Министерство образования и науки Российской Федерации
Университет ИТМО
Кафедра вычислительной техники

ДИСЦИПЛИНА
«ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ»

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

Фамилия: Кудашев
Имя: Дмитрий
Отчество: Анатольевич
Группа: P4110

СОДЕРЖАНИЕ

Оглавление

Описание задачи

В рамках выполнения работы необходимо было модифицировать программный код, полученный в результате выполнения лабораторной работы №3, так, чтобы генерируемый набор случайных чисел не зависел от количества потоков, выполняющих программу, а для измерения времени выполнения использовать функцию omp_get_wtime из библиотеки OpenMP.

Далее на этапе Sort необходимо было распараллелить вычисления таким образом, чтобы массив разбивался на n частей (n – количество вычислителей на ПК), а затем объединить отсортированные части в единый массив.

На четвертом этапе выполнения лабораторной работы необходимо было выводить сообщение о текущем проценте завершения работы программы. Данный функционал должен был быть запущен в отдельном потоке, параллельно работающем с основным вычислительным циклом.

На пятом этапе выполнения работы необходимо было обеспечить прямую совместимость разработанной параллельной программы.

На последнем этапе необходимо было провести эксперименты, замеряя время двумя способами:

1. Использование минимального из дести полученных замеров;
2. Расчет по десяти измерениям доверительного интервала.

Также на последнем этапе необходимо было привести графики параллельного ускорения для обоих методов в одной системе координат. При этом нижнюю и верхнюю границу доверительного интервала првиести двумя независимыми графиками.

Характеристики системы.

Процессор:

- Модель: Intel i7-770HQ;
- Количество ядер: 4;
- Количество потоков: 8 (во время проведения экспериментов технология HT была отключена);

- Базовая частота: 2,80 GHz;
- Максимальная тактовая частота с технологией Turbo Boost: 3,80 GHz (во время проведения экспериментов Turbo Boost был отключен);

ОЗУ: размер 8 Гб.

Операционная система: Ubuntu 16.04.03 LTS.

Использованный компилятор: icc version 18.0.1 (gcc version 5.0.0 compatibility).

Программный код.

