Министерство образования Российской Федерации

Нижегородский государственный университет имени Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчёт

по Лабораторной работе

Построение выпуклой оболочки – проход Грэхема

Выполнил:

Студент группы 381506-3

Сергеев А. П.

Проверил:

Доцент кафедры МОСТ

Сысоев А. В.

Нижний Новгород

2018

Содержание

[**Введение**](#_gjdgxs) **3**

[**Постановка задачи**](#_qvcouyj19blb) **4**

[**Описание алгоритмов**](#_j9qd3zxmexcw) **5**

[Задача построения минимальной выпуклой оболочки](#_lnxbz9) 5

[Алгоритм Грэхема](#_35nkun2) 6

[Генератор тестов – generator](#_81sr98p3t9z8) 7

[Проверка результата – checker](#_67m7watb3elz) 8

[Алгоритм Грэхема – параллельный вариант OpenMP](#_jvdvwv425k74) 8

[Алгоритм Грэхема – параллельный вариант TBB](#_njqq3m52cf5g) 9

[**Вывод**](#_l8ytuws0hcka) **10**

[**Список литературы**](#_io2dw64kzyq7) **12**

[**Приложение**](#_2bn6wsx) **13**

[Код программы](#_qsh70q) 13

[Grehem.h](#_7z2pgkx5oifz) 13

[Grehem\_Parallel.h](#_mfk3wyjdk0v4) 15

[Grehem\_Parallel\_TBB.h](#_lig0r5xrv5v4) 21

# Введение

Вычислительная геометрия занимается изучением разработки и исследования алгоритмов для решения геометрических проблем. Задача построения выпуклых оболочек, является одной из центральных задач вычислительной геометрии. Важность этой задачи происходит не только из-за огромного количества приложений: в распознавании образов, обработке изображений, базах данных, в задаче раскроя и компоновки материалов, математической статистике, но также и из-за полезности выпуклой оболочки как инструмента решения множества задач вычислительной геометрии.

Эта задача позволяет разрешить целый ряд других, иногда с первого взгляда не связанных с ней вопросов: построение диаграмм Вороного, построение триангуляций и т.д. Построение выпуклой оболочки конечного множества точек на плоскости довольно широко исследовано и имеет множество приложений. Очень широко алгоритмы построения выпуклой оболочки используются в геоинформатике и геоинформационных системах.

Задача построения выпуклой оболочки имеет давнюю историю. Она является одной из первых задач вычислительной геометрии, с которой начала зарождаться эта наука. В настоящее время известно достаточно большое число алгоритмов построения выпуклой оболочки, например алгоритм Грэхема, который рассматривается в данной лабораторной работе.

# Постановка задачи

Цель данной лабораторной работы – написать программу, в которой будет реализовано построение выпуклой оболочки методом Грэхема. Предоставить разные программные реализации алгоритма: последовательную, параллельную с использованием OpenMP и параллельную с использованием TBB. Также реализовать генератор тестов - generator и программу для проверки результата - checker.

# Описание алгоритмов

## Задача построения минимальной выпуклой оболочки

Рассмотрим задачу построения минимальной выпуклой оболочки. Пусть на плоскости задано конечное множество точек X, требуется построить минимальную выпуклую оболочку множества X.

Оболочкой множества X называется любая замкнутая кривая L без самопересечений такая, что все точки из X лежат внутри этой кривой, например рис. 3.

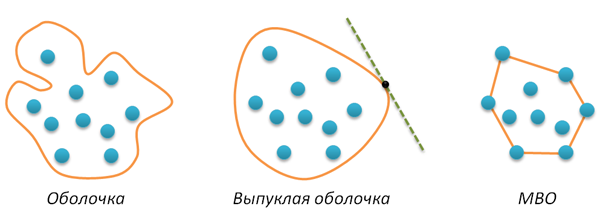


Рис. 3 – Оболочка множества X

Если кривая L является выпуклой, то соответствующая оболочка также называется выпуклой (рис. 4).

