Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования «Кубанский государственный университет»

Кафедра вычислительных технологий

**ОТЧЕТ**

о выполнении лабораторной работы № 4

по дисциплине «Теория параллельных алгоритмов»

Выполнил:

Корнилов К.А.

Проверил: преподаватель

Нигодин Е. А.

Краснодар

2024

**Тема работы:** Сортировка методом пузырька, сортировка Шелла, быстрая сортировка.

**Цель:** Распараллеливание процесса сортировки массива с помощью методов пузырька, Шелла и быстрой сортировки с использованием технологии OpenMP.

**Задание:**

1. Построить последовательные алгоритмы перечисленных методов. Построить графы алгоритмов. Исследовать зависимости и возможности распараллеливания.
2. Написать параллельную реализацию алгоритмов.
3. Составить таблицу, отражающая сравнительное время выполнения последовательного и параллельных алгоритмов для разных массивов на 2-х и 4-х ядрах.

**Ход работы:**

1. Был составлен и написан на языке С++ последовательный алгоритм сортировки массива с помощью метода пузырька.

Входными данными алгоритма выступают: массив чисел размера n.

Результатом выступает отсортированный изначальный массив размера n.

Данная сортировка достигается за счет сравнения значений соседних элементов и дальнейшей их перестановки в случае нарушения порядка. Для обеспечения возможности распараллеливания данного алгоритма на каждой итерации происходит сравнения только пар элементов, начинающихся на нечетных индексах, если номер итерации нечетный, и на четных индексах, если номер итерации четный. В таком случае сравнения элементов пар будут независимы.

Тогда последовательный алгоритм сортировки массива с помощью метода пузырька изображен на рисунке 1 для частного случая (5 элементов):

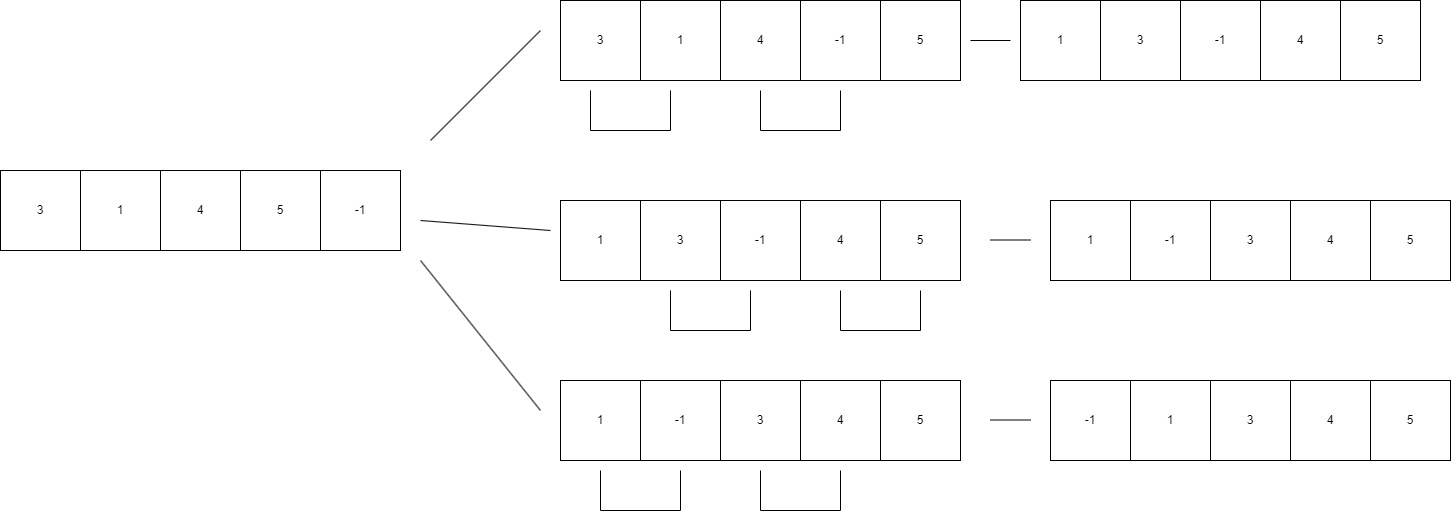


Рисунок 1 – Алгоритм сортировки массива с помощью метода пузырька

Код реализованного алгоритма имеет следующий вид:

double\* bubble(double\* ar, int N) {

double\* res = new double[N];

