**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

Параллельные вычисления

|  |
| --- |
| Параллельные алгоритмы сортировки в OpenMP и их практическое применение |

Руководитель А.Г. Зотин

подпись, дата инициалы, фамилия

Обучающийся БПИ20-02, 201219047 Р.А. Сухачев

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2022 г.

# ЦЕЛЬ РАБОТЫ

Изучить различные алгоритмы сортировки и их параллельные варианты реализации. Познакомится с медианным фильтром для обработки изображений.

# порядок выполнения работы

1. Ознакомиться с теоретическими сведениями.

2. Ознакомиться с алгоритмами сортировки.

3. Выполнить задания 1–2.

4. Подготовить отчет по лабораторной работе.

5.  Защитить лабораторную работу перед преподавателем.

# постановка задачи

Вариант 20

Задание 3.1. Разработайте консольное приложение, реализующее сортировку одномерного массива разными алгоритмами (в последовательно и параллельной реализации). При помощи этого приложения выполните экспериментальное исследование на четырех наборах (20 000–50 000 эл.) по обработке разных типов данных (int, double) с разными реализациями (не менее 30 запусков для каждой реализации). В приложении необходимо реализовать следующие методы сортировки:

1. пузырька (классический алгоритм), только последовательный вариант;

2. пузырька (алгоритм Чет-нечетной перестановки);

3. Шелла;

4. быстрой сортировки (qsort).

Для каждой реализации функций сортировки массива выполните расчет времени и сформируйте сводные таблицы. При этом сортировку производить для одного и того же набора данных, т. е. исходные значения массива копируются в массив, предназначенный для сортировки. Заполнение массива осуществлять с учетом некоторой закономерности (придумать функцию генерации данных в зависимости от индекса). Результаты эксперимента оформите в виде таблицы.

Задание 3.2. Разработайте консольное приложение, реализующее выполнение медианной фильтрации изображения. В приложение должен загружаться набор из четырех файлов input\_X.bmp (где Х – номер файла) и формироваться выходные файлы output\_X\_alg.bmp, где alg означает имя используемого алгоритма обработки (или его номер). При реализации алгоритмов медианной фильтрации необходимо использовать не менее двух алгоритмов сортировки (на выбор студента взять из задания 3.1) и не менее двух вариантов распараллеливания (например, на основе распараллеливания циклов или создания параллельных секций).

При помощи разработанного приложения проведите экспериментальные исследования по обработке данных с разными реализациями (не менее 20 запусков для каждой реализации). При применении каждого алгоритма получить показатели для разных размеров окрестности Ksize×Ksize, для четырех изображений с разрешением от 1280×720 до 3840×21603. В рамках лабораторной работы Ksize = (RH×2+1) = (RW×2+1). Результаты экспериментального исследования оформить в виде таблицы.

# ХОД РАБОТЫ

Задание 3.1.

Код программы:

#include <iostream>

#include <fstream>

#include <omp.h>

#include <vector>

#include <Math.h>

#include <stdio.h>

using namespace std;

typedef double(\*TestFunctTemp2)(double\*&, int&);

template <typename T>

void fillArr(T\*& array, int& N) {

for (int i = 0; i < N; i++)

array[i] = pow(i, 2.0 / 3.0) \* sin(i / 2.0);

}

template <typename T>

void printArr(T\*& array, int& N) {

for (int i = 0; i < N; i++)

cout << array[i] << endl;

}

template <typename T>

void compare\_exchange(T& array1, T& array2) {

T temp;

if (array1 > array2) {

temp = array2;

array2 = array1;

array1 = temp;

}

}

template <typename T>

double bubbleSort(T\*& array, int& N) {

double time\_start = omp\_get\_wtime();

for (int i = 0; i < N; i++) {

for (int j = 0; j < N - i - 1; j++) {

compare\_exchange(array[j], array[j + 1]);

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double oddEvenSorting(T\*& array, int& N) {

double time\_start = omp\_get\_wtime();

int upper\_bound;

if (N % 2 == 0)

upper\_bound = N / 2 - 1;

else

upper\_bound = N / 2;

for (int i = 0; i < N; i++) {

if (i % 2 == 0)

for (int j = 0; j < N / 2; j++)

compare\_exchange(array[2 \* j], array[2 \* j + 1]);

else

for (int j = 0; j < upper\_bound; j++)

compare\_exchange(array[2 \* j + 1], array[2 \* j + 2]);