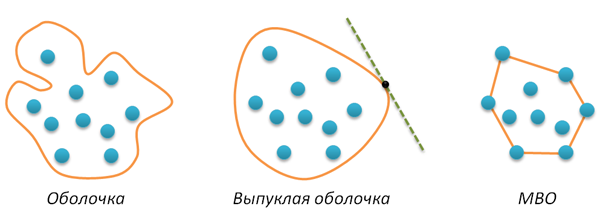


Рис. 4 – Выпуклая оболочка множества X

Минимальной выпуклой оболочкой называется выпуклая оболочка минимальной длины (рис. 5).

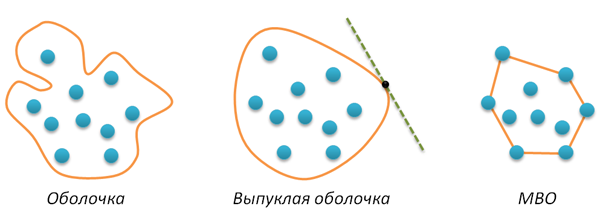


Рис. 5 – Минимальная выпуклая оболочка множества X

Главной особенностью минимальной выпуклой оболочки множества точек является то, что эта оболочка представляет собой выпуклый многоугольник, вершинами которого являются некоторые точки из этого множества. Поэтому задача поиска минимальной выпуклой оболочки в конечном итоге сводится к упорядочиванию и отбору нужных точек из множества. Для построения минимальной выпуклой оболочки существует множество алгоритмов, одним из них является алгоритм Грэхема.

## Алгоритм Грэхема

Алгоритм Грэхема является достаточно известным, так как с его помощью можно построить минимальную выпуклую оболочку за время O(N\*logN), также он является асимптотически оптимальным (доказано, что не существует алгоритма с лучшей асимптотикой).

Алгоритм Грэхема по сути делится на три шага.

1. На первом шаге ищется любая точка из заданных, которая гарантированно входит в минимальную выпуклую оболочку. Очевидно, что такой точкой будет, например, точка с наименьшей координатой по оси X или по оси Y. Эту точку, далее называемой опорной, перемещаем в начало массива точек. Далее для более удобного упорядочивания точек меняется система координат — опорная точка переносится в начало координат (0,0). Следовательно для всех точек выполняется сдвиг на координаты опорной точки.
2. Второй шаг в алгоритме Грэхема — сортировка всех точек , по степени их “левизны” относительно опорной точки. Полагается, что B<C, если точка С находится в левой плоскости относительно вектора RB, где R — опорная точка (рис. 6).

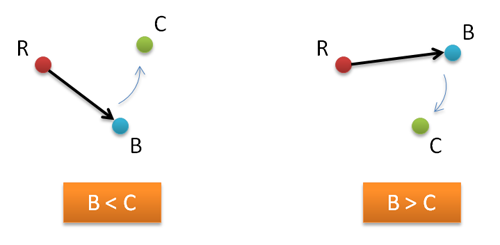


Рис. 6 – Положение точки

Для выполнения такого упорядочивания можно применить любой алгоритм сортировки, основанный на попарном сравнении элементов, например, уже реализованную быструю сортировку.

В результате после сортировке мы получим рис. 7.

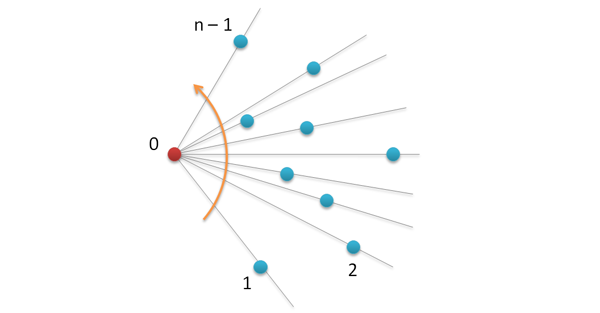


Рис. 7 – Упорядоченный набор точек относительно опорной

1. Третий шаг алгоритма — построение минимальной выпуклой оболочки, также называемый обходом Грэхема.

Обход Грэхема:

* 1. Создаётся стек, в него вставляют первые 2 точки.
  2. Цикл для текущей точки z не принадлежащей стеку:
     1. Читается точка y из стека и точка x следующая в стеке после y.
     2. Если z строго принадлежит левой полуплоскости относительно вектора xy, то z заносятся в стек. Цикл заканчивается
     3. Иначе достаётся y. Цикл начинается сначала, если в стеке осталось хотя бы 2 элемента, в противном случае вставить z в стек.

