Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет Программной инженерии и компьютерной техники

Направление подготовки (специальность) Системное и прикладное ПО

ОТЧЕТ

Лабораторная работа №4
по предмету «Параллельные вычисления»

Тема проекта: «Метод доверительных интервалов при изменении времени выполнения параллельной OpenMP-программы».

Обучающийся Кирюшин В. А. (**Фамилия И.О.**) P4114 (**номер группы**)

Преподаватель Жданов А. Д. (**Фамилия И.О.**)

Санкт-Петербург
2023 г.

Содержание

Описание решаемой задачи

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

- Отсортировать первую и вторую половину массива в двух независимых нитях
- Объединить отсортированные половины в единый массив

”#pragma omp parallel for default(none) private(...) shared(...)”.

Наличие параметра default(none) является обязательным.

Все циклы необходимо проверить на наличие зависимостей между итерациями и при их наличии использовать специальные директивы OpenMP. В ряде случаев стоит вообще отказаться от распараллеливания цикла, но это необходимо обосновать.

Краткая характеристика «железа»

Операционная система: Ubuntu 22.04 LTS

Процессор: AMD Ryzen 5 4600H with Radeon Graphics

Кол-во физических ядер: 6

Кол-во логических ядер: 12

Семейство процессоров: 23

Модель: 96

Версия GCC: 11.3.0

Листинг программы lab4.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <unistd.h>

#ifdef _OPENMP
#include <omp.h>
#else
```

```
int omp_get_max_threads()
{
    return 1;
}

int omp_get_num_procs()
{
    return 1;
}

int omp_get_thread_num()
{
    return 0;
}

void omp_set_num_threads(int thrds)
{
    return;
}

double omp_get_wtime()
{
    struct timeval T;
    double time_ms;

    gettimeofday(&T, NULL);
    time_ms = (1000.0 * ((double)T.tv_sec) + ((double)T.tv_usec) / 1000.0);
    return (double)(time_ms / 1000.0);
}

void omp_set_nested(int b)
{
    return;
}
#endif

int min_el(int *restrict a, int * restrict b)
{
    return (*a) < (*b) ? (*a) : (*b);
}

void generate_array(double *restrict m, int size, unsigned int min, unsigned int
max, int seed)
{
    //int tmp_seed;
    // #pragma omp for private(tmp_seed)
    for (int i = 0; i < size; ++i)
    {
        unsigned int tmp_seed = sqrt(i + seed);
        m[i] = ((double)rand_r(&tmp_seed) / (RAND_MAX)) * (max - min) + min;
```

```
    }
}

void copy_array(double *dst, double *src, int n) {
    for (int i = 0; i < n; ++i)
    {
        dst[i] = src[i];
    }
}

void swap(double *xp, double *yp)
{
    double temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(double arr[], int start, int end)
{
    int i, j, min_idx;
    // One by one move boundary of unsorted subarray
    for (i = start; i < end-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < end; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
           if(min_idx != i)
            swap(&arr[min_idx], &arr[i]);
    }
}

void merge_sorted(double *src1, int n1, double *src2, int n2, double *dst) {
    int i = 0, i1 = 0, i2 = 0;
    while (i < n1 + n2) {
        dst[i++] = src1[i1] > src2[i2] && i2 < n2 ? src2[i2++] : src1[i1++];
    }
}

void sort_k(double * restrict MM, int size, double *restrict dst) {
    //int n_threads = omp_get_num_procs();
    int n_threads = omp_get_num_threads();
    int chunk_size = size / n_threads;
    int tid = omp_get_thread_num();
    int start = tid * chunk_size;
    int end = (tid == n_threads -1) ? size + 1 : (tid+1) * chunk_size;
    selectionSort(MM, start, end);
```