Текст программы:

```
#define _POSIX_C_SOURCE 1
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

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

#ifdef _OPENMP
	#include <omp.h>
#else
int omp_get_num_procs()
{
	return 1;
}

int omp_get_thread_num()
{
	return 1;
}

void omp_set_num_threads(int thrds)
{
	return;
}

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

	gettimeofday(&T, NULL);
```

```
      time_ms = ((double) T.tv_sec) + ((double)T.tv_usec) / 1000000;
      return (double)time_ms;
}
void omp_set_nested(int b){
      return;
}
#endif


int get_x(int a, int b){
    return 1 + ((a % 47) % b);
}

void print_variants(){
    int a = 7*7*11;
    printf("Этап 2, таблица 1, вариант № %d\n", get_x(a, 7));
    printf("Этап 2, таблица 2, вариант № %d\n", get_x(a, 8));
    printf("Этап 3, вариант № %d\n", get_x(a, 6));
    printf("Этап 4, вариант № %d\n", get_x(a, 7));
}

double get_M1_el(int A, unsigned int *seedp){
    return 1 + (A - 1) * ((double)rand_r(seedp) / RAND_MAX);
}

double get_M2_el(int A, int m2_end, unsigned int *seedp){
    return A + (m2_end - A) * ((double)rand_r(seedp) / RAND_MAX);
}

void swap(double *x, double *y){
    double t;
    t = *x;
    *x = *y;
    *y = t;
}

void combosort(double *M, int a, int b){
    double shrink_factor = 1.3; //
    int gap = b - a;
     int size = b - a;
    int i;
    int swapped = 1;//флаг, выполянет ту же функцию, что и переменная
типа bool
    while( (gap > 1) || swapped ){
        if(gap > 1)
            gap = gap / shrink_factor;

        swapped = 0;

        i = a;

        while( (gap + i) < size ){
            if(M[i] > M[i + gap]){
                swap(&M[i], &M[i + gap]);
                swapped = 1;
            }
            ++i;
```

```
        }
    }
}

/* New array should be initialized */
void merge_arrays(double *array_old, double *array_new, unsigned int n,
int num, int chunk) {
     unsigned int i, m;
     unsigned int *arr_i = calloc(num, (sizeof(unsigned int)));
     unsigned int min;

     for (i = 0; i < n; i++) {
          min = 0;
          //заполняем элемент нового массива первым попавшимся
элементом из старого массива
          for (m = 0; m < num; m++) {
               if (arr_i[m] < chunk) {
                    array_new[i] = array_old[m*chunk + arr_i[m]];
                    min = m;
                    break;
               }
          }
          //здесь уже смотрим на части массива и делаем merge
          for (m = 0; m < num; m++) {
               if((m*chunk + arr_i[m] < n) && //если не вышли за границы
всего сортируемого массива
                    (arr_i[m] < chunk) && // если находимся в пределах
конкретной отсортированной части
                    array_old[m*chunk + arr_i[m]] < array_new[i]) {
//если элемент из старого массива <чем установленный элемент из нового
массива
                    array_new[i] = array_old[m*chunk + arr_i[m]];
                    min = m;
               }
          }
          arr_i[min]++;
     }
}

void printM(double M[], int size){
    for(int i = 0; i < size; ++i){
        printf("M[%d] = %f\n", i, M[i]);
    }
}
//сортировка:
//- разбиваем массив на num_thread частей и сортируем каждую из частей
отдельно
//- потом делаем merge этих частей с окончательным упорядочиванием
массива
void sort(double **M, int size){
     int num_thread = omp_get_num_procs();
     int num, a, b;
     //получаем "шаг" (количество элементов) в каждой из сортируемых
частей
     int curr_chunk = size % num_thread ? size / num_thread + 1 : size
/ num_thread;
     unsigned int i;
```

```
      double *array_new = malloc(sizeof(double) * size);

      #pragma omp parallel for default(none) shared(M, size, curr_chunk,
num_thread) private (i, num, a, b) num_threads(num_thread)
      for(i = 0; i < num_thread; i++){
            num = omp_get_thread_num();
            a = num * curr_chunk;
            b = num * curr_chunk - 1;
            combosort(*M, a < size - 1 ? a : size - 1, b < size - 1 ? b
: size - 1);
      }
      merge_arrays(*M, array_new, size, num_thread, curr_chunk);
      free(*M);
      *M = array_new;
}


//возвращаем первый ненулевой элемент массива,
// предполагаем, что массив отсортирован по возрастанию