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double oddEvenSortingParallel(T\*& array, int& N) {

double time\_start = omp\_get\_wtime();

int upper\_bound;

if (N % 2 == 0)

upper\_bound = N / 2 - 1;

else

upper\_bound = N / 2;

for (int i = 0; i < N; i++) {

if (i % 2 == 0)

#pragma omp parallel for

for (int j = 0; j < N / 2; j++)

compare\_exchange(array[2 \* j], array[2 \* j + 1]);

else

#pragma omp parallel for

for (int j = 0; j < upper\_bound; j++)

compare\_exchange(array[2 \* j + 1], array[2 \* j + 2]);

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

void InsertSort(T\* arr, int i, int length, int half) {

T temp = 0;

int j = 0;

for (int f = half + i; f < length; f = f + half)

{

j = f;

while (j > i && arr[j - half] > arr[j])

{

temp = arr[j];

arr[j] = arr[j - half];

arr[j - half] = temp;

j = j - half;

}

}

}

template <typename T>

double shellSort(T\*& array, int& N) {

double time\_start = omp\_get\_wtime();

int i, step;

for (step = N / 2; step > 0; step /= 2)

for (i = 0; i < step; i++)

{

InsertSort(array, i, N, step);

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double shellSortParallel(T\*& array, int& N) {

double time\_start = omp\_get\_wtime();

int i, step;

for (step = N / 2; step > 0; step /= 2)

#pragma omp parallel for shared( array, N, step, i) default(none)

for (i = 0; i < step; i++)

{

InsertSort(array, i, N, step);

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double quickSort(T\* array, int N) {

double time\_start = omp\_get\_wtime();

long i = 0, j = N;

T temp, p;

p = array[N >> 1];

do {

while (array[i] < p) i++;

while (array[j] > p) j--;

if (i <= j) {

temp = array[i]; array[i] = array[j]; array[j] = temp;

i++; j--;

}

} while (i <= j);

if (j > 0) quickSort(array, j);

if (N > i) quickSort(array + i, N - i);

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double quickSortParallel(T\* array, int N) {

double time\_start = omp\_get\_wtime();

long i = 0, j = N;

T temp, p;

p = array[N >> 1];

do {

while (array[i] < p) i++;

while (array[j] > p) j--;

if (i <= j) {

temp = array[i]; array[i] = array[j]; array[j] = temp;

i++; j--;

}

} while (i <= j);

#pragma omp task shared(array)

if (j > 0) quickSortParallel(array, j);

#pragma omp task shared(array)

if (i < N) quickSortParallel(array + i, N - i);

#pragma omp taskwait

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

template <typename T>

double testBubbleSort(T\*& array, int& N) {

return bubbleSort(array, N);

}

template <typename T>

double testOddEvenSorting(T\*& array, int& N) {

return oddEvenSorting(array, N);

}

template <typename T>

double testShellSort(T\*& array, int& N) {

return shellSort(array, N);

}

template <typename T>

double testQuickSort(T\*& array, int& N) {

return quickSort(array, N);

}

template <typename T>

double testOddEvenSortingParallel(T\*& array, int& N) {

return oddEvenSortingParallel(array, N);

}

template <typename T>

double testShellSortParallel(T\*& array, int& N) {

return shellSortParallel(array, N);

}

template <typename T>

double testQuickSortParallel(T\*& array, int& N) {

return quickSortParallel(array, N);

}

double AvgTrustedInterval(double& avg, double\*& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

template <typename T>

double TestIter(void\* Funct, T\* array, int size, int iterations)

{

typedef double(\*TestFunctTempl)(T\*&, int&);

double curtime = 0, avgTime = 0, avgTimeT = 0, correctAVG = 0;;

double\* Times = new double[iterations];

cout << endl;

for (int i = 0; i < iterations; i++)

{

T\* arr = new T[size];

memcpy(arr, array, sizeof(int) \* size);

curtime = ((\*(TestFunctTempl)Funct)(arr, size)) \* 1000;