```
    #pragma omp single
    {
        double * restrict cpy = malloc(size * sizeof(double));

        copy_array(cpy, MM, size);
        copy_array(dst, MM, size);
        for (int k = 1; k < n_threads; ++k)
        {
            int n_done = chunk_size * k;
            int vsp = size - n_done;
            int n_cur_chunk = min_el(&(vsp), &(chunk_size));
            if(k==n_threads-1)
            {
                n_cur_chunk = size - n_done;
            }
            int n_will_done = n_done + n_cur_chunk;
            //printf("n_done = %d, n_cur_chunk = %d\n",n_done,n_cur_chunk);
            merge_sorted(cpy, n_done, MM + n_done, n_cur_chunk, dst);
            copy_array(cpy, dst, n_will_done);
        }
    }
}

void sort_half(double * restrict MM, int size, double *restrict dst, int
num_threads)
{
    int n1 = size / 2;
    //omp_set_num_threads(2);
    #pragma omp sections
    {
        #pragma omp section
        selectionSort(MM, 0, n1);
        #pragma omp section
        selectionSort(MM, n1, size + 1);
    }
    #pragma omp single
    merge_sorted(MM, n1, MM + n1, size - n1, dst);
    //omp_set_num_threads(num_threads);
}

int mainpart(int argc, char *argv[],int* progress, int *i)
{
    int N, key;
    double T1, T2, X;
    unsigned int seed;
    long long delta_ms;
    N = atoi(argv[1]);  /* N равен первому параметру командной строки */
    T1 = omp_get_wtime();
    int N_2 = N / 2;
```

```
    double A = 490.0; /* Ф*И*О */
    double min = 1; double max = A; double max_2 = max * 10;
    double *restrict M1 = malloc(N * sizeof(double));
    double *restrict M2 = malloc(N_2 * sizeof(double));
    double *restrict M2_old = malloc(N_2 * sizeof(double));
    double *restrict M2_sorted = malloc(N_2 * sizeof(double));
    const int num_threads = atoi(argv[2]); /* amount of threads */
    int extra_task = atoi(argv[3]);
    #if defined(_OPENMP)
        omp_set_dynamic(0);
        omp_set_num_threads(num_threads);
    #endif

    for (int j = 0; j < 100; ++j) {
        X = 0.0;
        seed = j;
        *i = j;
        /* Generate */
        generate_array(M1, N, min, max, seed);
        generate_array(M2, N_2, max, max_2, seed+2);
        /*--------------------------------------------------------------------*/
