

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования

**“Санкт-Петербургский национальный исследовательский университет  
информационных технологий, механики и оптики”**

**(НИУ ИТМО)**

---

Факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Программная инженерия

**Лабораторная работа №**

**“ Метод доверительных интервалов при измерении времени выполнения  
параллельной OpenMP-программы ”**

**По дисциплине “Параллельные вычисления”**

Студент группы Р4114

Шитов Григорий

Семенович

Преподаватель:

Жданов Андрей Дмитриевич

Санкт-Петербург, 2024 г.

## Оглавление

1. Описание решаемой задачи.....	3
2. Краткая характеристика системы .....	4
3. Программа lab3.c .....	5
4. Совместимость и вывод в консоль .....	10
5. Распараллеливание сортировки на 2 части.....	11
6. Разбиение сортировки на k-нитей .....	12
7. Доверительный интервал и минимальное значение.....	12
Выводы .....	13

## 1. Описание решаемой задачи

1. В программе, полученной в результате выполнения ЛР №3, так изменить этап Generate, чтобы генерируемый набор случайных чисел не зависел от числа потоков, выполняющих программу.

2. Заменить вызовы функции `gettimeofday` на `omp_get_wtime`.

3. Распараллелить вычисления на этапе Sort, для чего выполнить сортировку в два этапа:

- Отсортировать первую и вторую половину массива в двух независимых нитях (можно использовать OpenMP-директиву «parallel sections»).
- Объединить отсортированные половины в единый массив.

4. Написать функцию, которая один раз в секунду выводит в консоль сообщение о текущем проценте завершения работы программы. Указанную функцию необходимо запустить в отдельном потоке, параллельно работающем с основным вычислительным циклом. Нельзя использовать PThreads, сделать только средствами OpenMP.

5. Обеспечить прямую совместимость (forward compatibility) написанной параллельной программы. Для этого все вызываемые функции вида «omp\_\*» можно условно переопределить в препроцессорных директивах.

6. Провести эксперименты, варьируя  $N$  от  $\min(Nx/2, N1)$  до  $N2$ , где значения  $N1$  и  $N2$  взять из ЛР №1, а  $Nx$  — это такое значение  $N$ , при котором накладные расходы на распараллеливание превышают выигрыш от распараллеливания.

7. Необязательное задание №1. Уменьшить число итераций основного цикла с 100 до 10 и провести эксперименты, замеряя время выполнения следующими методами:

- Использование минимального из десяти полученных замеров.
- Расчёт по десяти измерениям доверительного интервала с уровнем доверия 95%.

8. Необязательное задание №2. В п. 3 задания на этапе Sort выполнить параллельную сортировку не двух частей массива, а  $k$  частей в  $k$  нитях (тредах), где  $k$  — это число процессоров (ядер) в системе, которое становится известным только на этапе выполнения программы с помощью команды: `int k = omp_get_num_procs()`

## **2. Краткая характеристика системы**

Операционная система: Ubuntu 22.04

Процессор: AMD® Ryzen 3 2200u with radeon vega mobile gfx × 4

Оперативная память: 8ГБ

Количество физических ядер: 2

Количество логических ядер: 4

gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

### 3. Программа lab3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include "omp.h"
#define _USE_MATH_DEFINES
#include <math.h>

int i;

#define M1SIZE (N)
#define M2SIZE (N / 2)

double sum = 0;

void gnomeSort(double *array, int size);
void Generate(double *M1, double *M2, unsigned int seed, int N);
void Map(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void Merge(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void Reduce(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize);
void sort(double *M2, size_t size);
void CompJob();

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Not enough args!\n Exit..");
        return 1;
    }
    int N, j;
    unsigned int seed;

    struct timeval T1, T2;
    double Tstart, Tend;
    long delta_ms;
    N = atoi(argv[1]); // N равен первому параметру командной строки
    int threadCount = atoi(argv[2]);
    int chunkSize = atoi(argv[3]);

    double *restrict M1 = (double *)calloc(M1SIZE, sizeof(double));
    double *restrict M2 = (double *)calloc(M2SIZE, sizeof(double));
    double *restrict M2temp = (double *)calloc(M2SIZE, sizeof(double));
    FILE *resultOfTest = fopen("Results/Results.txt", "a");

    gettimeofday(&T1, NULL); // запомнить текущее время T1
#ifdef _OPENMP
    Tstart = omp_get_wtime();
    omp_set_nested(1);
#endif
    // #pragma omp parallel for default(none) private(i,seed,j, sum) shared(N)
    num_threads(threadCount)
    #pragma omp parallel sections shared(i) num_threads(threadCount)
```