В результате в стеке будут лежать вершины минимальной выпуклой оболочки в новой системе координат с центром в опорной точке. Для получения ответа потребуется вернутся к старой системе координат.

## Генератор тестов – generator

На вход генератору теста входят (запуск через командную строку):

1. Номер теста.
2. Имя файла входных данных.
3. Имя файла с выходными данными, в котором будет находится выпуклая оболочка.

Всего тестов 27 с номерами от 0 до 26. От номера теста зависит размер создаваемого массива точек, каждый раз генерируется массив из случайных точек. После генерации массив и его размер записываются в файл с входными данными (имя файла указывается при запуске программы). После чего генератор построит выпуклую оболочку и запишет её в файл с выходными данными.

## Проверка результата – checker

На вход поступают (запуск через командную строку):

1. Имя файла выходных данных генератора тестов.
2. Файл с проверяемыми результатами.

Checker проверяет верность результата сопоставляя результат генератора тестов и вводимый результат.

Вердикт проверки будет записан в файл result.txt.

## Алгоритм Грэхема – параллельный вариант OpenMP

Алгоритм Грэхема не может быть полностью выполнен параллельно, так как построение выпуклой оболочки происходит полностью последовательно. Но возможно распараллелить другие действия: используемый алгоритм сортировки и поиск минимального элемента.

Именно с помощью распараллеливания перечисленных действий выполнена параллельная реализация алгоритма Грэхема.

* Алгоритм сортировки — используется параллельная реализация сортировки Хоара.
* Поиск минимального элемента, также можно осуществить путём сортировки.

Для параллельной реализации сортировки используется стратегия простого слияния.

Идея простого слияния заключается в том, что один поток может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние n массивов могут выполнять n/2 параллельных потоков. На следующем шаге слияние n/2 полученных массивов будут выполнять n/4 потоков и т.д. (рис. 8).



Рис. 8 – Простое слияние

То есть изначальный массив делится на части равный size/nThreads, где size – размер массива, а nThreads – количество потоков. Каждый поток выполняет сортировку своего участка массива, после чего с помощью простого слияния массив объединяется в один упорядоченный.

## Алгоритм Грэхема – параллельный вариант TBB

В реализации TBB версии сортировки используется распараллеливание рекурсии с помощью средств TBB, а именно логических задач – task.

Для этого создаётся специальный класс SortTask, наследуемый от task(рис 9).