        #pragma omp parallel default(none) shared(N, N_2, M1, M2, M2_old,
M2_sorted, extra_task, num_threads, key, X)
        {
            // MAP
            #pragma omp for nowait
            for (int k = 0; k < N; ++k) {
                M1[k] = exp(sqrt(M1[k]));
            }
            #pragma omp for
            for (int k = 0; k < N_2; ++k) {
                M2_old[k] = M2[k];
            }
            #pragma omp for
            for(int k = 1; k < N_2; ++k) {
                M2[k] = M2[k] + M2_old[k-1];
            }
            #pragma omp for
            for(int k = 0; k < N_2; ++k) {
                M2[k] = log(fabs(tan(M2[k])));
            }
            /*--------------------------------------------------------------------
---*/
            // MERGE
            #pragma omp for
            for(int k=0; k < N_2; ++k) {
                M2[k]= M1[k] * M2[k];
            }
            /*--------------------------------------------------------------------
---*/
```

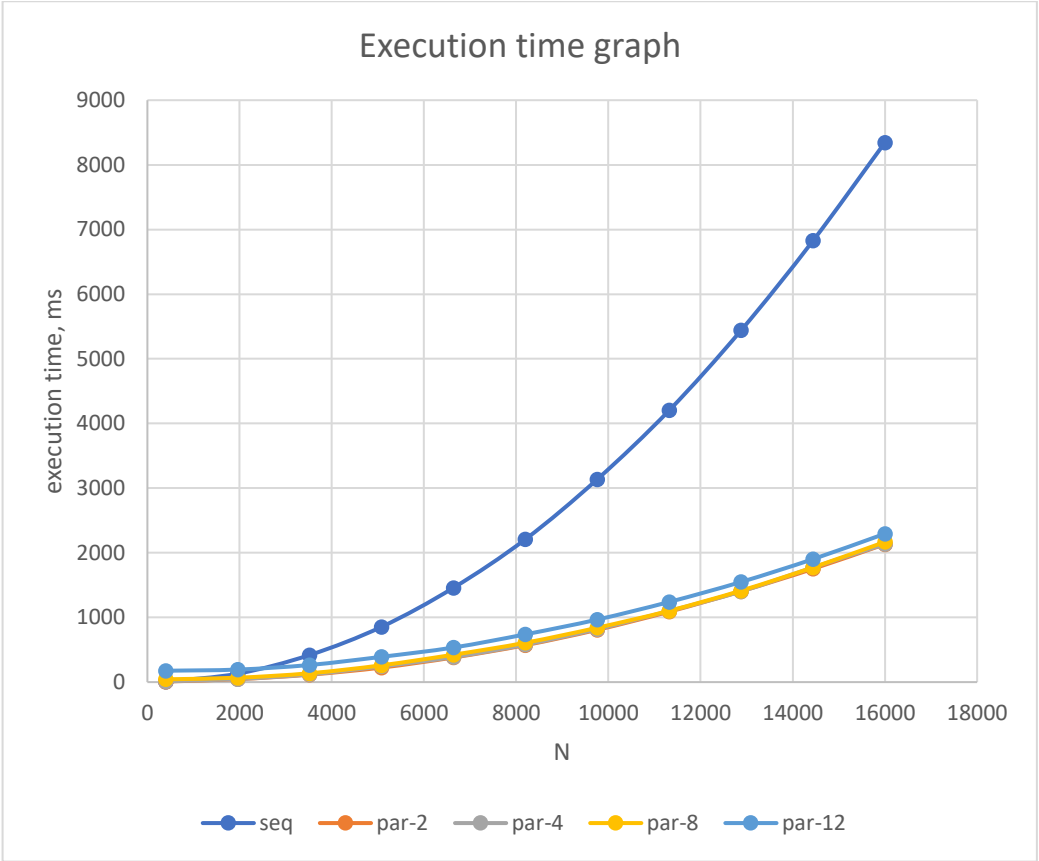
```
            // SORT
            if (extra_task == 0) {
                sort_half(M2, N_2, M2_sorted, num_threads);
            }
            else {
                sort_k(M2, N_2, M2_sorted);
            }
            /*-------------------------------------------------------------------
---*/
            // REDUCE
            #pragma omp single
            key = M2_sorted[0];
            /*
            #pragma omp single
            for (int k = 1; k < N_2; ++k) {
                if(M2_sorted[k] != 0) {
                    if (key == 0 || M2_sorted[k] < key) {
                        key = M2_sorted[k];
                    }
                }
            }
            */
            #pragma omp for reduction(+ : X)
            for(int k = 0; k < N_2; ++k) {
                if (((int)(M2_sorted[k] / key) % 2) == 0) {
                    X += sin(M2_sorted[k]);
                }
            }
            /*-------------------------------------------------------------------
---*/
        }
    }
    *progress = 1;
    printf("X= %f\n", X);
    T2 = omp_get_wtime();
    delta_ms = 1000* (T2 - T1);
    printf("%lld\n", delta_ms);
    return 0;
}