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

res[i] = ar[i];

}

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

{

int first = i % 2;

for (int j = first; j < N - 1; j += 2)

{

if (res[j] > res[j + 1])

{

double prom = res[j];

res[j] = res[j + 1];

res[j + 1] = prom;

}

}

}

return res;

}

Согласно схеме алгоритма сравнивания пар элементов объектов происходят независимо друг от друга. Тогда данные сравнения можно на каждой итерации работы выполнять паралелльно. Схема алгоритма параллельного алгоритма сортировки массива с помощью метода пузырька представлена на рисунке 2.

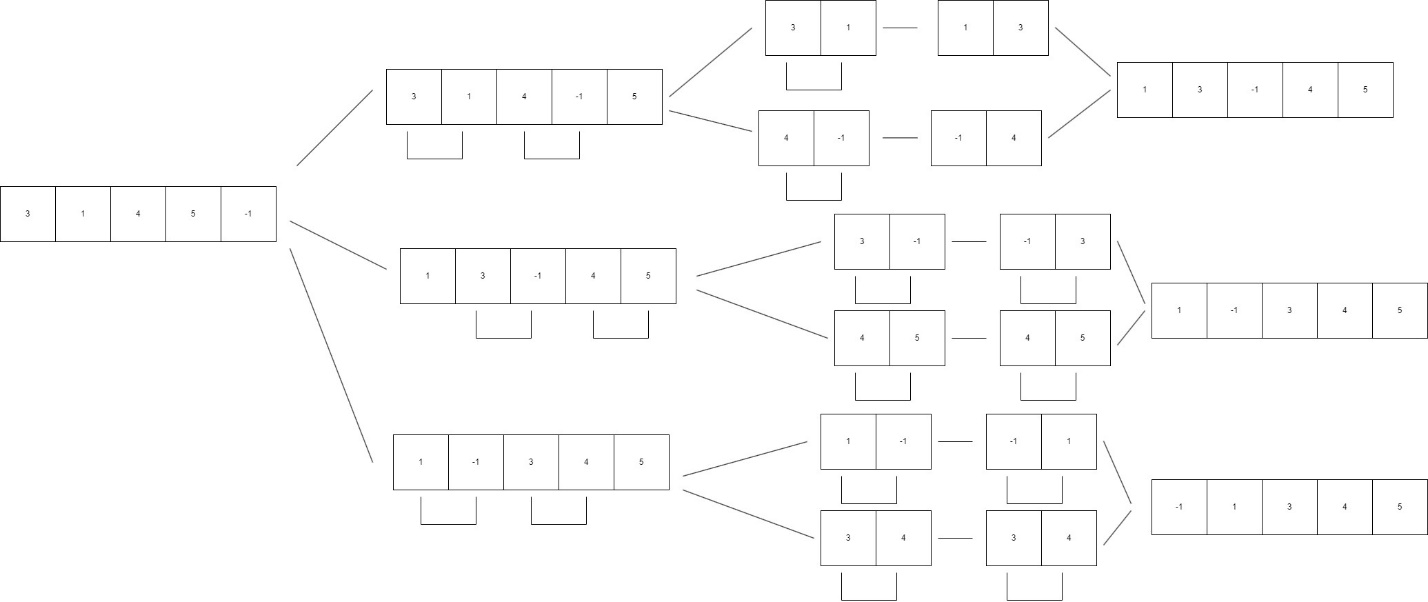


Рисунок 2 – Схема параллельного алгоритма сортировки массива с помощью метода пузырька

1. Была написана параллельная реализация алгоритма сортировки массива с помощью метода пузырька на языке С++ с использованием технологии OpenMP.