```

    {
#pragma omp section
    {
#ifdef _OPENMP
        CompJob();
#endif
    }
#pragma omp section
    {
        for (i = 0; i < 100; i++)
        {
            // 100 экспериментов
            seed = i * i; // инициализировать начальное значение ГСЧ
            // Заполнить массив исходных данных размером N
            Generate(M1, M2, seed, N);
            // Решить поставленную задачу, заполнить массив с результатами
            Map(M1, M2, M2temp, N, threadCount, chunkSize);

            // Merge
            Merge(M1, M2, M2temp, N, threadCount, chunkSize);

            // Отсортировать массив с результатами указанным методом mared(M1,
N)
            sort(M2, M2SIZE);

            // REDUCE
            Reduce(M1, M2, M2temp, N, threadCount, chunkSize);
        }
        free(M1);
        free(M2);
        free(M2temp);
#ifdef _OPENMP
        Tend = omp_get_wtime();
#endif
        printf("X: %lf\n", sum);
        gettimeofday(&T2, NULL); // запомнить текущее время T2
        delta_ms = (T2.tv_sec - T1.tv_sec) * 1000 +
            (T2.tv_usec - T1.tv_usec) / 1000;
        printf("\nN=%d. Milliseconds passed: %ld\n", N, delta_ms);
        printf("Work took %f seconds\n", Tend - Tstart);
        fprintf(resultOfTest, "%ld\n", delta_ms);
        fclose(resultOfTest);
    }
}
return 0;
}

void Generate(double *M1, double *M2, unsigned int seed, int N)
{
    int j;
    for (j = 0; j < M1SIZE; j++)
    {
        M1[j] = 1 + rand_r(&seed) % 360;
    }
    for (j = 0; j < M2SIZE; j++)
    {
        M2[j] = 360 + rand_r(&seed) % (10 * 360 - 361);
    }
}

```

```

void Map(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    int j;
#pragma omp parallel for default(none) private(j) shared(M1, N, chunkSize)
schedule(guided, chunkSize) num_threads(threadCount)
    for (j = 0; j < M1SIZE; j++)
    {
        M1[j] = 1.0 / tanh(sqrt(M1[j]));
    }

    // copy m2 array

#pragma omp parallel for default(none) private(j) shared(M2, M2temp, N,
chunkSize) schedule(guided, chunkSize) num_threads(threadCount)
    for (j = 0; j < M2SIZE; j++)
    {
        M2temp[j] = M2[j];
    }

    // new value in M2mak
#pragma omp parallel for default(none) private(j) shared(M2, M2temp, N,
chunkSize) schedule(guided, chunkSize) num_threads(threadCount)
    for (j = 1; j < M2SIZE; j++)
    {
        M2[j] += M2temp[j - 1];
        M2[j] = sqrt(M2[j] * M_E);
    }
}

void Merge(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    int j;
#pragma omp parallel for default(none) private(j) shared(M2, M1, N, chunkSize)
schedule(guided, chunkSize) num_threads(threadCount)
    for (j = 0; j < M2SIZE; j++)
    {
        M2[j] = M1[j] / M2[j];
    }
}

void Reduce(double *M1, double *M2, double *M2temp, int N, int threadCount, int
chunkSize)
{
    int j;
    double min = 10 * 360 + 1;
#pragma omp parallel for default(none) reduction(min : min) private(j) shared(M2,
N, chunkSize) schedule(guided, chunkSize) num_threads(threadCount)
    for (j = 0; j < M2SIZE; j++)
    {
        if (M2[j] < min)
        {
            min = M2[j];
        }
    }

#pragma omp parallel for default(none) reduction(+ : sum) private(j) shared(M2,
min, N, chunkSize) schedule(guided, chunkSize) num_threads(threadCount)

```