Times[i] = curtime;

avgTime += curtime;

cout << "+";

//printArr(arr, size);

delete[] arr;

}

cout << endl;

avgTime /= iterations;

cout << "AvgTime:" << avgTime << endl;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

cout << "AvgTimeTrusted:" << avgTimeT << endl;

return avgTimeT;

}

template <typename T>

void test\_functions(void\*\* Functions, string(&function\_names)[7])

{

int iters = 30;

int nd = 0;

double times[4][7][3];

T\* arr;

for (int size = 20000; size <= 50000; size += 10000)

{

for (int threads = 1; threads <= 4; threads++)

{

omp\_set\_num\_threads(threads);

//перебор алгоритмов по условиям

for (int alg = 0; alg <= 6; alg++)

{

arr = new T[size];

fillArr(arr, size);

if (threads == 1)

{

if (alg == 0 || alg == 1 || alg == 3 || alg == 5) {

times[nd][alg][0] = TestIter<T>(Functions[alg], arr, size, iters);

// iters - кол-во запусков алгоритма

times[nd][alg][1] = times[nd][alg][0];

times[nd][alg][2] = times[nd][alg][0];

delete[] arr;

}

}

else

{

if (alg != 0 && alg != 1 && alg != 3 && alg != 5)

{

times[nd][alg][threads - 2] = TestIter<T>(Functions[alg], arr, size, iters);

delete[] arr;

}

}

}

}

nd++;

}

ofstream fout("output.txt");

fout.imbue(locale("Russian"));

for (int ND = 0; ND < 4; ND++)

{

switch (ND)

{

case 0:

cout << "\n----------20000 elements of arrray----------" << endl;

break;

case 1:

cout << "\n----------30000 elements of arrray----------" << endl;

break;

case 2:

cout << "\n----------40000 elements of arrray----------" << endl;

break;

case 3:

cout << "\n----------50000 elements of arrray----------" << endl;

break;

default:

break;

}

for (int alg = 0; alg <= 6; alg++)

{

for (int threads = 1; threads <= 4; threads++)

{

cout << "Threads " << threads << " --------------" << endl;

if (threads == 1)

{

if (alg == 0 || alg == 1 || alg == 3 || alg == 5) {

cout << function\_names[alg] << "\t" << times[ND][alg][0] << " ms." << endl;

fout << times[ND][alg][0] << endl;

}

}

else

{

if (alg != 0 && alg != 1 && alg != 3 && alg != 5)

{

cout << function\_names[alg] << "\t" << times[ND][alg][threads - 2] << " ms." << endl;

fout << times[ND][alg][threads - 2] << endl;

}

}

}

}

}

fout.close();

}

int main() {

void\*\* FunctionsINT = new void\* [7] { testBubbleSort<int>, testOddEvenSorting<int>, testOddEvenSortingParallel<int>,

testShellSort<int>, testShellSortParallel<int>, testQuickSort<int>, testQuickSortParallel<int> };

void\*\* FunctionsDOUBLE = new void\* [7] { testBubbleSort<double>, testOddEvenSorting<double>, testOddEvenSortingParallel<double>,

testShellSort<double>, testShellSortParallel<double>, testQuickSort<double>, testQuickSortParallel<double> };

string function\_names[7]{ "Bubble sort(consistent)", "OddEven sort(consistent)", "OddEven sort(parallel)", "Shell sort(consistent)", "Shell sort(parallel)",

"Quick sort(consistent)", "Quick sort(parallel)" };

cout << "---------------------------TYPE INT---------------------------" << endl;

test\_functions<int>(FunctionsINT, function\_names);

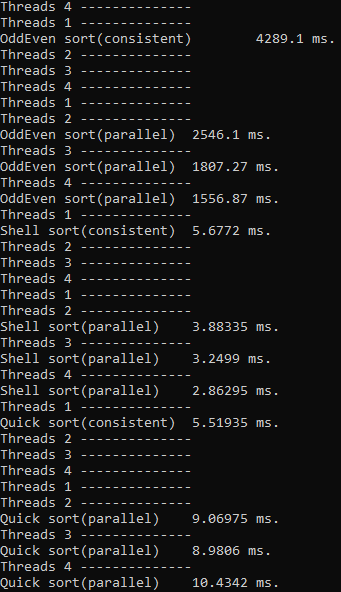
cout << "---------------------------TYPE DOUBLE---------------------------" << endl;

test\_functions<double> (FunctionsDOUBLE, function\_names);

return 0;

}

Демонстрация работы программы представлена на рисунке 1.