double min_double(double M[], int size){
    for(int i = 0; i < size; i++){
        if(M[i] != 0){
            return M[i];
        }
    }
    return 0;
}

//Этап 2, таблица 1, вариант № 2
//Этап 2, таблица 2, вараинт № 7
//Этап 3, вариант № 5
//Этап 4, вариант № 2
int main(int argc, char* argv[]) {
    //печатаем варианты
    //print_variants();
    double T1, T2;
    int A = 7*7*11;
    int m2_end = 10*A;
    long time_ms, minimal_time_ms = -1;
    int N = atoi(argv[1]);
    int num_thread = atoi(argv[2]);
    //минимальный X этапа Reduce
    double X, minimal_X = RAND_MAX;
    double min_in_M2;
    srand(42);//одинаковый генератор случайных чисел

    //генерация массивов

    srand(42);
    unsigned int seedp;//одинаковый генератор случайных чисел

    int i;
     omp_set_nested(1);
     char flag = 1;
     int k = 0;

     int count_iter = 10;
```

```
      printf("N=%d\n", N);
      printf("Измерение времени:\n");
/*
#pragma omp parallel sections shared (k)
{
      #pragma omp section
      {
*/
           for(k = 0; k < count_iter; ++k){
                double *M1 = malloc(N * sizeof(double));
               double *M2 = malloc(N / 2 * sizeof(double));
               T1 = omp_get_wtime();
               seedp = 90;

               //Этап 1
      #pragma omp parallel for default(none) private(i, seedp) shared(M1,
N, A) num_threads(num_thread)
                for (i = 0; i < N; ++i) {
                   seedp = sqrt(i);
                   M1[i] = get_M1_el(A, &seedp);
               }

               //Этап Generate
      #pragma omp parallel for default(none) private(i, seedp) shared(M2,
A, m2_end, N) num_threads(num_thread)
               for (i = 0; i < N / 2; ++i) {
                   seedp = sqrt(i);
                   M2[i] = get_M2_el(A, m2_end, &seedp);
               }
      #pragma omp parallel for default(none) private(i) shared(M1, N)
num_threads(num_thread)
               //Этап Map
               for (i = 0; i < N; ++i) {
                   M1[i] = cosh(M1[i]) + 1;
               }
               for (i = 1; i < N / 2; ++i) {
                   M2[i] = M2[i - 1] + M2[i];
               }
               //вычисляем кубический корень
      #pragma omp parallel for default(none) private(i) shared(M2, N)
num_threads(num_thread)
               for (i = 0; i < N / 2; ++i) {
                   M2[i] = cbrt(M2[i] * M_PI);
               }
               //Этап 3 merge
      #pragma omp parallel for default(none) private(i) shared(M1, M2,
N) num_threads(num_thread)
               for (i = 0; i < N / 2; ++i) {
                   M2[i] = fmin(M1[i], M2[i]);
               }
               //Этап Sort
      //        combosort(M2, 0, N / 2);
                sort(&M2, N/2);
               //Этап Reduce
               //ищем сумму синусов
               X = 0;
                //printM(M2, N/2);
```

```
            min_in_M2 = min_double(M2, N / 2);
      #pragma omp parallel for reduction(+:X) num_threads(num_thread)
            for(i = 0; i < N / 2; ++i){
                if( ( (int) trunc( (M2[i] / min_in_M2) )) % 2 == 0){
                    X += sin(M2[i]);
                }
            }

            //определяем минимальную сумму синусов
            if(X < minimal_X)
                minimal_X = X;

            T2 = omp_get_wtime();
            time_ms = (int)((T2 - T1) * 1000);

             printf("time_ms[%d]=%ld\n", k, time_ms);

            if    ((minimal_time_ms    ==    -1)    ||    (time_ms    <
minimal_time_ms))
                minimal_time_ms = time_ms;
            free(M1);
            free(M2);
        }
        flag = 0;
/*
      }
      #pragma omp section
      {
           printf("Task is completed for:\n0  %%");
           while(flag) {
                printf("%c[2K\r%d    %%",    27,    (int)((double)k    /
(double)count_iter * 100));
                fflush(stdout);
                sleep(1);
           }
           printf("\n");
      }
}
*/
      printf("------Окончательные результаты---------\n");
    printf("Best X (sum sin): %f\n", X);
    printf("Best time (ms): %ld\n", minimal_time_ms);
      printf("-----------------------------------------------------------
\n");
    return 0;
}
```