```

    for (j = 0; j < M2SIZE; j++)
    {
        if ((int)(M2[j] / min) % 2 == 0)
        {
            sum += sin(M2[j]);
        }
    }
}

void CompJob()
{
#pragma omp single nowait
    while (1)
    {
        if (i == 100)
        {
            printf("Job completed on %d% \n", i);
            break;
        }
        printf("Job completed on %d% \n", i);
        sleep(1);
    }
    return;
}

void gnomeSort(double *arr, int size)
{
    int pos = 0;
    while (pos < size)
    {
        if (pos == 0 || arr[pos] >= arr[pos - 1])
        {
            pos++;
        }
        else
        {
            double temp = arr[pos];
            arr[pos] = arr[pos - 1];
            arr[pos - 1] = temp;
            pos--;
        }
    }
}

void sort(double *M2, size_t size)
{
    size_t mid = size / 2;
#pragma omp parallel sections shared(M2, size)
    {
#pragma omp section
        gnomeSort(M2, mid);
#pragma omp section
        gnomeSort(M2 + mid, mid);
    }
    double temp[size];
    int i = 0, j = mid, k = 0;
    while (i < mid && j < size)
    {
        if (M2[i] <= M2[j])
        {

```



```

        temp[k++] = M2[i++];
    }
    else
    {
        temp[k++] = M2[j++];
    }
}
while (i < mid)
{
    temp[k++] = M2[i++];
}
while (j < size)
{
    temp[k++] = M2[j++];
}

// Копирование отсортированного массива обратно в исходный
for (int i = 0; i < size; i++)
{
    M2[i] = temp[i];
}
}

```

Листинг 1- Код программы lab3.c

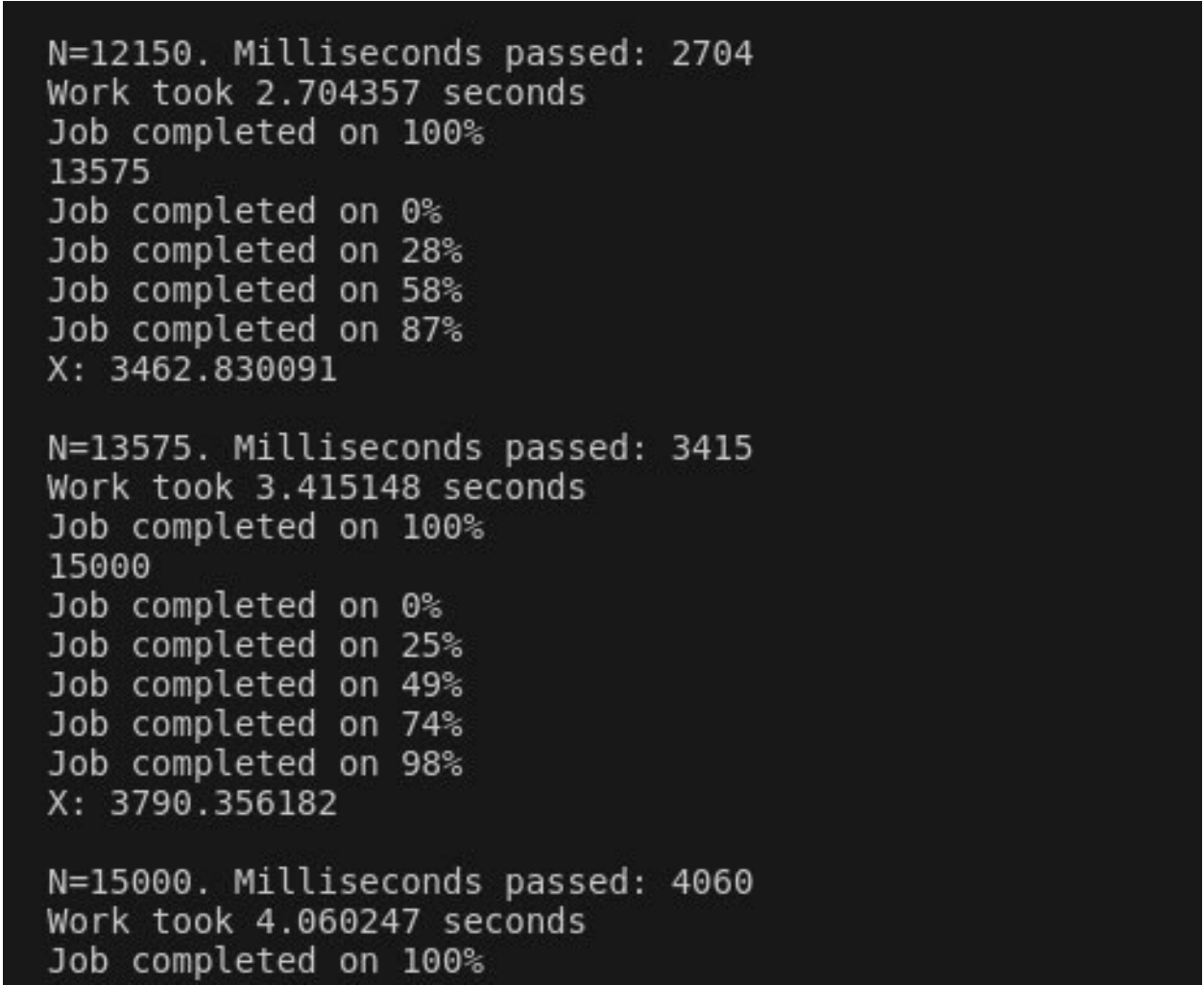
#### 4. Совместимость и вывод в консоль

Прямая совместимость достигается:

```
#ifdef _OPENMP  
    CompJob();  
#endif
```

Листинг 2 – прямая совместимость

Вывод в консоль процента выполнения задачи:



```
N=12150. Milliseconds passed: 2704  
Work took 2.704357 seconds  
Job completed on 100%  
13575  
Job completed on 0%  
Job completed on 28%  
Job completed on 58%  
Job completed on 87%  
X: 3462.830091  
  
N=13575. Milliseconds passed: 3415  
Work took 3.415148 seconds  
Job completed on 100%  
15000  
Job completed on 0%  
Job completed on 25%  
Job completed on 49%  
Job completed on 74%  
Job completed on 98%  
X: 3790.356182  
  
N=15000. Milliseconds passed: 4060  
Work took 4.060247 seconds  
Job completed on 100%
```

Рисунок 1- Отображение процента выполнения программы

## 5. Распараллеливание сортировки на 2 части

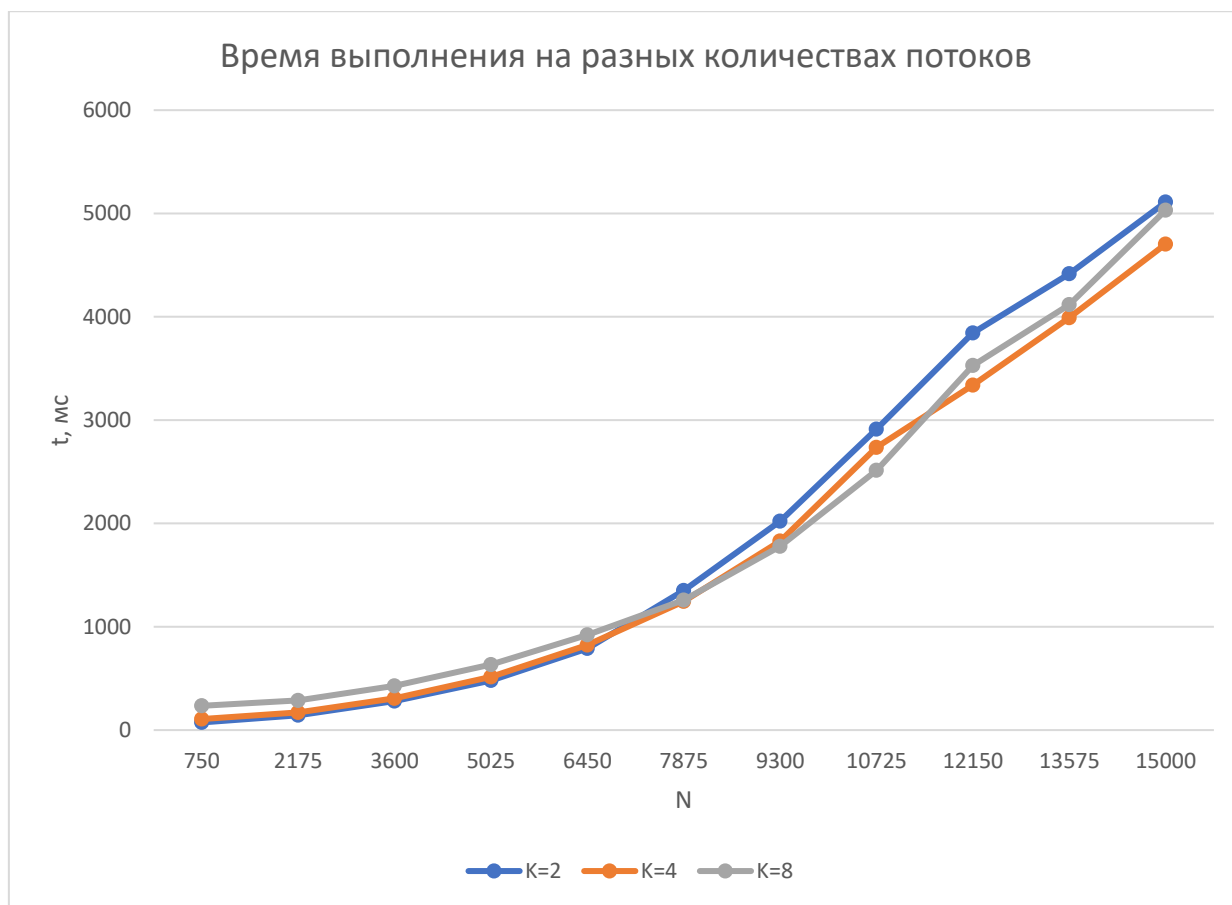


Рисунок 1- Результаты выполнения программы при распараллеливании сортировки на 2 части

Быстрее всего выполняется программа на 4 потоках, так как на моей машине максимум 4 потока.

N	k=2	k=4	k=8
750	74	107	235
2175	144	171	287
3600	281	307	428
5025	480	515	634
6450	790	823	921
7875	1351	1246	1258
9300	2022	1829	1779
10725	2913	2736	2514
12150	3844	3339	3529
13575	4416	3991	4117
15000	5110	4704	5032