Рисунок 1 – работа программы

Задание 3.2.

Код программы:

#include <iostream>

#include <fstream>

#include <omp.h>

#include <cstring>

#include <string>

#include "BMPFileRW.h"

using namespace std;

#pragma warning(disable : 4996)

typedef double(\*TestFunctTemp1)(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize);

void InsertSort(int\* arr, int i, int length, int half) {

int temp = 0;

int j = 0;

for (int f = half + i; f < length; f = f + half)

{

j = f;

while (j > i && arr[j - half] > arr[j])

{

temp = arr[j];

arr[j] = arr[j - half];

arr[j - half] = temp;

j = j - half;

}

}

}

double shellSort(int\*& array, int& N) {

double time\_start = omp\_get\_wtime();

int i, step;

for (step = N / 2; step > 0; step /= 2)

for (i = 0; i < step; i++)

{

InsertSort(array, i, N, step);

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

void quickSort(int\* array, int N) {

long i = 0, j = N;

int temp, p;

p = array[N >> 1];

do {

while (array[i] < p) i++;

while (array[j] > p) j--;

if (i <= j) {

temp = array[i]; array[i] = array[j]; array[j] = temp;

i++; j--;

}

} while (i <= j);

if (j > 0) quickSort(array, j);

if (N > i) quickSort(array + i, N - i);

}

void fillMEDMAS(int\* MEDMAS\_R, int\* MEDMAS\_G, int\* MEDMAS\_B, RGBTRIPLE\*\* rgb\_in, int height, int width, int y, int x, int RH, int RW) {

int masind = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

MEDMAS\_R[masind] = rgb\_in[ky][kx].rgbtRed;

MEDMAS\_G[masind] = rgb\_in[ky][kx].rgbtGreen;

MEDMAS\_B[masind] = rgb\_in[ky][kx].rgbtBlue;

masind++;

}

}

}

double medianShellFilter(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianShellFilterParallelFor(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

#pragma omp parallel for

for (int y = 0; y < height; ++y)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianShellFilterSections(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

int p;

#pragma omp parallel

{

p = omp\_get\_num\_threads();

}

int iteration1 = height / p;

int iteration2 = height \* 2 / p;

int iteration3 = height \* 3 / p;

#pragma omp parallel sections

{

#pragma omp section

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = 0; y < iteration1; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

#pragma omp section

{

if (p > 1)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration1; y < iteration2; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

#pragma omp section

{

if (p > 2)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration2; y < iteration3; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

#pragma omp section

{

if (p > 3)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration3; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

shellSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

shellSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

shellSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianQuickFilter(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianQuickFilterParallelFor(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

#pragma omp parallel for

for (int y = 0; y < height; ++y)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianQuickFilterSections(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

int p;

#pragma omp parallel

{

p = omp\_get\_num\_threads();

}

int iteration1 = height / p;

int iteration2 = height \* 2 / p;

int iteration3 = height \* 3 / p;

#pragma omp parallel sections

{

#pragma omp section

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = 0; y < iteration1; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

#pragma omp section

{

if (p > 1)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration1; y < iteration2; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

#pragma omp section

{

if (p > 2)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration2; y < iteration3; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

#pragma omp section

{

if (p > 3)

{

int\* MEDMAS\_R = new int[size];

int\* MEDMAS\_G = new int[size];

int\* MEDMAS\_B = new int[size];

for (int y = iteration3; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

fillMEDMAS(MEDMAS\_R, MEDMAS\_G, MEDMAS\_B, rgb\_in, height, width, y, x, RH, RW);

quickSort(MEDMAS\_R, size);

rgb\_out[y][x].rgbtRed = MEDMAS\_R[size / 2];

quickSort(MEDMAS\_G, size);

rgb\_out[y][x].rgbtGreen = MEDMAS\_G[size / 2];

quickSort(MEDMAS\_B, size);

rgb\_out[y][x].rgbtBlue = MEDMAS\_B[size / 2];