Скрипты для автоматизации проведения экспериментов.

Скрипт для выполнения программы без параметра schedule

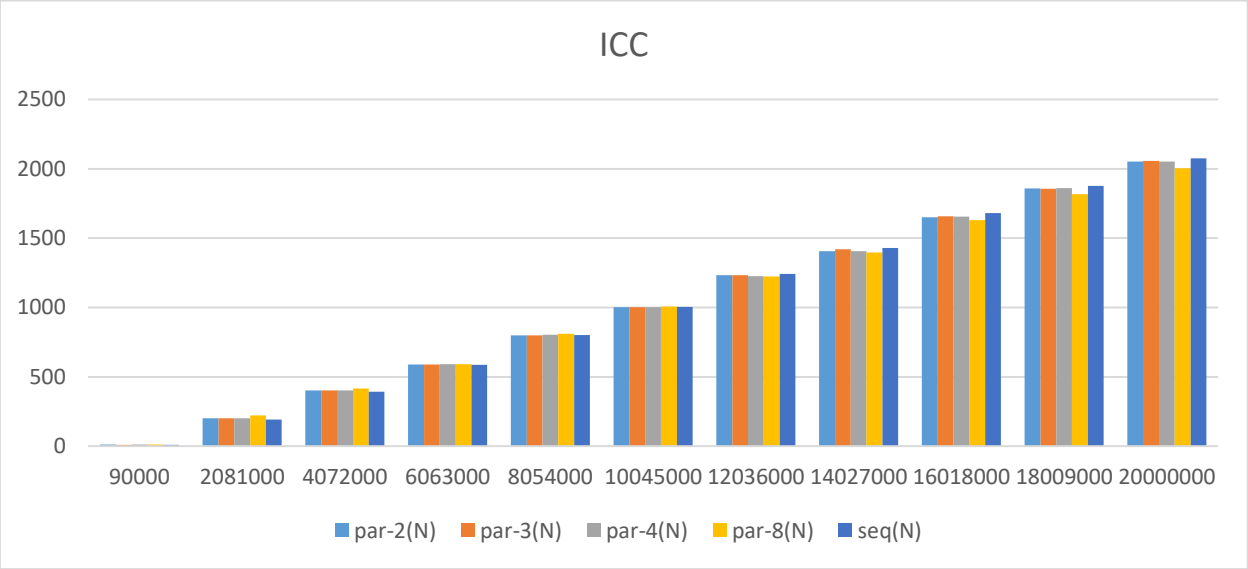
```
#!/bin/bash
icc -O3 -qopenmp main.c -lm -o lab4
echo "программа скомпилирована"
start=150000
finish=20000000
step=$(( (finish - start) / 10 ))
```

```
file=lab4-par1.txt

export OMP_SCHEDULE=auto

mkdir result
cd result

for i in `seq 1 1 8`
do
      file=lab4-par$i
      echo $file Начинается выполнение
      echo start, step, finish > $file.txt
      echo $start, $step, $finish >> $file.txt
      echo "" >> $file.txt
      for j in `seq $start $step $finish`
      do
            echo "" >> $file.txt
            ".././lab4" $j $i >> $file.txt
            echo $j Выполнено
      done
      echo $file Выполнено
      echo ""
done
```