Код первого варианта параллельной реализации численного интегрирования по формуле Симпсона:

double\* bubble\_parallel(double\* ar, int N,int numthreads) {

double\* res = new double[N];

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

res[i] = ar[i];

}

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

{

int first = i % 2;

#pragma omp parallel for collapse(1) num\_threads(numthreads) shared(res,first,N)

for (int j = first; j < N - 1; j += 2)

{

if (res[j] > res[j + 1])

{

double prom = res[j];

res[j] = res[j + 1];

res[j + 1] = prom;

}

}

}

return res;

}

1. Был составлен и написан на языке С++ последовательный алгоритм сортировки массива с помощью метода Шелла.

Входными данными алгоритма выступают: массив чисел размера n.

Результатом выступает отсортированный изначальный массив размера n.

Данная сортировка достигается за счет сравнения на каждой итерации работы значений элементов, расположенных на расстоянии d. Если нарушается условие порядка, то элемент – нарушитель проталкивается вперед по элементам с шагом d, пока на очередном шаге не выполнится условие порядка. Когда алгоритм проходится по всем элементам, размер шага d уменьшается и процесс начинается заново для нового значения d.

Тогда последовательный алгоритм сортировки массива с помощью метода Шелла изображен на рисунке 3 для частного случая (5 элементов):