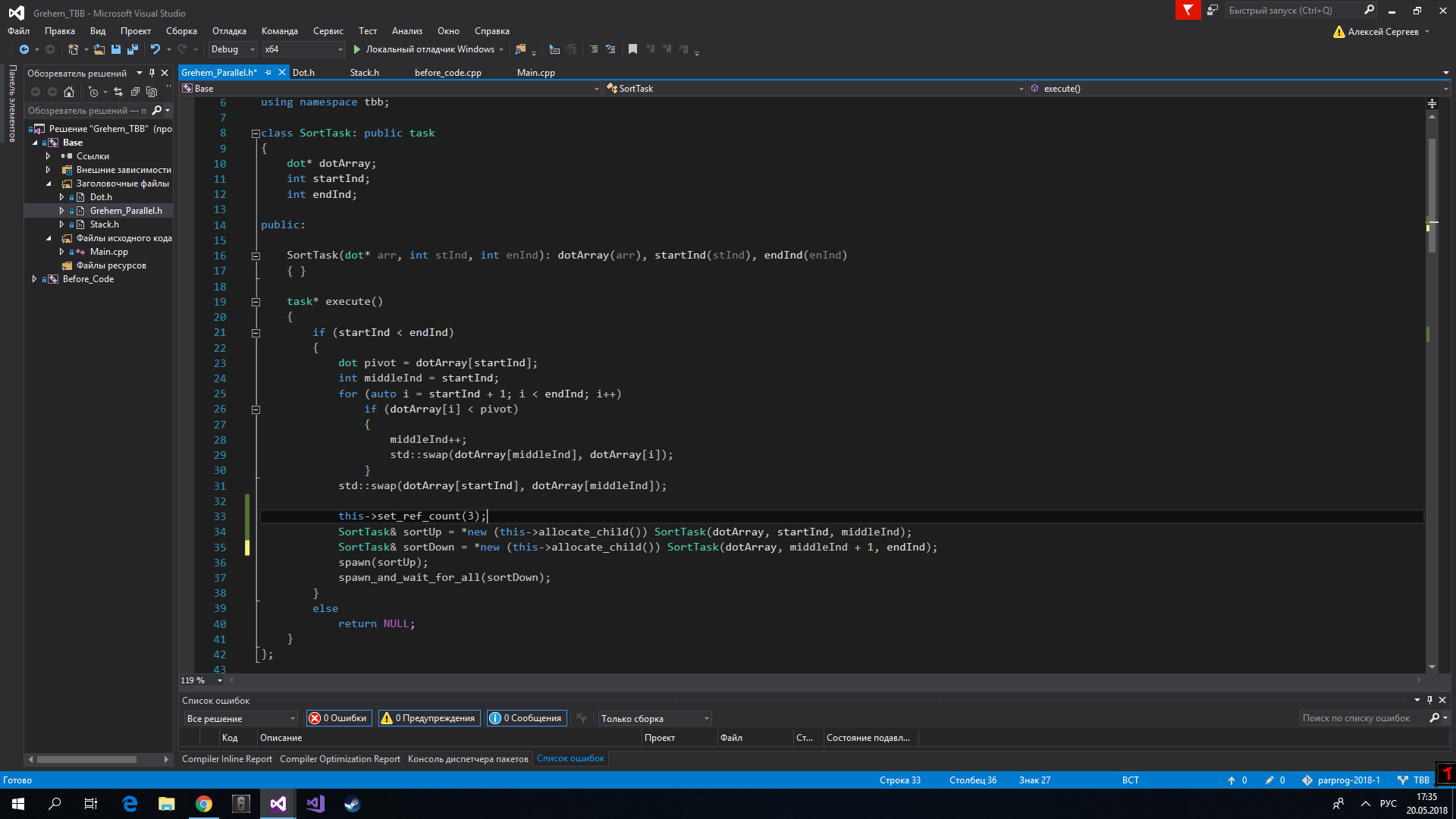


Рис. 9 – Класс SortTask

SortTask выполняет “рекурсивную” сортировку массива.

# Вывод

В результате выполнения работы была написана программа, в которой реализовано построение выпуклой оболочки методом Грэхема с несколькими реализациями: последовательная, OpenMp версия, TBB версия.

Также реализованы генератор тестов - generator и программа для проверки результата - checker.

Далее в Таблице 1 представлено сравнение времени выполнения разных версий реализации метода Грэхема.

Таблица 1

Результаты выполнений алгоритма Грэхема

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер | Версия | | | | |
| Последовательная | OpenMP | | TBB | |
| Количество потоков | | | | |
| 1 | 2 | 4 | 2 | 4 |
| Время выполнения в микросекундах | | | | |
| 100 | 15 | 1.392 | 2.146 | 1.567 | 1.988 |
| 1.000 | 137 | 1.042 | 2.182 | 1.854 | 2.367 |
| 10.000 | 2.251 | 2.441 | 2.391 | 2.678 | 2.650 |
| 100.000 | 24.915 | 13.423 | 12.550 | 15.024 | 13.721 |
| 1.000.000 | 255.519 | 169.504 | 106.896 | 186.739 | 137.540 |
| 10.000.000 | 5998368 | 3792728 | 2026985 | 4017815 | 2652746 |

Изучив полученные значения из таблицы, нетрудно заметить, что для размера меньшем 10.000 последовательный алгоритм работает быстрее чем параллельные из-за накладных расходов. Но уже при размере большем 10.000 обе версии OpenMP и TBB дают прирост к скорости.

