увеличение числа потоков приводит к постепенному снижению эффективности: при 8 потоках она составляет уже 0.66, что указывает на появление накладных расходов, вызванных конкуренцией потоков за ресурсы.

Система демонстрирует максимальное ускорение при использовании 16 потоков, где его значение достигает 6.18. Однако это ускорение сопровождается значительным падением эффективности до 0.39, что объясняется значительными накладными расходами, возникающими из-за превышения числа потоков над количеством физических ядер.

Таким образом, использование 7 потоков можно считать наиболее сбалансированным решением: программа достигает ускорения в 5.51 при эффективности 0.79, что позволяет эффективно использовать доступные ресурсы без значительных накладных расходов.

Использование менее 7 потоков обеспечивает более высокую эффективность, но при этом не позволяет добиться максимального ускорения. Использование более 7 потоков, хотя и увеличивает ускорение, не является оптимальным из-за существенного падения эффективности (вплоть до 2 раз) и незначительного прироста производительности (около 1.2 раза).

Оптимальным числом потоков для моей системы является диапазон от 1 до 7, в зависимости от приоритетов между ускорением и эффективностью.

5. Приложение

5.1 Листинг параллельной программы

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

int main(int argc, char** argv)
{
   const int count = 10000000;     ///< Number of array elements
   const int seeds[10] = {920214, 123456, 654321, 789101, 112131,
                          141516, 171819, 202122, 232425, 262728};
   const int target = 16;          ///< Number to look for
   const int max_threads = 16;     ///< Maximum number of threads to test
   const int num_arrays = 10;      ///< Number of random arrays
   const int runs_per_array = 10;  ///< Number of runs per array

   int* array = 0;                 ///< Array for searching
   FILE* output_file = fopen("/home/dt/ParProg/lab2/solve/performance_data.txt", "w");

   if (!output_file) {
      printf("Failed to open file for writing.\n");
      return 1;
   }

   //fprintf(output_file, "# Threads\tTime (s)\tSpeedup\tEfficiency\n");

   double linear_avg_time = 0.0;   ///< Average time for linear search

   // Создание массива для хранения данных
   int** arrays = (int**)malloc(num_arrays * sizeof(int*));
   for (int i = 0; i < num_arrays; i++) {
      arrays[i] = (int*)malloc(count * sizeof(int));
      if (!arrays[i]) {
         printf("Memory allocation failed.\n");
         return 1;
      }

      // Генерация массива с использованием уникального seed
      srand(seeds[i]);
      for (int j = 0; j < count; j++) {
         arrays[i][j] = rand();
```

```
    }
}

// Линейный проход
for (int arr = 0; arr < num_arrays; arr++) {
    double total_time = 0.0;

    for (int run = 0; run < runs_per_array; run++) {
        double start_time = omp_get_wtime();
        int index_linear = -1;

        for (int i = 0; i < count; i++) {
            if (arrays[arr][i] == target) {
                index_linear = i;
                break;
            }
        }

        total_time += (omp_get_wtime() - start_time);
    }

    linear_avg_time += total_time / runs_per_array;
}

linear_avg_time /= num_arrays;
printf("Linear average time: %.6f seconds.\n", linear_avg_time);

// Тестирование с потоками от 1 до max_threads
for (int threads = 1; threads <= max_threads; threads++) {
    omp_set_num_threads(threads);

    double avg_time = 0.0;

    for (int arr = 0; arr < num_arrays; arr++) {
        double total_time = 0.0;

        for (int run = 0; run < runs_per_array; run++) {
            int index = -1;
            int found = 0;

            double start_time = omp_get_wtime();

            #pragma omp parallel
            {
```

```
            int local_index = -1;

            #pragma omp for
            for (int i = 0; i < count; i++) {
                if (found) continue;

                if (arrays[arr][i] == target) {
                    local_index = i;

                    #pragma omp atomic write
                    found = 1;
                }
            }

            // Обновление общего индекса
            if (local_index != -1) {
                #pragma omp critical
                {
                    if (index == -1 || local_index < index) {
                        index = local_index;
                    }
                }
            }
        }

        total_time += (omp_get_wtime() - start_time);
    }

    avg_time += total_time / runs_per_array;
}

avg_time /= num_arrays;

// Вычисление ускорения и эффективности
double speedup = linear_avg_time / avg_time;
double efficiency = speedup / threads;

printf("Threads: %d, Avg Time: %.6f s, Speedup: %.2f, Efficiency: %.2f\n",
       threads, avg_time, speedup, efficiency);

fprintf(output_file, "%d\t %.6f\t %.2f\t %.2f\n",
        threads, avg_time, speedup, efficiency);
}
```

```
    // Освобождение памяти
    for (int i = 0; i < num_arrays; i++) {
        free(arrays[i]);
    }
    free(arrays);

    fclose(output_file);

    printf("Performance data saved to 'performance_data.txt'.\n");
    return 0;
}
```

Листинг 1. Параллельная программа

5.2. Листинг Gnuplot-Script

```
# Установка выходного формата
set terminal pngcairo size 800,600 enhanced font "Arial,12"
set output 'performance_plots.png'

# Настройки стилей
set style data linespoints
set grid

# Установка общих меток
set xlabel "Number of Threads"
set xtics 1
set key outside

# Первый график: Среднее время от числа потоков
set title "Average Time vs Number of Threads"
set ylabel "Average Time (s)"
plot "performance_data.txt" using 1:2 with linespoints title "Average Time"

# Сохранение графика
set output 'average_time.png'
replot

# Второй график: Ускорение от числа потоков
set title "Speedup vs Number of Threads"
set ylabel "Speedup"
plot "performance_data.txt" using 1:3 with linespoints title "Speedup"

# Сохранение графика
```

```
set output 'speedup.png'
replot

# Третий график: Эффективность от числа потоков
set title "Efficiency vs Number of Threads"
set ylabel "Efficiency"
plot "performance_data.txt" using 1:4 with linespoints title "Efficiency"

# Сохранение графика
set output 'efficiency.png'
replot
```

Листинг 2. GNUPlot-Script для построения графиков

5.3. Таблица полученных данных

Threads	Avg Time	Speedup	Efficiency
1	0.010343	0.85	0.85
2	0.005209	1.69	0.85
3	0.003504	2.51	0.84
4	0.002744	3.21	0.80
5	0.002178	4.04	0.81
6	0.001884	4.68	0.78
7	0.001599	5.51	0.79
8	0.001660	5.31	0.66
9	0.002341	3.76	0.42
10	0.002099	4.20	0.42
11	0.001950	4.52	0.41
12	0.001795	4.91	0.41
13	0.001682	5.24	0.40
14	0.001574	5.60	0.40
15	0.001469	6.00	0.40
16	0.001426	6.18	0.39

Таблица 1. Полученные данные в ходе работы