

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 № 4

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

По дисциплине “Параллельные вычисления”

Студент группы Р4114

Трофимова Полина
Владимировна

Преподаватель:

Жданов Андрей Дмитриевич

Санкт-Петербург, 2024 г.

Оглавление

Описание решаемой задачи	3
Краткая характеристика системы	3
Программа lab1.c	4
Результаты	8
Разделение сортировки на 2 секции	9
Выводы	11

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()`

Краткая характеристика системы

Операционная система: Windows 10 Домашняя

Тип системы: 64-разрядная операционная система

Процессор: AMD Ryzen 7 5700U

Оперативная память: 8ГБ

Количество физических ядер: 8

Количество логических ядер: 16

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

Программа lab1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <sys/time.h>
#include <omp.h>
// #define _OPENMP
#ifdef _OPENMP
// here omp_get_num_procs implementation exists
#else
int omp_get_num_procs() { return 1; }
double omp_get_wtime() {
    struct timeval t;
    gettimeofday(&t, NULL);
    return t.tv_sec + t.tv_usec / 1000000.0;
}
#endif
// Сортировка расчёской
void combSort(double arr[], int n) {
    int gap = n;
    bool swapped = true;
    int temp;
    while (gap != 1 || swapped) {
        gap *= 10 / 13;
        if (gap < 1) {
            gap = 1;
        }
        swapped = false;
        for (int i = 0; i < n - gap; i++) {
            if (arr[i] > arr[i + gap]) {
                temp = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = temp;
                swapped = true;
            }
        }
    }
}
// Объединение отсортированного
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++];
    }
}
// Копирование массива
void copy_array(double* dst, double* src, int n) {
    for (int i = 0; i < n; ++i) {
        dst[i] = src[i];
    }
}
// Сортировка по половинам
void sort(double* array, int n, double* dst) {
    int n1 = n / 2;
    int n2 = n - n1;
#pragma omp sections
    {
#pragma omp section
    {
```

```

        combSort(array, n1);
    }
#pragma omp section
    {
        combSort(array + n1, n2);
    }
}
merge_sorted(array, n1, array + n1, n2, dst);
}
//Сортировка >2 частей массива
void sort_H(double* array, int n, double* dst, int n_threads)
{
    int n_chunk = 0;
#pragma omp single
    n_chunk = n_threads < 2 ? n : (int)((double)(n + n_threads - 1) / n_threads);

#pragma omp single
    for (int k = 0; k < n_threads; ++k)
    {
        //int n_chunk = n_threads < 2 ? n : (int)((double)(n + n_threads - 1) /
n_threads);
        int n_l = n_chunk * k;
        int n_cur_chunk = fmin((n - n_l), n_chunk);
        printf("Parallel for k: %d n_chunk: %d n_l: %d n_cur_chunk: %d \n", k,
n_chunk, n_l, n_cur_chunk);
        combSort(array + n_l, n_cur_chunk);
    }
#pragma omp single
    {
        double* restrict cpy = malloc(n * sizeof(double));
        copy_array(cpy, array, n);
        copy_array(dst, array, n);
        for (int k = 1; k < n_threads; ++k)
        {
            int n_l = n_chunk * k;
            int n_cur_chunk = fmin(n - n_l, n_chunk);
            int n_will_done = n_l + n_cur_chunk;
            merge_sorted(cpy, n_l, array + n_l, n_cur_chunk, dst);
            copy_array(cpy, dst, n_will_done);
        }
        free(cpy);
    }
}

int main(int argc, char* argv[])
{
    int i, j, N, N2;
    int N_sort_threads;
    double T1, T2;
    T1 = omp_get_wtime(); // Заменяем gettimeofday на omp_get_wtime для начала
измерения времени
    double A = 2;
    long delta_ms;
    double sum_sin = 0.0;
    double min_nonzero = 0;
    int progress = 0;

    omp_set_nested(2);
#pragma omp parallel sections num_threads(N_sort_threads) shared(i, progress)
//shared(i)
    {
        #ifdef _OPENMP
        #pragma omp section
        {
            double time = 0;

```

```

while (progress < 1)
{
    double time_temp = omp_get_wtime();
    if (time_temp - time < 1)
    {
        usleep(10);
        continue;
    };
    printf("\rPROGRESS: %d \n", i);
    fflush(stdout);
    time = time_temp;
}
}
#endif