void progressnotifier(int *progress, int *i)
{
    double time = 0;
    while (*progress < 1)
    {
        double time_temp = omp_get_wtime();
        if (time_temp - time < 1)
        {
            usleep(100);
            continue;
```

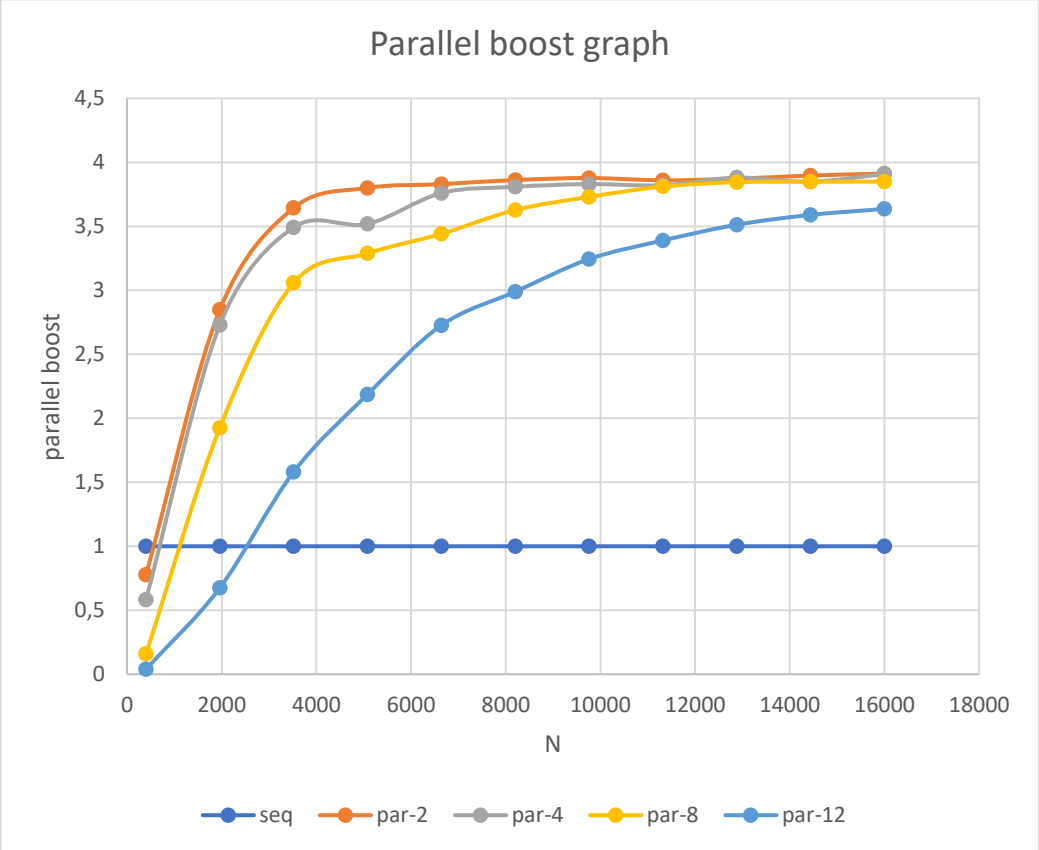
```
        };
        printf("\nPROGRESS: %d\n", *i);
        time = time_temp;
    }
}