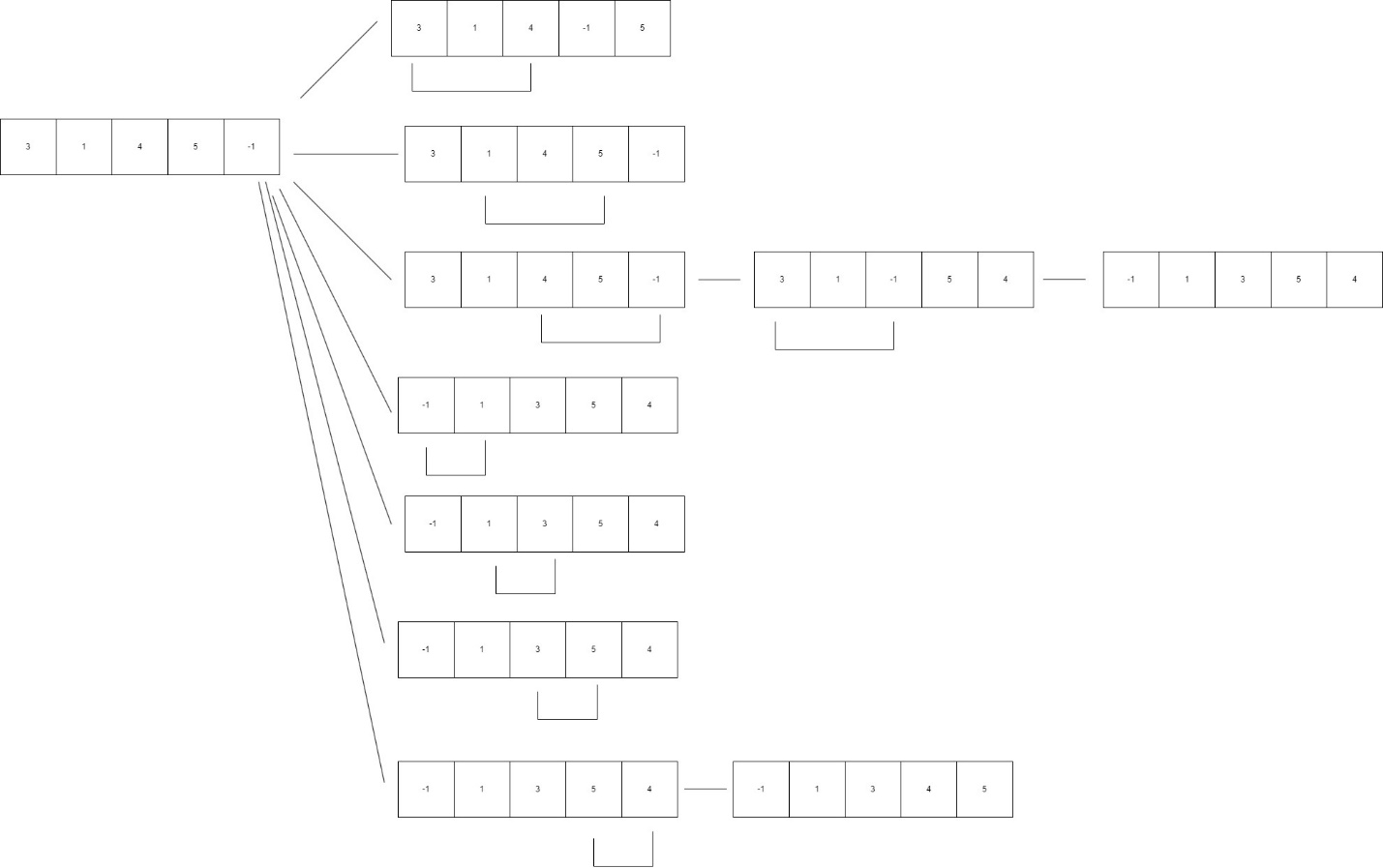


Рисунок 3 – Алгоритм сортировки массива с помощью метода Шелла

Код реализованного алгоритма имеет следующий вид:

double\*ShellSort(double\*arr,int N)

{

int i, j, step;

int tmp;

double\* res = new double[N];

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

res[i] = arr[i];

}

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

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

{

tmp = res[i];

for (j = i; j >= step; j -= step)

{

if (tmp < res[j - step])

res[j] = res[j - step];

else

break;

}

res[j] = tmp;

}

return res;

}

Согласно схеме алгоритма каждый элемент, с индексом меньше d, начинает независимую цепочку сравнений и перемещений элементов. Тогда данные цепочки могут быть выполнены параллельно. Схема алгоритма параллельного алгоритма сортировки массива с помощью метода Шелла представлена на рисунке 2.

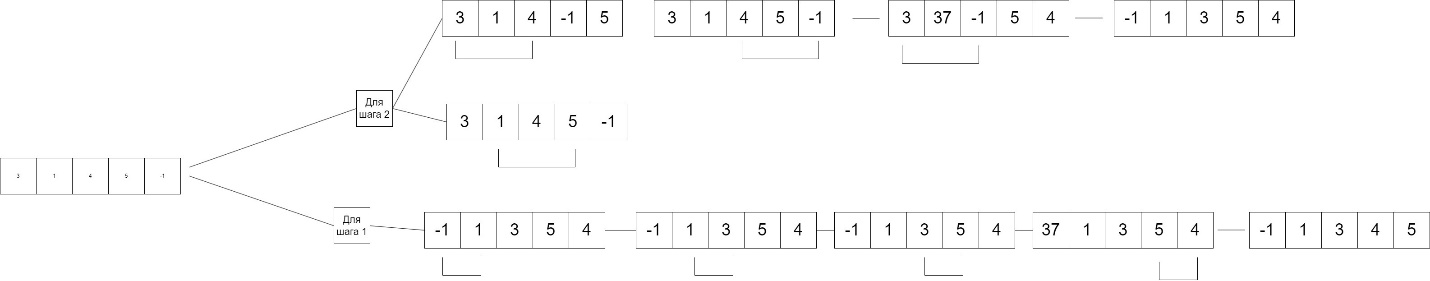


Рисунок 4 – Схема параллельного алгоритма сортировки массива с помощью метода Шелла

1. Была написана параллельная реализация алгоритма сортировки массива с помощью метода Шелла на языке С++ с использованием технологии OpenMP.

Код реализации алгоритма:

double\* ShellSort\_parallel(double\* arr, int N, int numthreads)

{

int i, j, step;

int tmp;

double\* res = new double[N];

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

res[i] = arr[i];

}

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

#pragma omp parallel for collapse(1) num\_threads(numthreads),shared(res,step,N)

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

{

for (int pr = i; pr < N; pr += step) {

tmp = res[pr];

for (j = pr; j >= step; j -= step)

{

if (tmp < res[j - step])

res[j] = res[j - step];

else

break;

}

res[j] = tmp;

}

}

return res;

}

1. Был составлен и написан на языке С++ последовательный алгоритм сортировки массива с помощью метода быстрой сортировки.

Входными данными алгоритма выступают: массив чисел размера n.

Результатом выступает отсортированный изначальный массив размера n.