}

}

delete[] MEDMAS\_R;

delete[] MEDMAS\_G;

delete[] MEDMAS\_B;

}

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double testMedianShellFilter(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianShellFilter(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianQuickFilter(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianQuickFilter(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianShellFilterSections(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianShellFilterSections(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianQuickFilterSections(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianQuickFilterSections(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianShellFilterParallelFor(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianShellFilterParallelFor(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianQuickFilterParallelFor(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianQuickFilterParallelFor(rgb\_in, rgb\_out, height, width, ksize);

}

char\* inBMP(int i) {

string str;

str = "c:\\users\\whoami\\source\\repos\\parallel computing\\lab\_3\\input\_X.bmp";

char\* cstr;

switch (i)

{

case 1:

str[60] = '1';

break;

case 2:

str[60] = '2';

break;

case 3:

str[60] = '3';

break;

case 4:

str[60] = '4';

break;

default:

break;

}

cstr = new char[str.length() + 1];

strcpy(cstr, str.c\_str());

return cstr;

}

char\* outBMP(int i, int alg) {

string str;

str = "C:\\Users\\whoami\\source\\repos\\Parallel Computing\\lab\_3\\output\_X\_Y.bmp";

char\* cstr;

switch (i)

{

case 1:

str[61] = '1';

break;

case 2:

str[61] = '2';

break;

case 3:

str[61] = '3';

break;

case 4:

str[61] = '4';

break;

default:

break;

}

switch (alg)

{

case 0:

str[63] = '1';

break;

case 1:

str[63] = '2';

break;

case 2:

str[63] = '3';

break;

case 3:

str[63] = '4';

break;

case 4:

str[63] = '5';

break;

case 5:

str[63] = '6';

break;

default:

break;

}

cstr = new char[str.length() + 1];

strcpy(cstr, str.c\_str());

cout << cstr << endl;

return cstr;

}

double AvgTrustedInterval(double& avg, double\*& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

double TestIter(void\* Funct, RGBTRIPLE\*\* rgb\_in, int Height, int Width, int ksize, int iterations, int i, int alg)

{

double curtime = 0, avgTime = 0, avgTimeT = 0, correctAVG = 0;;

double\* Times = new double[iterations];

cout << endl;

RGBTRIPLE\*\* rgb\_out;

rgb\_out = new RGBTRIPLE \* [Height];

rgb\_out[0] = new RGBTRIPLE[Width \* Height];

for (int j = 1; j < Height; j++)

{

rgb\_out[j] = &rgb\_out[0][Width \* j];

}

for (int j = 0; j < iterations; j++)

{

curtime = ((\*(TestFunctTemp1)Funct)(rgb\_in, rgb\_out, Height, Width, ksize)) \* 1000;

Times[j] = curtime;

avgTime += curtime;

cout << "+";

}

cout << endl;

avgTime /= iterations;

cout << "AvgTime:" << avgTime << endl;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

cout << "AvgTimeTrusted:" << avgTimeT << endl;

char\* cstr = outBMP(i, alg);

BMPWrite(rgb\_out, Width, Height, cstr);

delete[] rgb\_out[0];

delete[] rgb\_out;

delete[] cstr;

return avgTimeT;

}

void test\_functions(void\*\* Functions, string(&function\_names)[6])

{

RGBTRIPLE\*\* rgb\_in;

BITMAPFILEHEADER header;

BITMAPINFOHEADER bmiHeader;

int imWidth = 0, imHeight = 0;

int iters = 20;

int nd = 0;

double times[4][6][3][3];

for (int i = 1; i < 5; i++)

{

char\* cstr = inBMP(i);

BMPRead(rgb\_in, header, bmiHeader, cstr);

imWidth = bmiHeader.biWidth;

imHeight = bmiHeader.biHeight;

for (int threads = 1; threads <= 4; threads++)

{

omp\_set\_num\_threads(threads);

for (int alg = 0; alg < 6; alg++)

{

int ksize = 7;

for (int j = 0; j < 3; j++)

{

if (threads == 1)