int main(int argc, char *argv[])
{
    int *progress = malloc(sizeof(int));
    *progress = 0;
    int *i = malloc(sizeof(int));
    *i = 0;
    omp_set_nested(1);
    #pragma omp parallel sections num_threads(2) shared(i, progress)
    {
        #pragma omp section
        progressnotifier(progress, i);
        #pragma omp section
        mainpart(argc, argv, progress, i);
    }
    return 0;
}
```

Результаты экспериментов

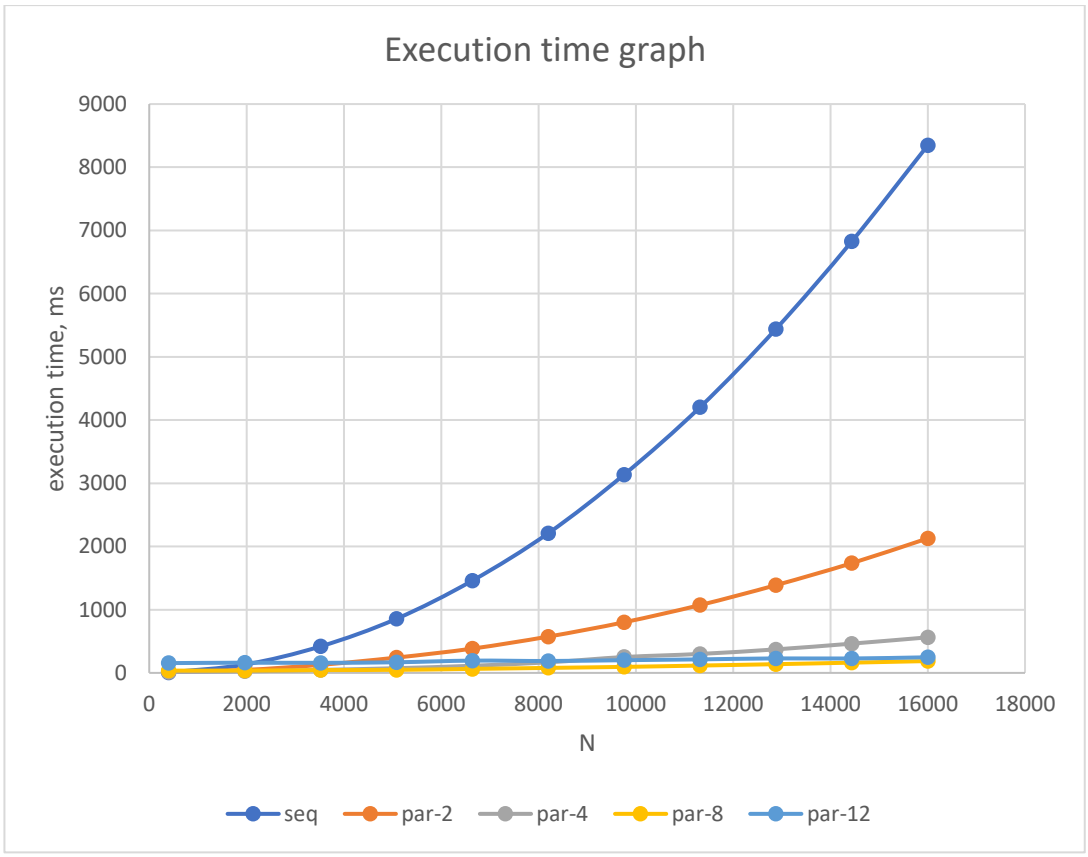
Сделаем сравнение с результатами распараллеливания из ЛР1.

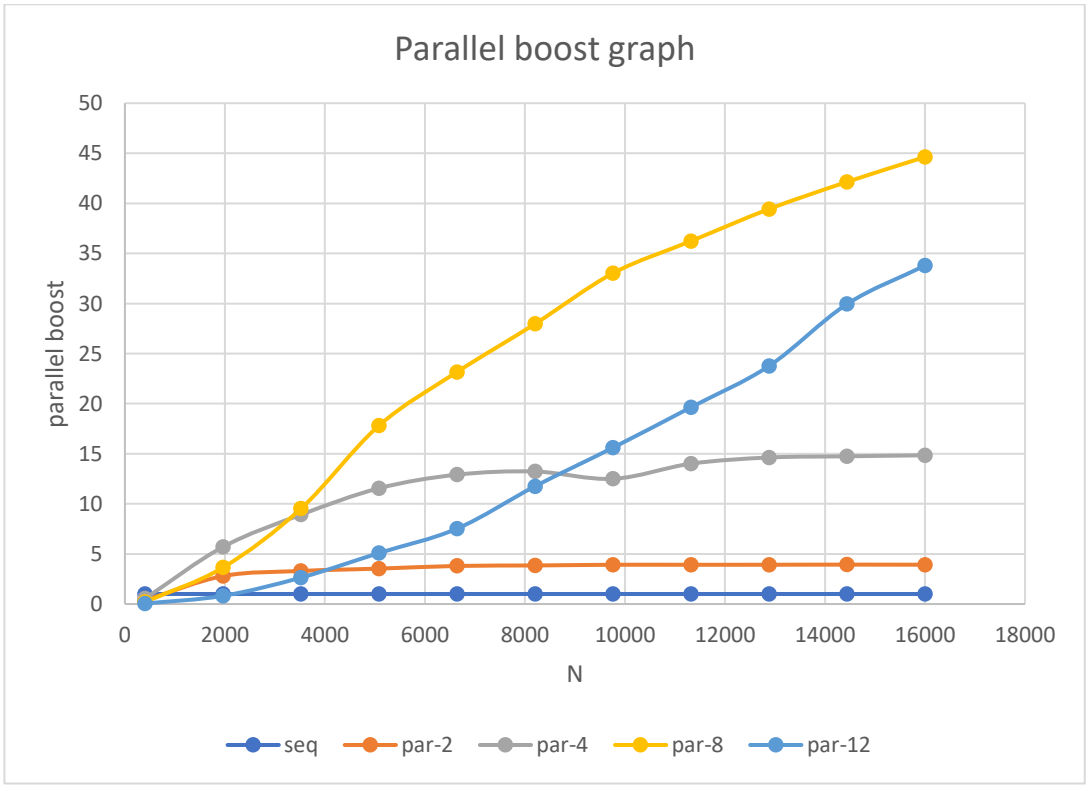
Рассмотрим графики времени выполнения и параллельного ускорения для каждой из работ. Результаты ЛР1 и результаты разделения массива для сортировки на 2 части:


Execution time graph
execution time, ms