Таблица 1- Результаты выполнения программы при распараллеливании сортировки на 2 части

## 6. Разбиение сортировки на k-нитей

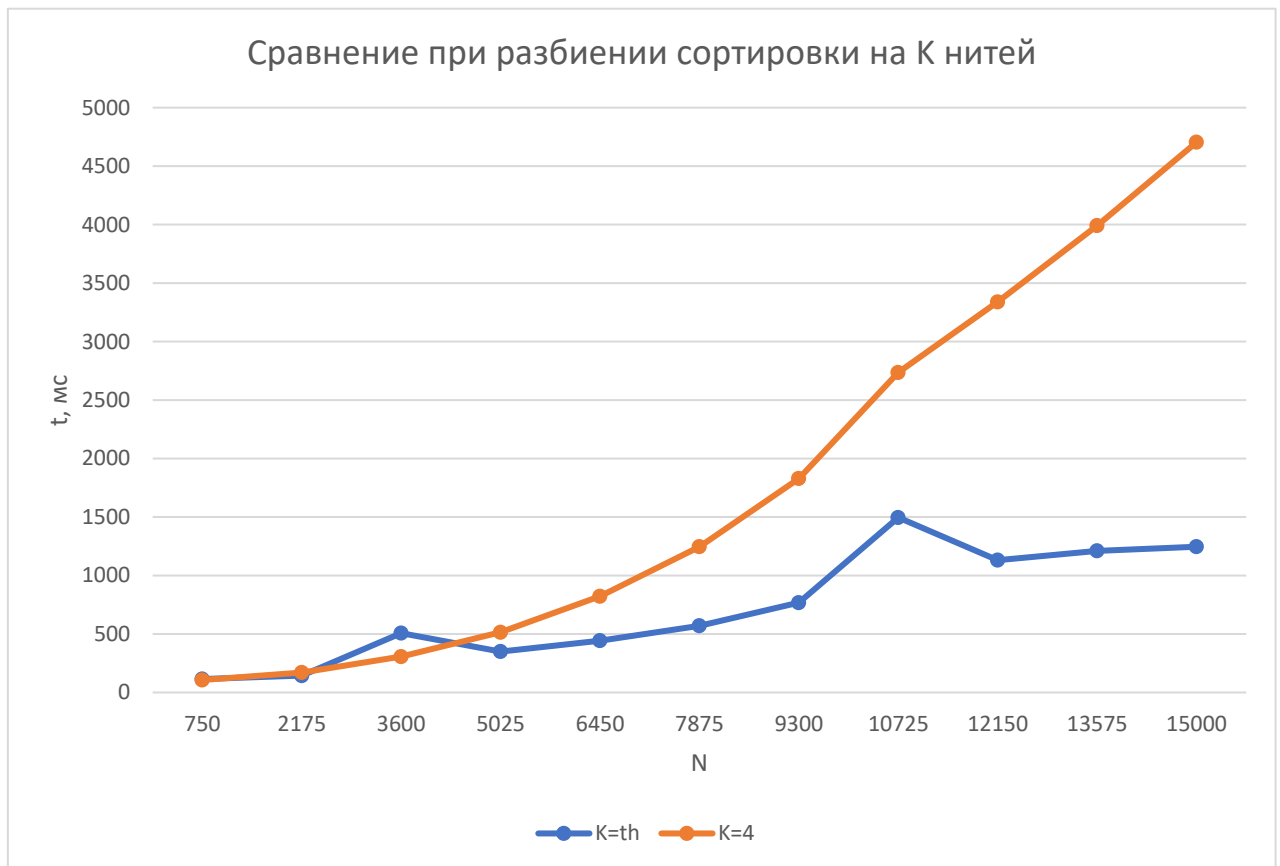


Рисунок 2- Результаты выполнения программы при распараллеливании сортировки на k частей

Для сравнения брался результат распараллеливания сортировки на 2 части при 4 потоках, так как при вызове функции `omp_get_num_procs()` возвращается значение 4. Из графика видно, что разбиение сортировки на k-нитей значительно ускоряет процесс выполнения программы.

N	k=th	k=2
750	115	74
2175	144	144
3600	507	281
5025	350	480
6450	443	790
7875	570	1351
9300	768	2022
10725	1496	2913