На каждой итерации выбирается опорный элемент, являющийся либо массива. Затем все элементы, которые больше или равны опорного, перемещаются в правую от опорного элемента часть массива. Все элементы, которые меньше опорного, перемещаются в левую от опорного элемента часть массива. Процесс длится пока индексы, идущие сначала и с конца, не столкнутся. Затем процесс повторяется отдельно для правой и левой частей массива, определяемой точкой столкновения индексов на предыдущем шаге.

Тогда последовательный алгоритм сортировки массива с помощью метода быстрой сортировки изображен на рисунке 5 для частного случая (5 элементов):

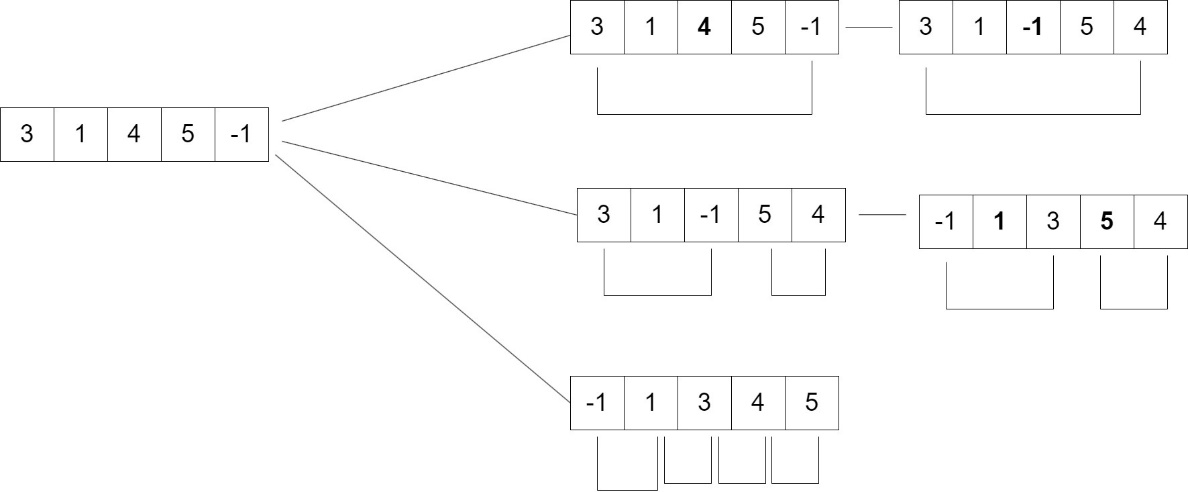


Рисунок 5 – Алгоритм сортировки массива с помощью метода быстрой сортировки

Код реализованного алгоритма имеет следующий вид:

void qsortRecursive(double\*arr, int size){

long i = 0;

int j = size - 1;

int mid = arr[size / 2];

do

{

while (arr[i] < mid)

i++;

while (arr[j] > mid)

j--;

cout << i << " " << j << endl;

if (i <= j)

{

swap(arr[i], arr[j]);

i++;

j--;

}

cout << i << " " << j << " "<<mid << endl;

} while (i <= j);

if (j > 0)

qsortRecursive(arr, j + 1);

if (size > i)

qsortRecursive(arr + i, size - i);

}

Согласно схеме алгоритма после сортировки относительно опорного элемента происходит разделение на два процесса сортировки подмассивов. Данные процессы могут быть выполнены независимо друг от друга. Схема параллельного алгоритма быстрой сортировки представлена на рисунке 6.