В итоге при размере массива равном 10.000.000 получаем:

* Прирост OpenMP на 2 потоках = 1.6, на 4 = 2.95
* Прирост TBB версии на 2 потоках = 1.5, а на 4 = 2.3

Очевидно, что в данных реализациях OpenMP версия получилась быстрее, чем TBB, следовательно эффективнее использовать OpenMP версию в данной программе.

# Список литературы

1. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.
2. Гергель В.П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем. Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И. Лобачевского, 2010 .
3. Корняков К.В., Мееров И.Б., Сиднев А.А., Сысоев А.В., Шишков А.В. Инструменты параллельного программирования в системах с общей памятью. – Учебное пособие / Под ред. проф. В.П. Гергеля. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2010. – 201 с.

# Приложение

## Код программы

### Grehem.h

#include "Stack.h"

#include <iostream>

void quickSort(dot\* arr, int startInd, int endInd)

{

if (startInd < endInd)

{

dot pivot = arr[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if (arr[i] < pivot)

{

middleInd++;

std::swap(arr[middleInd], arr[i]);

}

std::swap(arr[startInd], arr[middleInd]);

quickSort(arr, startInd, middleInd);

quickSort(arr, middleInd + 1, endInd);

}

}

void quickSortSearchMin(dot\* arr, int startInd, int endInd)

{

if (startInd < endInd)

{

dot pivot = arr[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if ((arr[i].x < pivot.x) || ((arr[i].x == pivot.x) && (arr[i].y < pivot.y)))

{

middleInd++;

std::swap(arr[middleInd], arr[i]);

}

std::swap(arr[startInd], arr[middleInd]);

quickSortSearchMin(arr, startInd, middleInd);

quickSortSearchMin(arr, middleInd + 1, endInd);

}

}

void searchMinElement(dot\* dotArray, int size)

{

quickSortSearchMin(dotArray, 0, size);

}

std::pair<dot\*, int> grehemMethod(dot\* dotArray, int size)

{

searchMinElement(dotArray, size - 1);

dot move = dotArray[0];

for (auto i = 0; i < size - 1; i++)

dotArray[i] = dotArray[i] - move;

quickSort(dotArray, 1, size - 1);

dotArray[size - 1] = dotArray[0];

stack dotStack(size / 2);

dot dotY, dotX;

dotStack.push(dotArray[0]);

dotStack.push(dotArray[1]);

for (auto i = 2; i < size; i++)

{

while (true)

{

if (dotStack.getSize() > 1)

{

dotY = dotStack.pop();

dotX = dotStack.get();

if ((dotArray[i].x\*(dotX.y - dotY.y) + dotArray[i].y\*(dotY.x - dotX.x) + dotX.x\*dotY.y - dotX.y\*dotY.x) > 0)

{

dotStack.push(dotY);

dotStack.push(dotArray[i]);

break;

}

else

{

if (dotStack.getSize() == 1)

{

dotStack.push(dotArray[i]);

break;

}

}

}

}

}

dot\* resultArray = dotStack.getArray();

for (auto i = 0; i < dotStack.getSize(); i++)

{

resultArray[i] = resultArray[i] + move;

}

return std::make\_pair(resultArray, dotStack.getSize());

}

### Grehem\_Parallel.h

#include "Stack.h"

#include <iostream>

#include <omp.h>

void merge(int firstThread, int secThread, dot\* arr, std::pair<int, int>\* ind\_array)

{

int size\_1 = ind\_array[firstThread].second - ind\_array[firstThread].first;

int size\_2 = ind\_array[secThread].second - ind\_array[secThread].first;

dot\* tmparr\_1 = new dot[size\_1];

for (int i = 0; i < size\_1; i++)

{

tmparr\_1[i] = arr[ind\_array[firstThread].first + i];

}

dot\* tmparr\_2 = new dot[size\_2];

for (int i = 0; i < size\_2; i++)

{

tmparr\_2[i] = arr[ind\_array[secThread].first + i];

}

int iter1 = 0;

int iter2 = 0;

while ((iter1 != size\_1) && (iter2 != size\_2))

{

if (tmparr\_1[iter1] < tmparr\_2[iter2])

{

arr[ind\_array[firstThread].first + iter1 + iter2] = tmparr\_1[iter1];

iter1++;