#pragma omp section
{
    // N равен первому параметру командной строки
    N = atoi(argv[1]);
    N2 = N / 2;
    N_sort_threads = atoi(argv[2]);
    //int W = argc > 2 ? atoi(argv[3]) : 2;
    //omp_set_num_threads(W);

    double* restrict M1 = (double*)malloc(N * sizeof(double));
    double* restrict M2 = (double*)malloc(N / 2 * sizeof(double));
    double* restrict M2_copy = (double*)malloc(N / 2 * sizeof(double));

    for (i = 0; i < 10; i++) // 100 экспериментов
    {
        //seed = i; // инициализировать начальное значение ГСЧ
        sum_sin = 0.0;
#pragma omp parallel default(none) shared(N, N2, A, M1, M2, M2_copy, i, j, sum_sin,
min_nonzero, N_sort_threads)
        {
#pragma omp single
            {
                unsigned int seed = i;
                for (j = 0; j < N; j++) // Заполнить массив исходных
данных размером N
                {
                    M1[j] = (((double)rand_r(&seed) / (RAND_MAX)) * A);
                }
                for (j = 0; j < N / 2; j++)
                {
                    M2[j] = ((double)rand_r(&seed) / (RAND_MAX)) * A *
10;
                    M2_copy[j] = M2[j];
                }
                //map
#pragma omp for
                for (j = 0; j < N; j++)
                {
                    // Гиперболический тангенс с последующим уменьшением на
1
                    M1[j] = ((tanh(M1[j])) - 1);
                }

#pragma omp for
                for (j = 1; j < N / 2; j++)
                {
                    // Десятичный логарифм, возведенный в степень e
                    M2[j] = pow((M2[j] + M2_copy[j - 1]), M_E);
                }
            }
        }
    }
}

```

```

    }

#pragma omp single
    M2[0] = pow(M2[0], M_E);

#pragma omp for
    for (j = 0; j < N / 2; j++) // Решить поставленную задачу,
    заполнить массив с результатами
    {
        //Деление (т.е. M2[i] = M1[i]/M2[i])
        M2[j] = M1[j] / M2[j];
    }

    //combSort(M2, N / 2);
    // Сортировка расчёской (Comb sort) результатов

#ifdef _OPENMP
    if (N_sort_threads == 2) {
        sort(M2_copy, N2, M2);
    }
    else {
        //sort_H(M2_copy, N2, M2, W);
        sort_H(M2_copy, N2, M2, omp_get_num_procs);
    }
#endif

#pragma omp for reduction(min:min_nonzero)
    for (j = 0; j < N / 2; j++)
    {
        if ((M2[j] != 0) && (M2[j] < min_nonzero))
        {
            min_nonzero = M2[j];
        }
    }

#pragma omp for reduction (+:sum_sin)
    for (j = 0; j < N / 2; j++)
    {
        if ((int)(M2[j] / min_nonzero) % 2 == 0)
        {
            sum_sin += sin(M2[j]);
        }
    }
    printf("sum_sin = %.10lf\n\n", sum_sin);

    }
    progress = 1;

    free(M1);
    free(M2);
    free(M2_copy);
}

//gettimeofday(&T2, NULL); // запомнить текущее время T2
T2 = omp_get_wtime();
delta_ms = (T2 - T1) * 1000;
printf("\nN=%d. Milliseconds passed: %ld\n", N, delta_ms);

return 0;
}

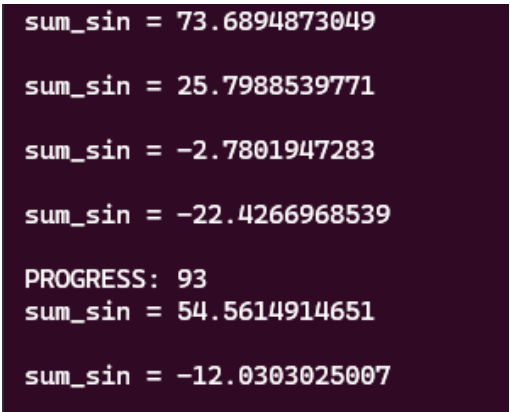
```

Результаты

Прямая совместимость достигается:

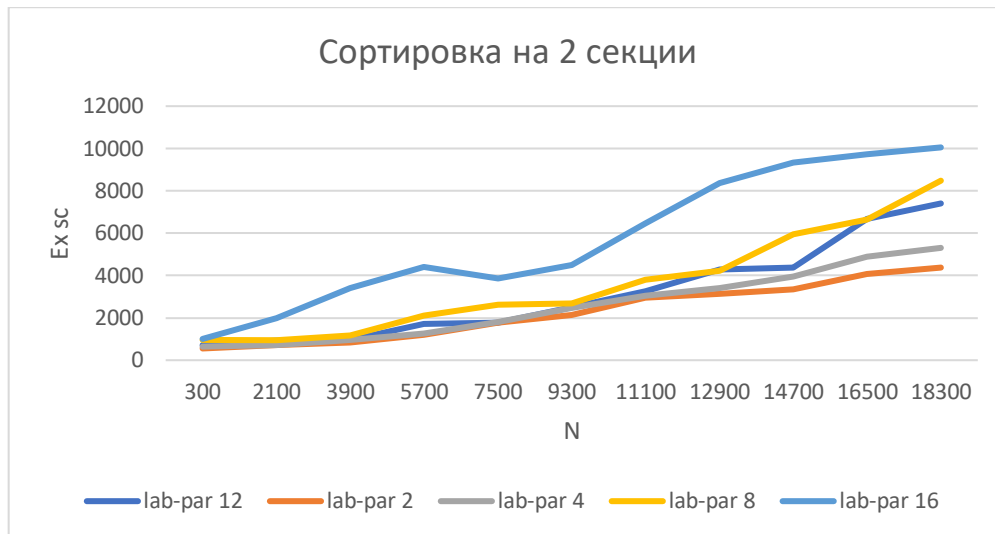
```
#ifdef _OPENMP
// here omp_get_num_procs implementation exists
#else
int omp_get_num_procs() { return 1; }
double omp_get_wtime() {
    struct timeval t;
    gettimeofday(&t, NULL);
    return t.tv_sec + t.tv_usec / 1000000.0;
}
#endif
```

Вывод в консоль текущего процента завершения программы:



```
sum_sin = 73.6894873049
sum_sin = 25.7988539771
sum_sin = -2.7801947283
sum_sin = -22.4266968539
PROGRESS: 93
sum_sin = 54.5614914651
sum_sin = -12.0303025007
```


Разделение сортировки на 2 секции



Наибольший прирост даёт разделение сортировки на 2 потока, т.к. массив делится пополам.



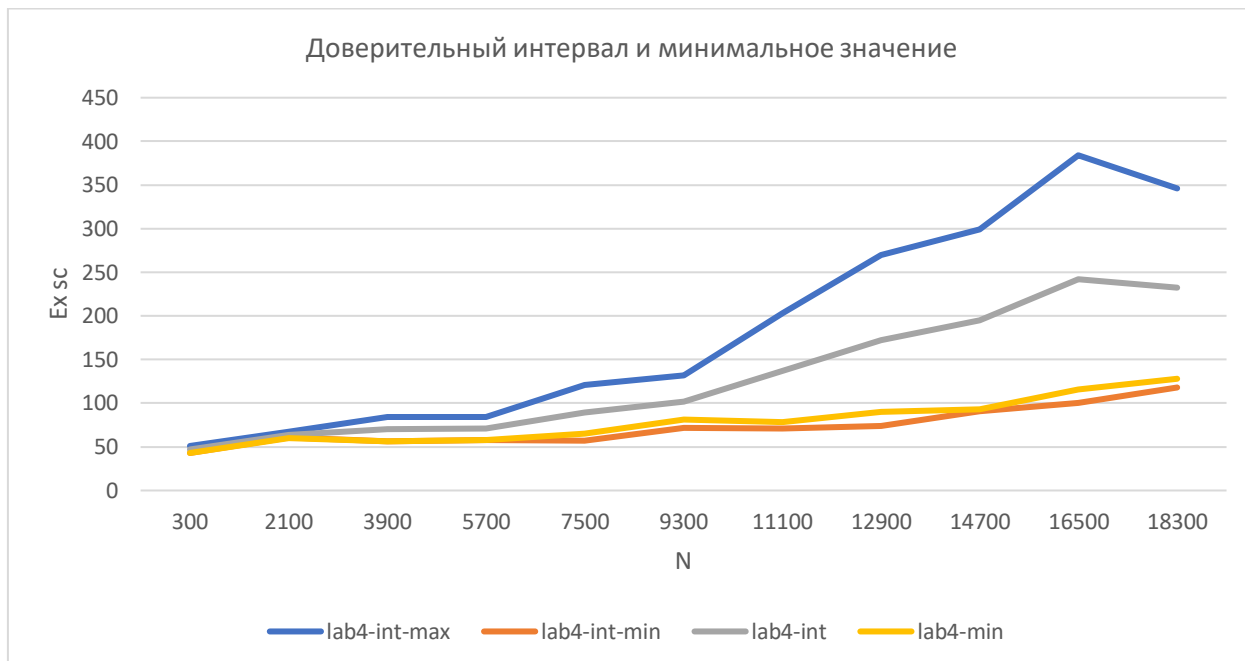
При разделении сортировки на количество существующих логических потоков (`int k = omp_get_num_procs()`), программа выполняется очень быстро, т.к. каждая секция сортировки обрабатывается своей нитью.