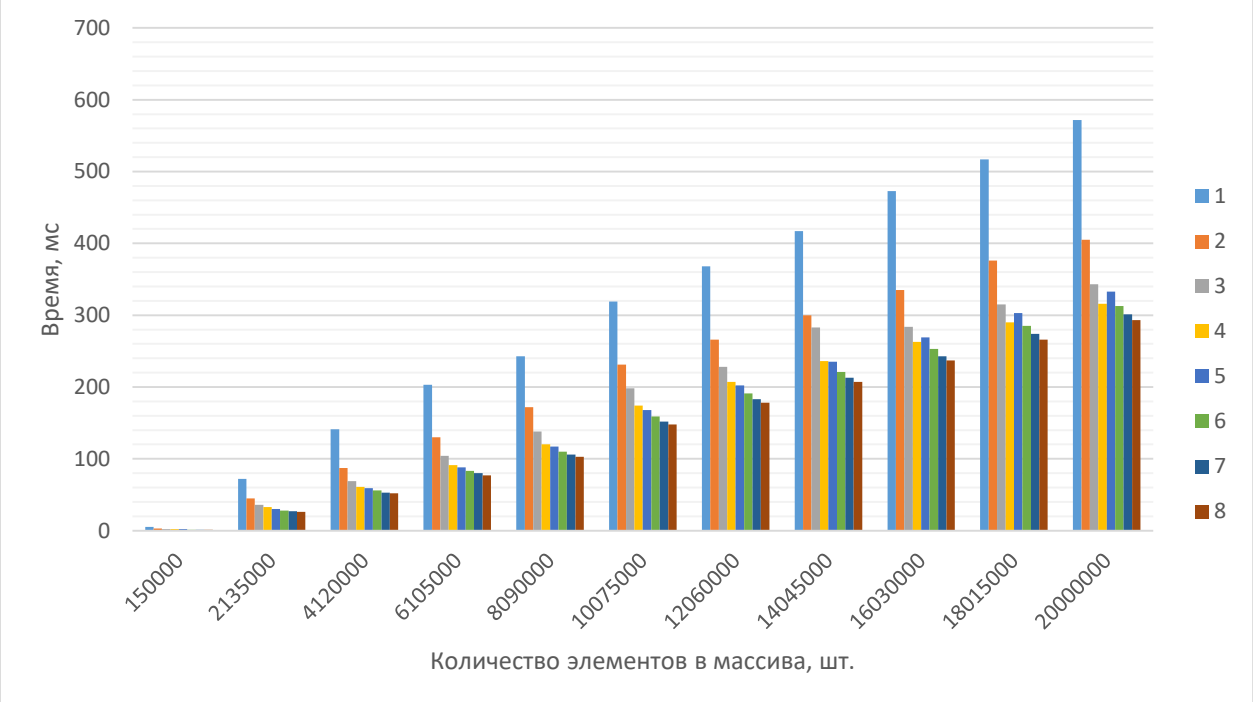
Таблицы значений функций и анализ результатов

Зависимость времени от N.