}

else

{

arr[ind\_array[firstThread].first + iter1 + iter2] = tmparr\_2[iter2];

iter2++;

}

}

if (iter1 == size\_1)

{

for (int i = iter2; i < size\_2; i++)

{

arr[ind\_array[firstThread].first + i + size\_1] = tmparr\_2[i];

}

}

else if (iter2 == size\_2)

{

for (int i = iter1; i < size\_1; i++)

{

arr[ind\_array[firstThread].first + i + size\_2] = tmparr\_1[i];

}

}

ind\_array[firstThread].second = ind\_array[secThread].second;

ind\_array[secThread].first = 0;

ind\_array[secThread].second = 0;

delete[] tmparr\_1;

delete[] tmparr\_2;

}

void mergeForMin(int firstThread, int secThread, dot\* arr, std::pair<int, int>\* ind\_array)

{

int size\_1 = ind\_array[firstThread].second - ind\_array[firstThread].first;

int size\_2 = ind\_array[secThread].second - ind\_array[secThread].first;

dot\* tmparr\_1 = new dot[size\_1];

for (int i = 0; i < size\_1; i++)

{

tmparr\_1[i] = arr[ind\_array[firstThread].first + i];

}

dot\* tmparr\_2 = new dot[size\_2];

for (int i = 0; i < size\_2; i++)

{

tmparr\_2[i] = arr[ind\_array[secThread].first + i];

}

int iter1 = 0;

int iter2 = 0;

while ((iter1 != size\_1) && (iter2 != size\_2))

{

if ((tmparr\_1[iter1].x < tmparr\_2[iter2].x) || ((tmparr\_1[iter1].x == tmparr\_2[iter2].x) && (tmparr\_1[iter1].y < tmparr\_2[iter2].y)))

{

arr[ind\_array[firstThread].first + iter1 + iter2] = tmparr\_1[iter1];

iter1++;

}

else

{

arr[ind\_array[firstThread].first + iter1 + iter2] = tmparr\_2[iter2];

iter2++;

}

}

if (iter1 == size\_1)

{

for (int i = iter2; i < size\_2; i++)

{

arr[ind\_array[firstThread].first + i + size\_1] = tmparr\_2[i];

}

}

else if (iter2 == size\_2)

{

for (int i = iter1; i < size\_1; i++)

{

arr[ind\_array[firstThread].first + i + size\_2] = tmparr\_1[i];

}

}

ind\_array[firstThread].second = ind\_array[secThread].second;

ind\_array[secThread].first = 0;

ind\_array[secThread].second = 0;

delete[] tmparr\_1;

delete[] tmparr\_2;

}

void quickSort(dot\* arr, int startInd, int endInd)

{

if (startInd < endInd)

{

dot pivot = arr[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if (arr[i] < pivot)

{

middleInd++;

std::swap(arr[middleInd], arr[i]);

}

std::swap(arr[startInd], arr[middleInd]);

quickSort(arr, startInd, middleInd);

quickSort(arr, middleInd + 1, endInd);

}

}

void sortParallel(dot\* arr, int startInd, int endInd, int numThreads)

{

if (endInd - startInd < numThreads \* 2) //Если размер массива меньше двойного кол-ва потоков выполняется сортировка всего массива в одном потоке

{

quickSort(arr, startInd, endInd);

return;

}

std::pair<int, int>\* ind\_array = new std::pair<int, int>[numThreads];

std::div\_t portion = div((endInd - startInd), numThreads); // size / num\_threads

ind\_array[0].first = startInd;

ind\_array[0].second = ind\_array[0].first + portion.quot;

if (portion.rem > 0)

ind\_array[0].second++;

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

{

ind\_array[i].first = ind\_array[i - 1].second;

ind\_array[i].second = ind\_array[i].first + portion.quot;

if (portion.rem > i)

ind\_array[i].second++;

}//закончено распределение индексов

int steps\_amount = std::ceil(log2(numThreads)); // расчёт необходимого числа шагов в цикле слияний массивов

#pragma omp parallel

{

int thread\_num = omp\_get\_thread\_num();

quickSort(arr, ind\_array[thread\_num].first, ind\_array[thread\_num].second);