{

if (alg == 0 || alg == 3) {

times[nd][alg][j][0] = TestIter(Functions[alg], rgb\_in, imHeight, imWidth, ksize, iters, i, alg);

times[nd][alg][j][1] = times[nd][alg][j][0];

times[nd][alg][j][2] = times[nd][alg][j][0];

}

}

else

{

if (alg != 0 && alg != 3)

{

times[nd][alg][j][threads - 2] = TestIter(Functions[alg], rgb\_in, imHeight, imWidth, ksize, iters, i, alg);

}

}

ksize += 4;

}

}

}

delete[] cstr;

nd++;

}

ofstream fout("output.txt");

fout.imbue(locale("Russian"));

for (int ND = 0; ND < 1; ND++)

{

switch (ND)

{

case 0:

cout << "\n----------1280\*720----------" << endl;

break;

case 1:

cout << "\n----------1920\*1200----------" << endl;

break;

case 2:

cout << "\n----------2560\*1440----------" << endl;

break;

case 3:

cout << "\n----------3840\*2160----------" << endl;

break;

default:

break;

}

for (int alg = 0; alg < 6; alg++)

{

for (int threads = 1; threads <= 4; threads++)

{

cout << "Thread " << threads << " --------------" << endl;

for (int j = 0; j < 3; j++)

{

cout << "Ksize = " << j << " --------------" << endl;

if (threads == 1)

{

if (alg == 0 || alg == 3) {

cout << function\_names[alg] << "\t" << times[ND][alg][j][0] << " ms." << endl;

fout << times[ND][alg][j][0] << endl;

}

}

else

{

if (alg != 0 && alg != 3)

{

cout << function\_names[alg] << "\t" << times[ND][alg][j][threads - 2] << " ms." << endl;

fout << times[ND][alg][j][threads - 2] << endl;

}

}

}

}

}

}

fout.close();

}

int main()

{

void\*\* FunctionsINT = new void\* [6]{ testMedianShellFilter, testMedianShellFilterSections, testMedianShellFilterParallelFor,

testMedianQuickFilter, testMedianQuickFilterSections, testMedianQuickFilterParallelFor };

string function\_names[6]{ "Median Filter(shell sort)", "Median Filter(shell sort sections)",

"Median Filter(shell sort parallel for)", "Median Filter(quick sort)", "Median Filter(quick sort sections)",

"Median Filter(quick sort parallel for)" };

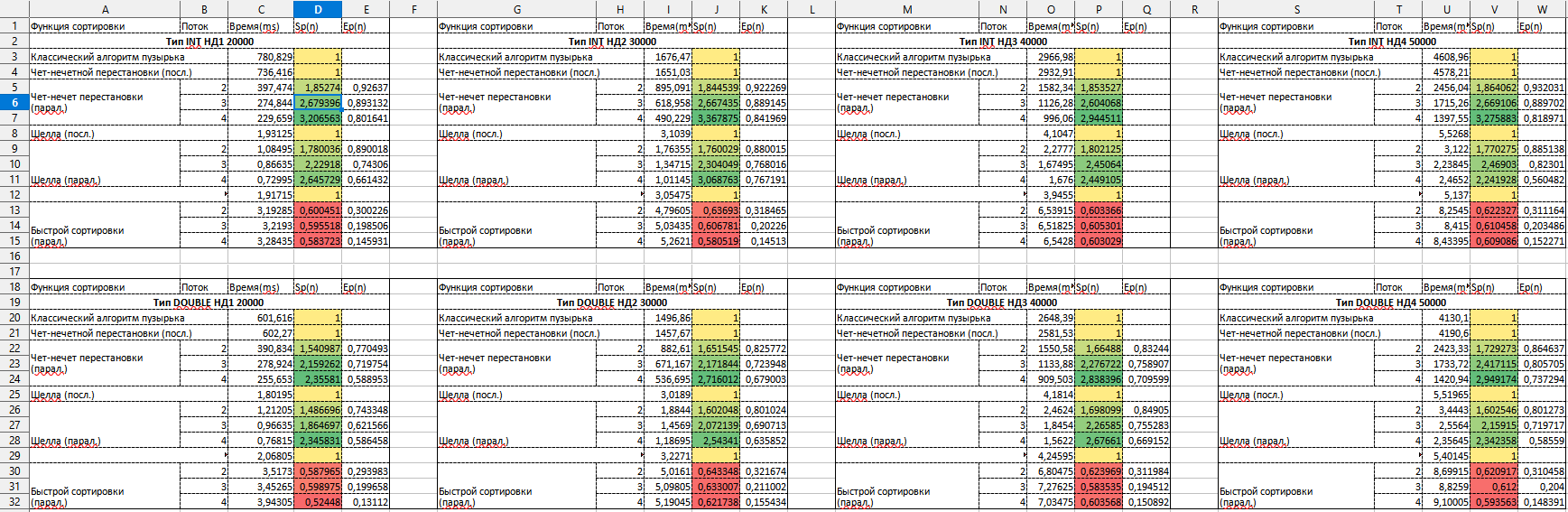
test\_functions(FunctionsINT, function\_names);