АВТОМАТИЧЕСКОЕ РАСПАРАЛЛЕЛИВАНИЕ

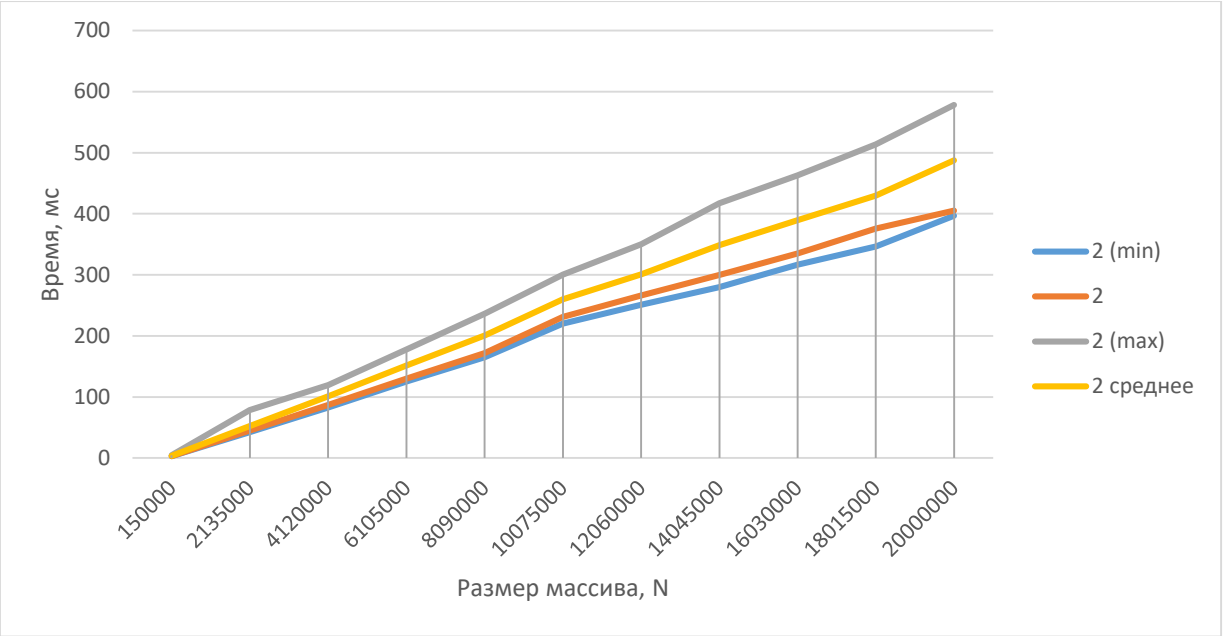


ТЕКУЩИЕ РЕЗУЛЬТАТЫ ПО МИНИМАЛЬНЫМ ЗАМЕРАМ

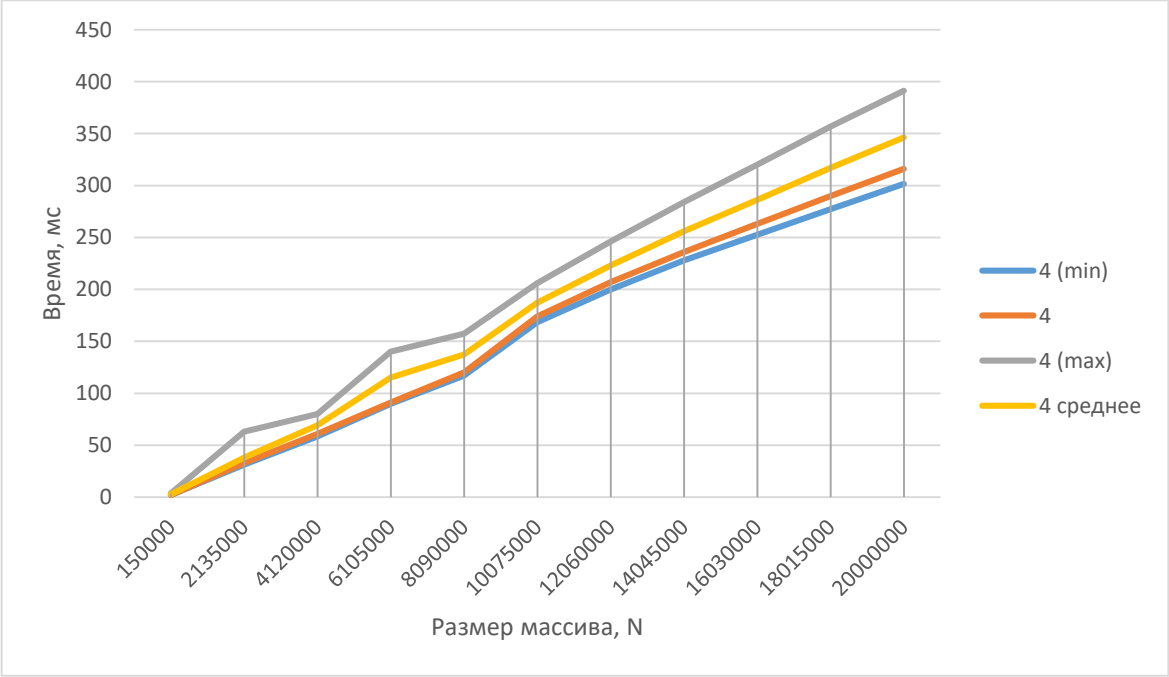


Анализируя результаты замеры по минимальному времени выполнения с результатами из первой лабораторной работы можно сделать вывод, что скорость работы программы выросла почти в 3 раза.