#pragma omp barrier

for (int i = 0; i < steps\_amount; i++)

{

if (thread\_num % (int)(pow(2, i + 1)) == 0) //Если поток ведущий - сделать слияние с "соседним" потоком

{

if (thread\_num + (int)pow(2, i) < numThreads) // Если поток с которым происходит слияние существует

{

merge(thread\_num, thread\_num + (int)pow(2, i), arr, ind\_array);// Слить массив первого потока с массивом второго

}

}

#pragma omp barrier

}

}

delete[] ind\_array;

}

void quickSortSearchMin(dot\* arr, int startInd, int endInd)

{

if (startInd < endInd)

{

dot pivot = arr[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if ((arr[i].x < pivot.x) || ((arr[i].x == pivot.x) && (arr[i].y < pivot.y)))

{

middleInd++;

std::swap(arr[middleInd], arr[i]);

}

std::swap(arr[startInd], arr[middleInd]);

quickSortSearchMin(arr, startInd, middleInd);

quickSortSearchMin(arr, middleInd + 1, endInd);

}

}

void searchMinElement(dot\* dotArray, int size, int numThreads)

{

if (size < numThreads \* 2) //Если размер массива меньше двойного кол-ва потоков выполняется сортировка всего массива в одном потоке

{

quickSortSearchMin(dotArray, 0, size);

return;

}

std::pair<int, int>\* ind\_array = new std::pair<int, int>[numThreads];

std::div\_t portion = div(size, numThreads); // size / num\_threads

ind\_array[0].first = 0;

ind\_array[0].second = portion.quot;

if (portion.rem > 0)

ind\_array[0].second++;

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

{

ind\_array[i].first = ind\_array[i - 1].second;

ind\_array[i].second = ind\_array[i].first + portion.quot;

if (portion.rem > i)

ind\_array[i].second++;

}//закончено распределение индексов

int steps\_amount = std::ceil(log2(numThreads)); // расчёт необходимого числа шагов в цикле слияний массивов

#pragma omp parallel

{

int thread\_num = omp\_get\_thread\_num();

quickSortSearchMin(dotArray, ind\_array[thread\_num].first, ind\_array[thread\_num].second);

#pragma omp barrier

for (int i = 0; i < steps\_amount; i++)

{

if (thread\_num % (int)(pow(2, i + 1)) == 0) //Если поток ведущий - сделать слияние с "соседним" потоком

{

if (thread\_num + (int)pow(2, i) < numThreads) // Если поток с которым происходит слияние существует

{

mergeForMin(thread\_num, thread\_num + (int)pow(2, i), dotArray, ind\_array);// Слить массив первого потока с массивом второго

}

}

#pragma omp barrier

}

}

delete[] ind\_array;

}

std::pair<dot\*, int> grehemMethod\_OpenMP(dot\* dotArray, int size, int numThreads)

{

searchMinElement(dotArray, size - 1, numThreads);

dot move = dotArray[0];

#pragma omp parallel for if (omp\_get\_num\_threads()\*2 < size)

for (auto i = 0; i < size - 1; i++)

dotArray[i] = dotArray[i] - move;

sortParallel(dotArray, 1, size - 1, numThreads);

dotArray[size - 1] = dotArray[0];

stack dotStack(size / 2);

dot dotY, dotX;

dotStack.push(dotArray[0]);

dotStack.push(dotArray[1]);

for (auto i = 2; i < size; i++)

{

while (true)

{

if (dotStack.getSize() > 1)

{

dotY = dotStack.pop();

dotX = dotStack.get();

if ((dotArray[i].x\*(dotX.y - dotY.y) + dotArray[i].y\*(dotY.x - dotX.x) + dotX.x\*dotY.y - dotX.y\*dotY.x) > 0)

{

dotStack.push(dotY);

dotStack.push(dotArray[i]);

break;

}

else

{

if (dotStack.getSize() == 1)

{

dotStack.push(dotArray[i]);

break;

}

}

}

}

}

dot\* resultArray = dotStack.getArray();

#pragma omp parallel for if (omp\_get\_num\_threads()\*2 < size)

for (auto i = 0; i < dotStack.getSize(); i++)

resultArray[i] = resultArray[i] + move;

return std::make\_pair(resultArray, dotStack.getSize());