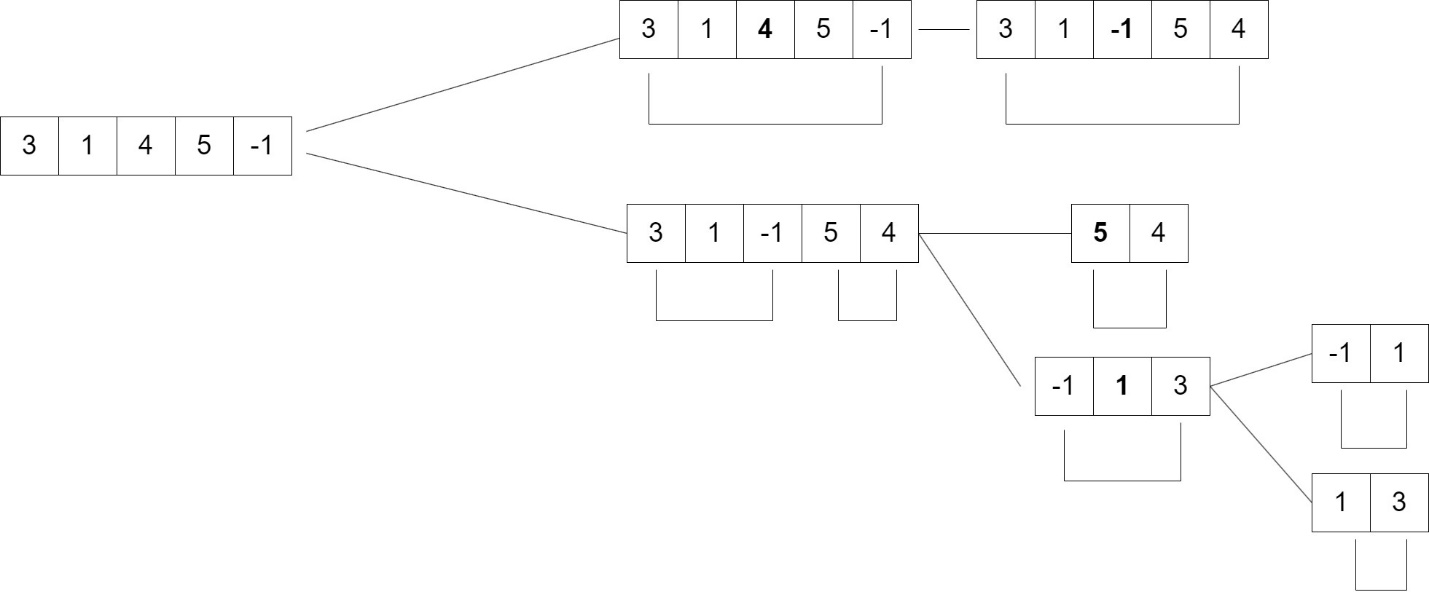


Рисунок 6 – Схема параллельного алгоритма сортировки массива с помощью метода быстрой сортировки

1. Была написана параллельная реализация алгоритма сортировки массива с помощью метода быстрой сортировке на языке С++ с использованием технологии OpenMP.

Код параллельной реализации быстрой сортировки:

void swap(double\* a, double\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

int partition(double\*arr, int start, int end)

{

double pivot = arr[end];

int i = (start - 1);

for (int j = start; j <= end - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[end]);

return (i + 1);

}

void quicksort(double\* arr, int start, int end)

{

int index;

if (start < end) {

index = partition(arr, start, end);

#pragma omp parallel sections

{

#pragma omp section

{

quicksort(arr, start, index - 1);

}

#pragma omp section

{

quicksort(arr, index + 1, end);

}

}

}

}

1. Используя написанные программы, была проведена оценка времени работы алгоритмов для различного размера массивов и для различного количество ядер. Результаты выполненных оценок представлены в виде графика времени выполнения на рисунке 4.

По оси Ox изображены количества элементов массива. По оси Oy изображено время работы алгоритма.

Голубым цветом изображено время выполнения для последовательного алгоритма пузырька.

Прозрачно – голубым изображено время выполнения для параллельного алгоритма пузырька на 2 ядрах.

Синим изображено время выполнения для параллельного алгоритма пузырька на 3 ядрах.

Темно – синим изображено время выполнения для параллельного алгоритма пузырька на 4 ядрах.

Зеленым цветом изображено время выполнения для последовательного алгоритма Шела.

Прозрачно – зеленым изображено время выполнения для параллельного алгоритма Шела на 2 ядрах.

Светло – зеленым изображено время выполнения для параллельного алгоритма Шела на 3 ядрах.

Темно – зеленым изображено время выполнения для параллельного алгоритма Шела на 4 ядрах.

Оранжевым цветом изображено время выполнения для последовательного алгоритма быстрой сортировки.