ДОВЕРИТЕЛЬНЫЙ ИНТЕРВАЛ ДЛЯ 2-Х ПОТОКОВ



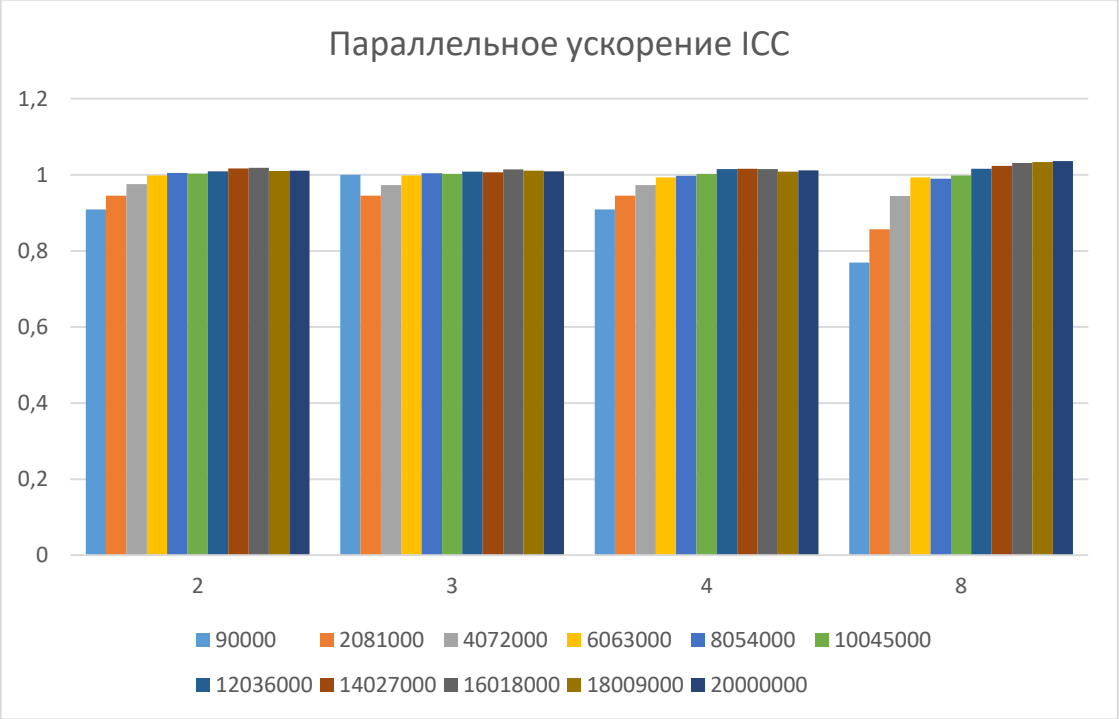
ДОВЕРИТЕЛЬНЫЙ ИНТЕРВАЛ ДЛЯ 4-Х ПОТОКОВ



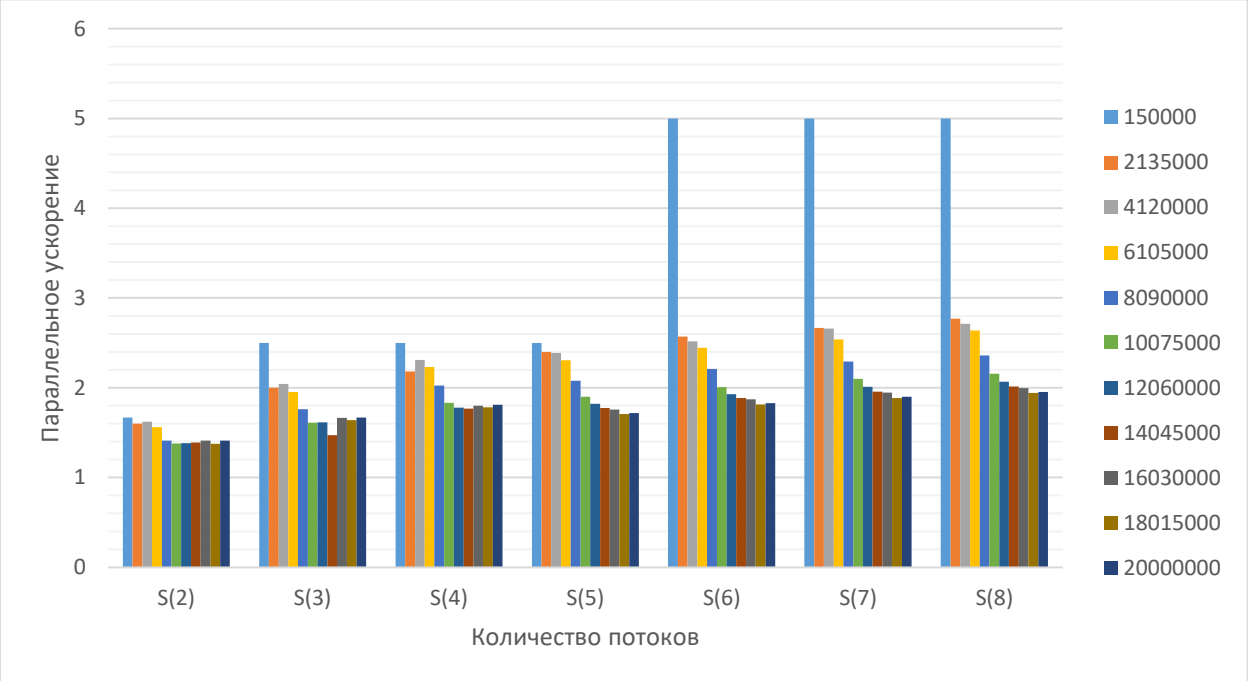
Анализируя графики зависимости времени от N, можно сделать вывод, что при любом N время работы программы меньше, чем при использовании средств автоматического распараллеливания. Это связано с тем, что с помощью OpenMP мы сами можем контролировать то, какие этапы выполнения программы необходимо распараллелить и берем ответственность за возможные неблагоприятные последствия принятого решения.

Параллельное ускорение

АВТОМАТИЧЕСКОЕ РАСПАРАЛЛЕЛИВАНИЕ



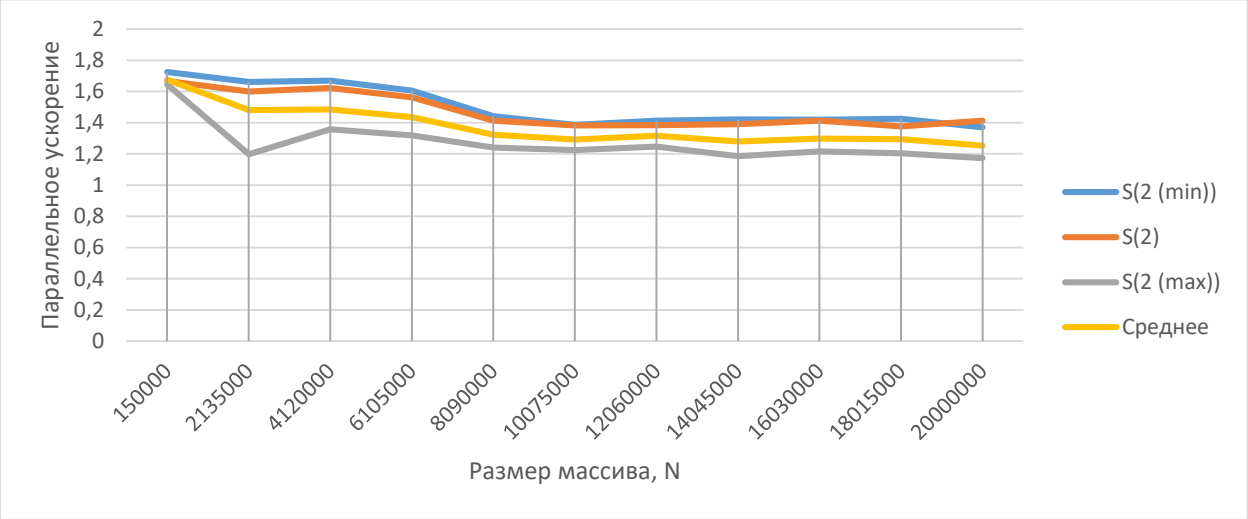
ТЕКУЩИЕ РЕЗУЛЬТАТЫ ПО МИНИМАЛЬНЫМ ЗАМЕРАМ ВРЕМЕНИ



Анализируя график параллельного ускорения можно сделать вывод, что скорость работы программы увеличилась в 2 раза при использовании 2-х и более потоков. Основной причиной стало то, что одна из самых трудоемких