}

### Grehem\_Parallel\_TBB.h

#include "Stack.h"

#include <iostream>

#include "task\_scheduler\_init.h"

#include "task.h"

using namespace tbb;

class SortTask: public task

{

dot\* dotArray;

int startInd;

int endInd;

public:

SortTask(dot\* arr, int stInd, int enInd): dotArray(arr), startInd(stInd), endInd(enInd)

{ }

task\* execute()

{

if (startInd < endInd)

{

dot pivot = dotArray[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if (dotArray[i] < pivot)

{

middleInd++;

std::swap(dotArray[middleInd], dotArray[i]);

}

std::swap(dotArray[startInd], dotArray[middleInd]);

this->set\_ref\_count(3);

SortTask& sortUp = \*new (this->allocate\_child()) SortTask(dotArray, startInd, middleInd);

SortTask& sortDown = \*new (this->allocate\_child()) SortTask(dotArray, middleInd + 1, endInd);

spawn(sortUp);

spawn\_and\_wait\_for\_all(sortDown);

}

else

return NULL;

}

};

class SortMinTask : public task

{

dot\* dotArray;

int startInd;

int endInd;

public:

SortMinTask(dot\* arr, int stInd, int enInd) : dotArray(arr), startInd(stInd), endInd(enInd)

{ }

task\* execute()

{

if (startInd < endInd)

{

dot pivot = dotArray[startInd];

int middleInd = startInd;

for (auto i = startInd + 1; i < endInd; i++)

if ((dotArray[i].x < pivot.x) || ((dotArray[i].x == pivot.x) && (dotArray[i].y < pivot.y)))

{

middleInd++;

std::swap(dotArray[middleInd], dotArray[i]);

}

std::swap(dotArray[startInd], dotArray[middleInd]);

this->set\_ref\_count(3);

SortMinTask& sortUp = \*new (this->allocate\_child()) SortMinTask(dotArray, startInd, middleInd);

SortMinTask& sortDown = \*new (this->allocate\_child()) SortMinTask(dotArray, middleInd + 1, endInd);

spawn(sortUp);

spawn\_and\_wait\_for\_all(sortDown);

}

else

return NULL;

}

};

void sortParallel(dot\* arr, int startInd, int endInd, int numThreads)

{

SortTask& sortT = \*new (task::allocate\_root()) SortTask(arr, startInd, endInd);

task::spawn\_root\_and\_wait(sortT);

}

void searchMinElement(dot\* dotArray, int size, int numThreads)

{

SortMinTask& sortMin = \*new (task::allocate\_root()) SortMinTask(dotArray, 0, size);

task::spawn\_root\_and\_wait(sortMin);

}

std::pair<dot\*, int> grehemMethod\_TBB(dot\* dotArray, int size, int numThreads)

{

tbb::task\_scheduler\_init init(numThreads);

searchMinElement(dotArray, size - 1, numThreads);

dot move = dotArray[0];

for (auto i = 0; i < size - 1; i++)

dotArray[i] = dotArray[i] - move;

sortParallel(dotArray, 1, size - 1, numThreads);

dotArray[size - 1] = dotArray[0];

stack dotStack(size / 2);

dot dotY, dotX;

dotStack.push(dotArray[0]);

dotStack.push(dotArray[1]);

for (auto i = 2; i < size; i++)

{

while (true)

{

if (dotStack.getSize() > 1)

{

dotY = dotStack.pop();

dotX = dotStack.get();

if ((dotArray[i].x\*(dotX.y - dotY.y) + dotArray[i].y\*(dotY.x - dotX.x) + dotX.x\*dotY.y - dotX.y\*dotY.x) > 0)

{

dotStack.push(dotY);

dotStack.push(dotArray[i]);

break;

}

else

{

if (dotStack.getSize() == 1)

{

dotStack.push(dotArray[i]);

break;

}

}

}

}

}

dot\* resultArray = dotStack.getArray();

for (auto i = 0; i < dotStack.getSize(); i++)

resultArray[i] = resultArray[i] + move;

return std::make\_pair(resultArray, dotStack.getSize());

}