Прозрачно – оранжевым изображено время выполнения для параллельного алгоритма быстрой сортировки на 2 ядрах.

Светло – желтым изображено время выполнения для параллельного алгоритма быстрой сортировки на 3 ядрах.

Темно – желтым изображено время выполнения для параллельного алгоритма быстрой сортировки на 4 ядрах.

Рисунок 7 – График зависимости времени работы программы от размера массива и количества ядер

**Вывод:** Было осуществлено распараллеливание процесса сортировки массива алгоритмами пузырька, Шела, быстрой сортировки с помощью технологии OpenMP и была осуществлена оценка времени работы данной программы. Согласно проведенными исследованиям максимальная скорость работы программы была достигнута на 2 потоках.

**ПРИЛОЖЕНИЕ.**

**Приложение 1. Полный код программы**

#include<iostream>

#include<string>

#include<sstream>

#include<omp.h>

using namespace std;

double func\_to\_calc(double x) {

return x\*x;

}

double Simpson(double a, double b, double N) {

double step = (double)(b - a) / N;

double\* X\_int = new double[N + 1];

X\_int[0] = a;

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

X\_int[i] = X\_int[i - 1] + step;

}

double res = 0;

for (int i = 0; i < N-1; i=i+2) {

res += step / 3 \* (func\_to\_calc(X\_int[i]) + 4 \* func\_to\_calc((double)(X\_int[i + 1])) + func\_to\_calc(X\_int[i + 2]));

}

return res;

}

double Simpson\_parallel(double a, double b, int N,int num\_thr) {

double step = (double)(b - a) / N;

double\* X\_int = new double[N + 1];

X\_int[0] = a;

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

X\_int[i] = X\_int[i - 1] + step;

}

double res = 0;

#pragma omp parallel num\_threads(num\_thr) reduction(+:res)

{

#pragma omp for collapse(1)

for (int i = 0; i < N - 1; i = i + 2)

{

res += step / 3 \* (func\_to\_calc(X\_int[i]) + 4 \* func\_to\_calc((double)(X\_int[i + 1])) + func\_to\_calc(X\_int[i + 2]));

}

};

return res;

}

double Simpson\_parallel\_2(double a, double b, int N,int num\_thr) {

double step = (double)(b - a) / N;

double\* X\_int = new double[N + 1];

X\_int[0] = a;

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

X\_int[i] = X\_int[i - 1] + step;

}

double\* res\_f = new double[3];

double res = 0;

#pragma omp parallel num\_threads(num\_thr) reduction(+:res)

{

#pragma omp for collapse(2)

for (int i = 0; i < N - 1; i = i + 2)

{

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

res\_f[j] = func\_to\_calc(X\_int[i + j]);

}

res += step / 3 \* (res\_f[0] + 4 \* res\_f[1] + res\_f[2]);

}

};

return res;

}

int main()

{

setlocale(LC\_ALL, "russian");

double a,b;

int N;

cout << "Начало отрезка" << endl;

cin >>a;

cout << "Конец отрезка" << endl;

cin >> b;

cout << "Количество отрезков для разбиения" << endl;

cin >> N;

double start\_time = clock();

cout << Simpson\_parallel\_2(a, b, N, 2) << endl;

double end\_time = clock();

cout <<"Второй вид 2 ядра:"<<end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson\_parallel(a, b, N, 2) << endl;

end\_time = clock();

cout << "Первый вид 2 ядра:" << end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson\_parallel\_2(a, b, N, 3) << endl;

end\_time = clock();

cout << "Второй вид 3 ядра:" << end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson\_parallel(a, b, N, 3) << endl;

end\_time = clock();

cout << "Первый вид 3 ядра:" << end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson\_parallel\_2(a, b, N,4)<<endl;

end\_time = clock();

cout << "Второй вид 4 ядра:" << end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson\_parallel(a, b, N,4)<<endl;

end\_time = clock();

cout << "Первый вид 4 ядра:" << end\_time - start\_time << endl;

start\_time = clock();

cout << Simpson(a, b, N) << endl;

end\_time = clock();

cout << "Последовательный вид:" << end\_time - start\_time << endl;

}