```
Parallel for k: 0 n_chunk: 572 n_l: 0 n_cur_chunk: 572
Parallel for k: 1 n_chunk: 572 n_l: 572 n_cur_chunk: 572
Parallel for k: 2 n_chunk: 572 n_l: 1144 n_cur_chunk: 572
Parallel for k: 3 n_chunk: 572 n_l: 1716 n_cur_chunk: 572
Parallel for k: 4 n_chunk: 572 n_l: 2288 n_cur_chunk: 572
Parallel for k: 5 n_chunk: 572 n_l: 2860 n_cur_chunk: 572
Parallel for k: 6 n_chunk: 572 n_l: 3432 n_cur_chunk: 572
Parallel for k: 7 n_chunk: 572 n_l: 4004 n_cur_chunk: 572
Parallel for k: 8 n_chunk: 572 n_l: 4576 n_cur_chunk: 572
Parallel for k: 9 n_chunk: 572 n_l: 5148 n_cur_chunk: 572
Parallel for k: 10 n_chunk: 572 n_l: 5720 n_cur_chunk: 572
Parallel for k: 11 n_chunk: 572 n_l: 6292 n_cur_chunk: 572
Parallel for k: 12 n_chunk: 572 n_l: 6864 n_cur_chunk: 572
Parallel for k: 13 n_chunk: 572 n_l: 7436 n_cur_chunk: 572
Parallel for k: 14 n_chunk: 572 n_l: 8008 n_cur_chunk: 572
Parallel for k: 15 n_chunk: 572 n_l: 8580 n_cur_chunk: 570
sum_sin = 76.4260798717
```

Доверительный интервал и минимальное значение

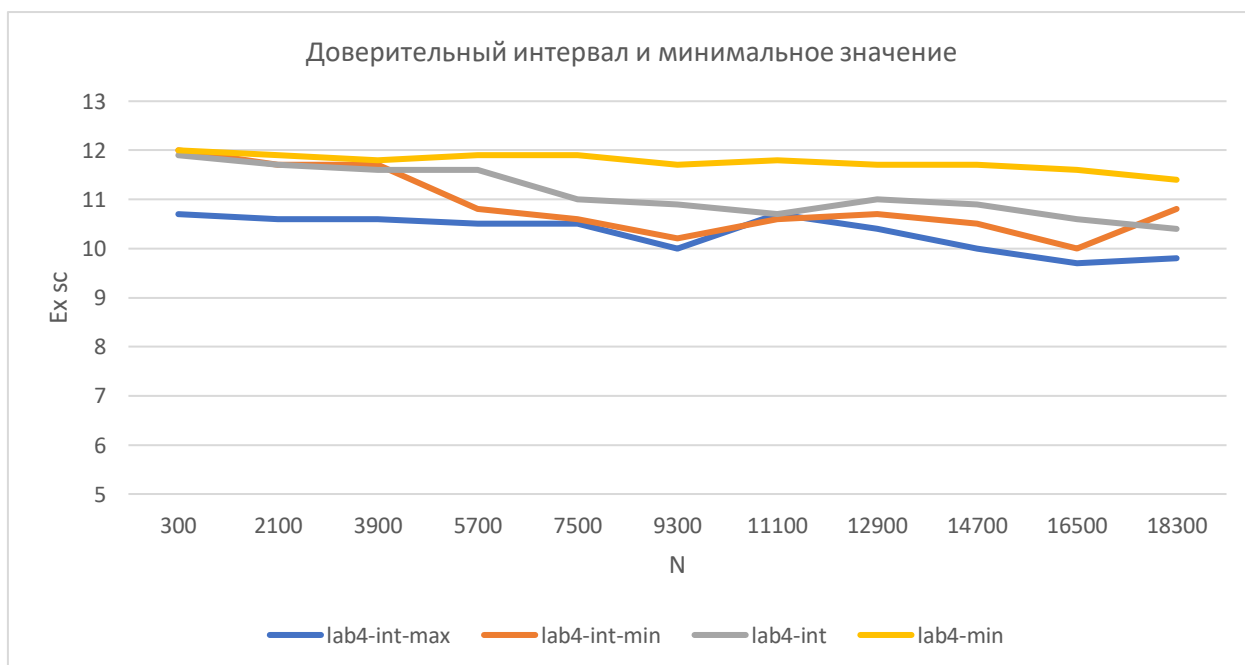
Количество экспериментов снижено до 10.

Доверительный интервал вычислялся методом Стьюдента.



Лучший результат показали замеры для минимального значения доверительного интервала.

График параллельного ускорения.



Выводы

При разделении сортировки на 2 секции улучшилась производительность.

При разделении сортировки на количество секции, равное количеству вычислительных ядер, производительность значительно увеличивается.