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
N
seq
par-2
par-4
par-8
par-12

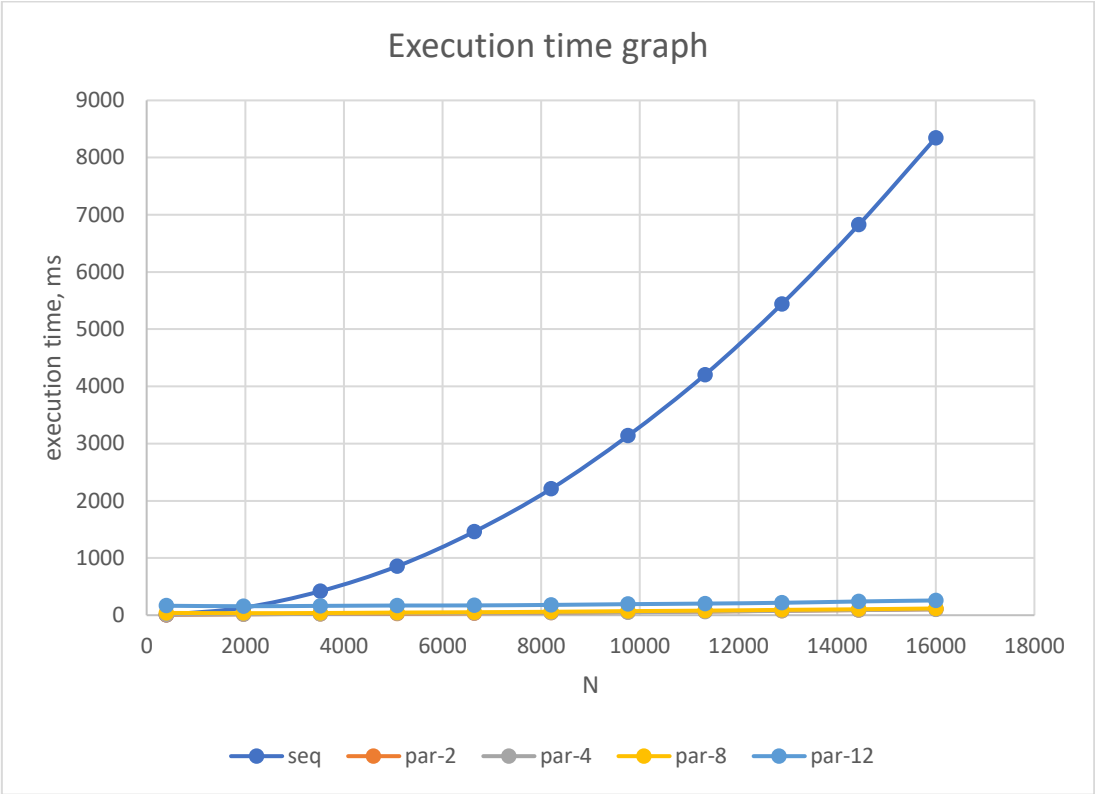

Parallel boost graph
parallel boost
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
N
seq
par-2
par-4
par-8
par-12

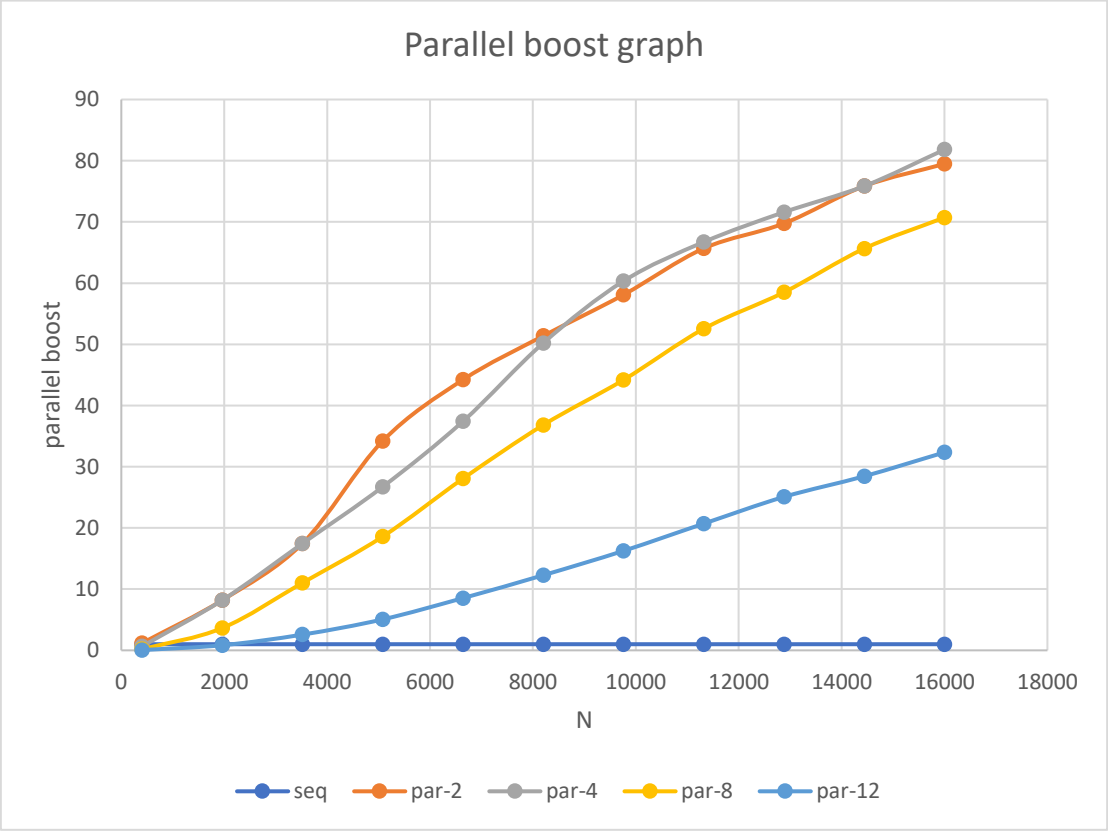
После чего приведём графики сравнения результатов ЛР1 и разбиения массива для сортировки на k-нитей, где k-число потоков:



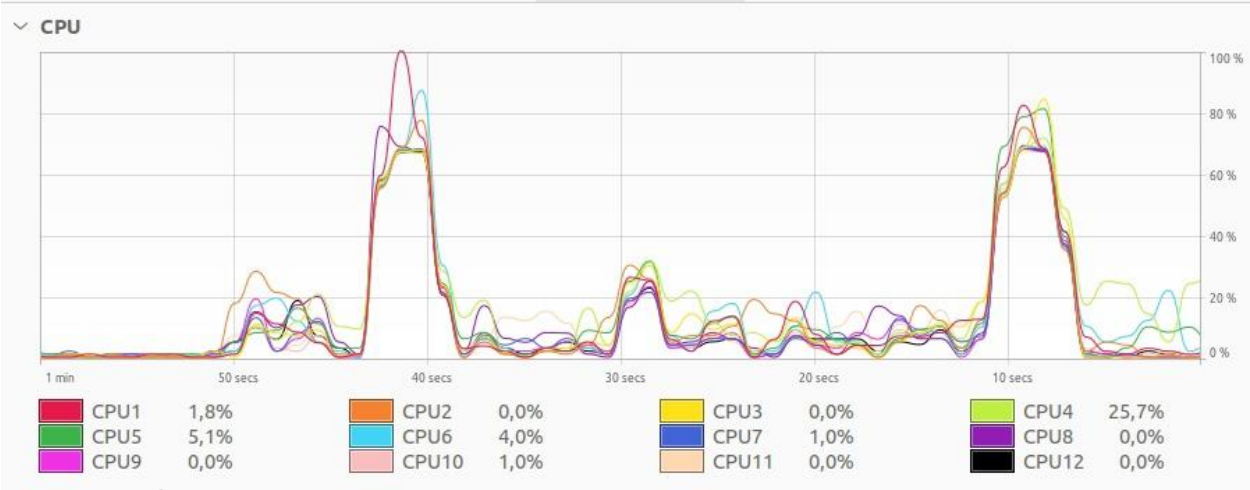


После чего приведём графики сравнения результатов ЛР1 и разбиения массива для сортировки на k-нитей, где k-число процессоров, k=12:





Системный монитор



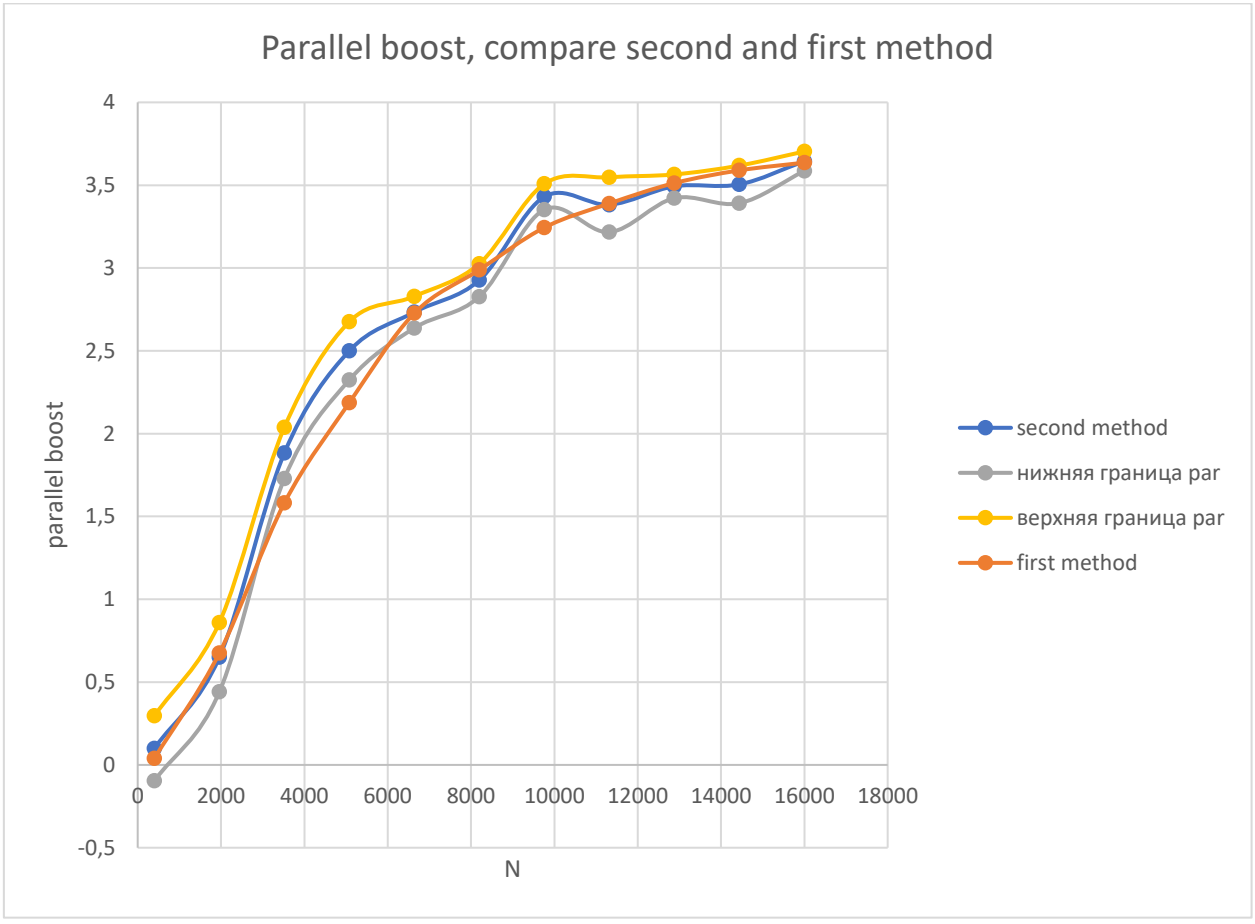
Доверительный интервал

Уменьшим количество итераций основного цикла с 100 до 10 и проведём эксперименты замеряя время выполнения следующими методами:

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

Приведём графики параллельного ускорения и время выполнения для обоих методов.


Execution time
execution time, ms
N
90
80
70
60
50
40
30
20
10
0
-10
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
par
seq
нижняя граница par
верхняя граница par
нижняя граница seq
верхняя граница seq


Parallel boost, compare second and first method
parallel boost
N
4
3,5
3
2,5
2
1,5
1
0,5
0
-0,5
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
second method
нижняя граница par
верхняя граница par
first method

Вывод

В рамках данной работы была распараллелена с помощью библиотеки OpenMP функция сортировки двумя методами. В первом случае массив, предназначенный для сортировки, разбивался на две части, каждая из которых сортировалась отдельно в соответствующем потоке. Параллельное ускорение получилось приблизительно равно 4, это связанно с тем, что алгоритмическая сложность по времени равна $O(N^2)$, разделяя массив на 2 части получаем сложность $O\left(\left(\frac{N}{2}\right)^2\right) = O(\frac{N^2}{4})$, то есть ускорение в 4 раза. Во втором случае массив разбивался на k частей, где k равно числу потоков. В это случае максимальное ускорение составило 45 для 8 потоков.

Так же параллельное ускорение было оценено альтернативным методом с помощью доверительного интервала, уровень доверия – 95%.

Для каждого этапа были построены соответствующие графики времени выполнения программы и параллельного ускорения от размера массива N.