12150	1131	3844
13575	1211	4416
15000	1246	5110

Таблица 2- Результаты выполнения программы при распараллеливании сортировки на k частей

## 7. Доверительный интервал и минимальное значение

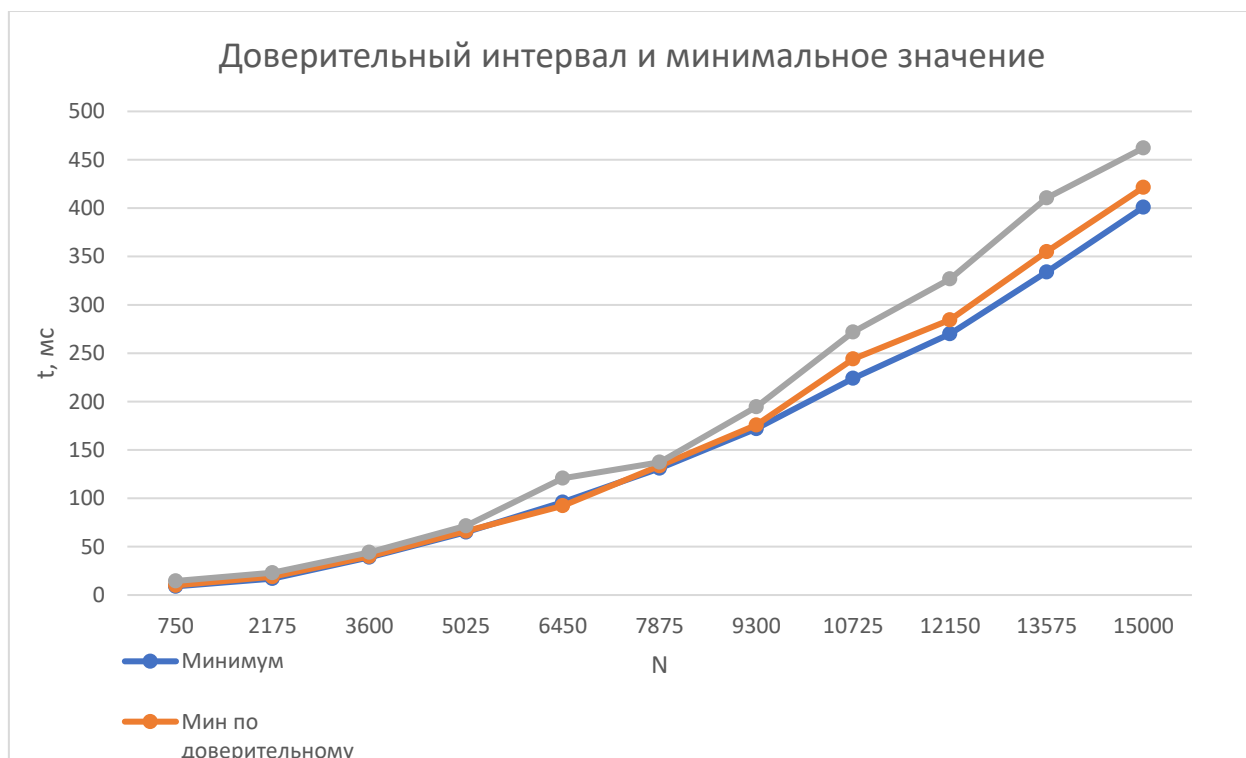


Рисунок 3 - Доверительный интервал и минимальное значение

## Выводы

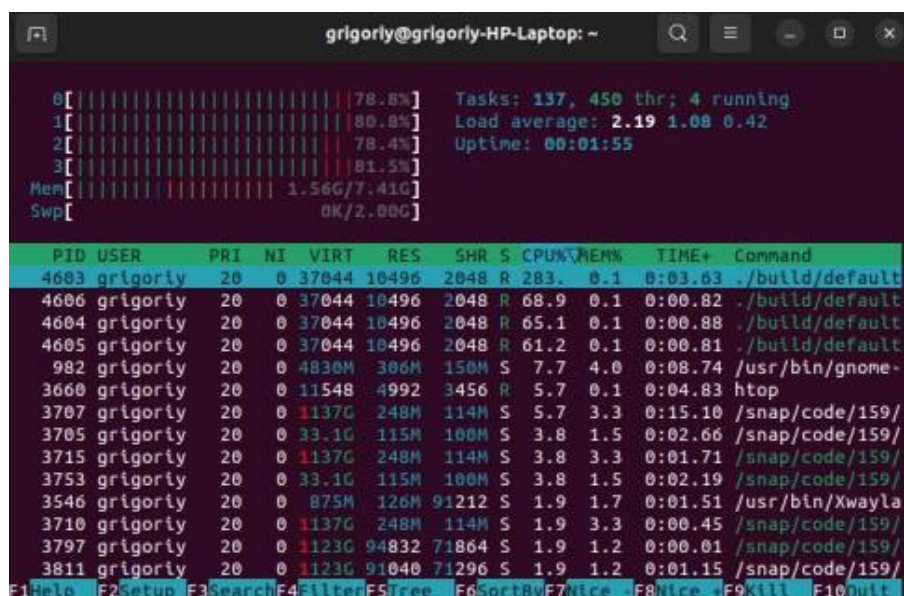


Рисунок 4 – Доказательство распараллеливания в начале

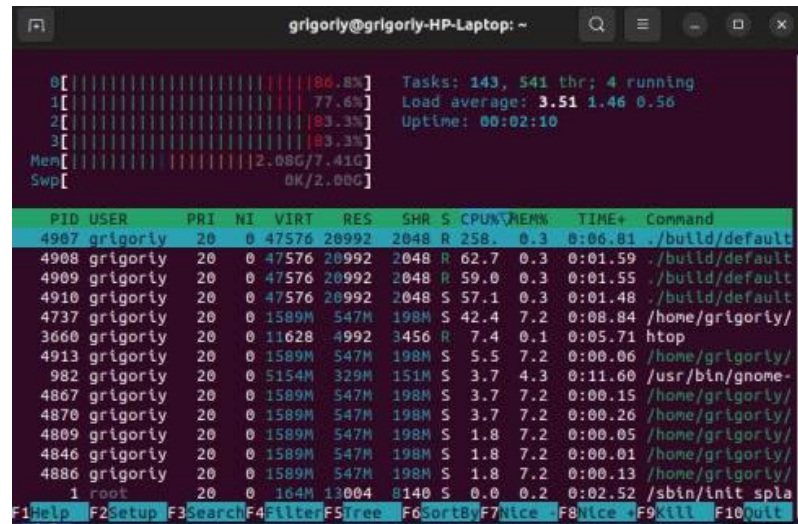


Рисунок 5 – Доказательство распараллеливания в конце

Увеличение количества секций распараллеливания на этапе сортировки дает значительный прирост к скорости выполнения всей программы