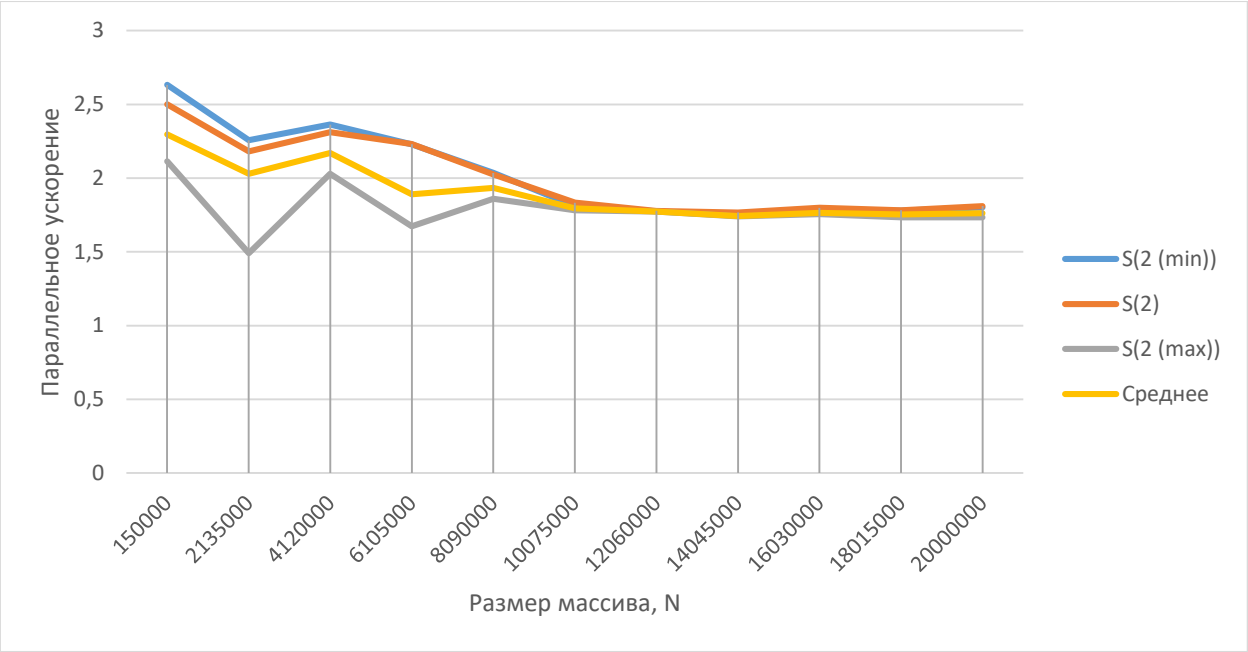
операций – сортировка была распараллелена. Но так как этап сортировки по заданию выполнялся на всех вычислительных ядрах, доступных на компьютере, не зависимо от указания количества потоков, то увеличение числа потоков на других этапах работы программы влияет на ее скорость выполнения в меньшей степени и не дает должного эффекта на параллельное ускорение.

ДОВЕРИТЕЛЬНЫЙ ИНТЕРВАЛ ДЛЯ ПАРАЛЛЕЛЬНОГО УСКОРЕНИЯ ДЛЯ 2-Х ПОТОКОВ.



Анализируя график можно сказать, что параллельное ускорение работы программы несколько снижается при возрастании объема вычислений, так как в системе мы используем только 2 вычислителя.

ДОВЕРИТЕЛЬНЫЙ ИНТЕРВАЛ ДЛЯ ПАРАЛЛЕЛЬНОГО УСКОРЕНИЯ ДЛЯ 4-Х ПОТОКОВ.



Анализируя график параллельного ускорения, построенного с использованием доверительного интервала, для 4-х потоков, можно сделать, что при увеличении объема вычислений все параметры начинают примерно совпадать (нижняя и верхняя границы доверительного интервала, среднее значение и минимальное время замеров).

Выводы

В результате проведения экспериментов можно сделать вывод, что использование OpenMP дает существенный эффект. В рамках выполнения лабораторной работы был распараллелен самый трудоемкий этап работы программы – сортировка. В результате время выполнения работы время работы программы уменьшилось в среднем в 6-7 раз. При этом при больших значениях N (размер массива) время не сильно отличалось при установке разного количества потоков. Данный эффект объясняется тем, что сортировка выполняется в количестве потоков, равном количеству вычислителей на ПК.