return 0;

}

Демонстрация работы программы представлена на рисунке 1.

Таблица 1 – результаты замеров времени работы первого задания с разными наборами функций



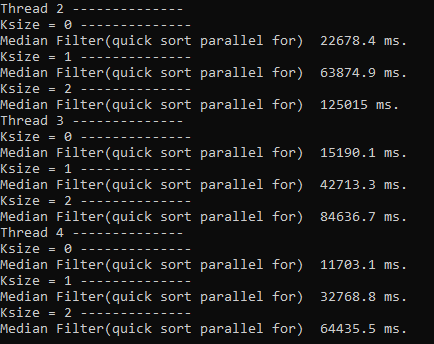
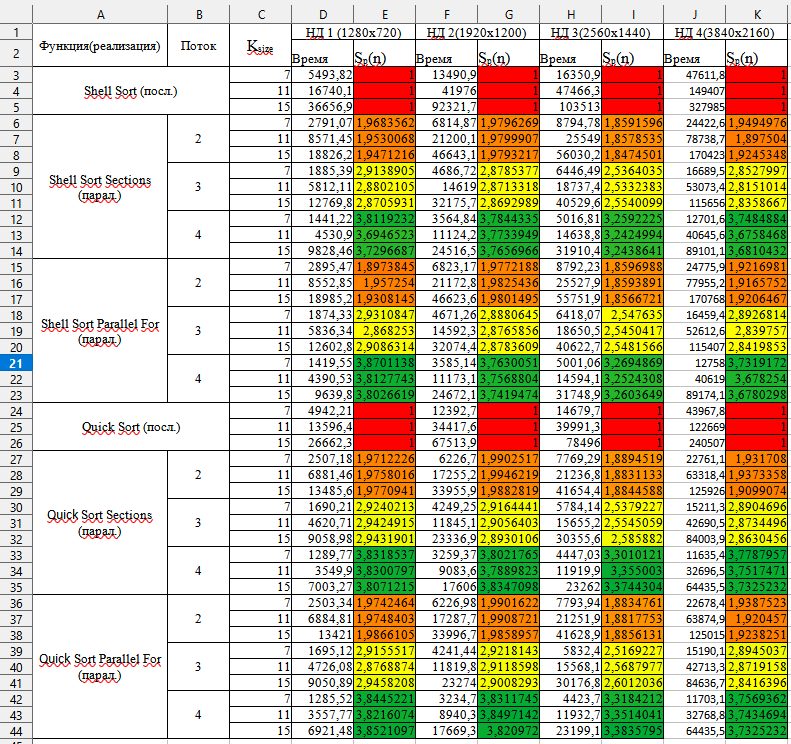
Рисунок 2 – работа программы

Таблица 2 – результаты замеров времени работы второго задания с разными наборами функций



Вывод: в ходе выполнения работы были получены замеры времени выполнения работы функций в последовательном и многопоточном режимах. В первом задании работа функций в параллельном режиме с использованием алгоритма Odd-Even Sort имеет наибольшее ускорение. Во втором задании работа функций в параллельном режиме с использованием алгоритмов Shell Sort Sections, Shell Sort Parallel For, Quick Sort Sections, Quick Sort Parallel For на 4 потоках имеет наибольшее ускорение.

# ВЫВОДЫ

Были изучены различные алгоритмы сортировки и их параллельные варианты реализации, был использован медианный фильтр для обработки изображений.