Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н. И. Лобачевского»

Институт информационных технологий, математики и механики

Направление: «Программная инженерия»

Отчет

по лабораторным работам курса «Параллельное программирование»

«Построение выпуклой оболочки. Алгоритм Джарвиса»

Выполнил:

студент группы 1608

Чураков С.С.

Проверил:

Доцент кафедры МОиСТ, к.т.н.

А.В. Сысоев

Нижний Новгород

2019

Содержание

1. Введение

Вычислительная геометрия - раздел информатики, в котором рассматриваются алгоритмы решения геометрических задач.

Некоторые области применения вычислительной геометрии:

- Инженерное проектирование
- Распознавание образов
- Компьютерная графика

Построение выпуклой оболочки - одна из центральных задач вычислительной геометрии. Выпуклые оболочки часто применяются в распознавании образов, обработке изображений, в задачах оптимальной компоновки частей при раскройке материала, в моделировании движения и много где еще.

Существует множество способов решения этой задачи. Одним из распространенных алгоритмов является обход Джарвиса, так же называемый алгоритмом заворачивания подарка.

2. Постановка задачи

Цель данной работы - реализация алгоритма Джарвиса, выполняющего поиск выпуклой оболочки. На вход алгоритму поступает N точек, лежащих на плоскости, результатом его выполнения является выпуклая оболочка заданного множества точек, представленная в виде множества M точек. Алгоритм должен быть реализован как для последовательного, так и для параллельного выполнения (с использованием технологий TBB и OpenMP).

Этапы выполнения задачи:

1. Изучение задачи построения выпуклой оболочки и алгоритма Джарвиса.
2. Программная реализация последовательной версии алгоритма Джарвиса.
3. Поиск оптимальной схемы распараллеливания алгоритма.
4. Программная реализация параллельной версии алгоритма Джарвиса с использованием OpenMP.
5. Программная реализация параллельной версии алгоритма Джарвиса с использованием TBB.
6. Сравнение времени выполнения алгоритма на различных реализациях, вычисление ускорения и эффективности параллельных реализаций.
7. Анализ результатов и подведение итогов.

3. Описание алгоритма

3.1. Сведения и определения вычислительной геометрии

Ниже приведены сведения и определения из вычислительной геометрии, необходимые для понимания алгоритма Джарвиса:

- **Выпуклое множество** - множество, в котором все точки отрезка, образуемого любыми двумя точками данного множества, также принадлежат данному множеству.
- **Выпуклая оболочка** заданного множества точек - наименьшее выпуклое множество, содержащее в себе все точки этого множества.

3.2. Краткое описание алгоритма

Сначала нужно найти начальную точку *p1* выпуклой оболочки. Найдем среди заданных точек самую нижнюю; если таких точек несколько, возьмем самую левую из них. Эта точка гарантированно является вершиной выпуклой оболочки.

Построим прямую *l*, проходящую через начальную точку и параллельную оси абсцисс. Находим такую точку *p2*, что угол между прямой *l* и отрезком *p1p2*, находящийся по левую сторону прямой *l*, минимален. *p2* - следующая вершина выпуклой оболочки.

Теперь находим такую точку *p3*, что угол между отрезком *p2p3* и продолжением отрезка *p1p2*, минимален. *p3* - следующая точка выпуклой оболочки.

Находим точки *p4*, *p5*... так же, как нашли *p3* на прошлом шаге, используя значения двух предыдущих вершин выпуклой оболочки вместо точек *p1*, *p2*.

Алгоритм Джарвиса будет завершен тогда, когда последняя найденная точка совпадет с начальной. На выход будет выдаваться упорядоченное множество точек выпуклой оболочки.

Сложность алгоритма - O(n*M), где n - число точек всего множества, M - число точек выпуклой оболочки. При равномерном случайном распределении точек на плоскости (именно так задается множество точек в программной реализации) M мало (обычно, не больше 30), сложность алгоритма при этих условиях близка к линейной.

3.3. Пример работы алгоритма

Рассмотрим алгоритм Джарвиса на простом примере.

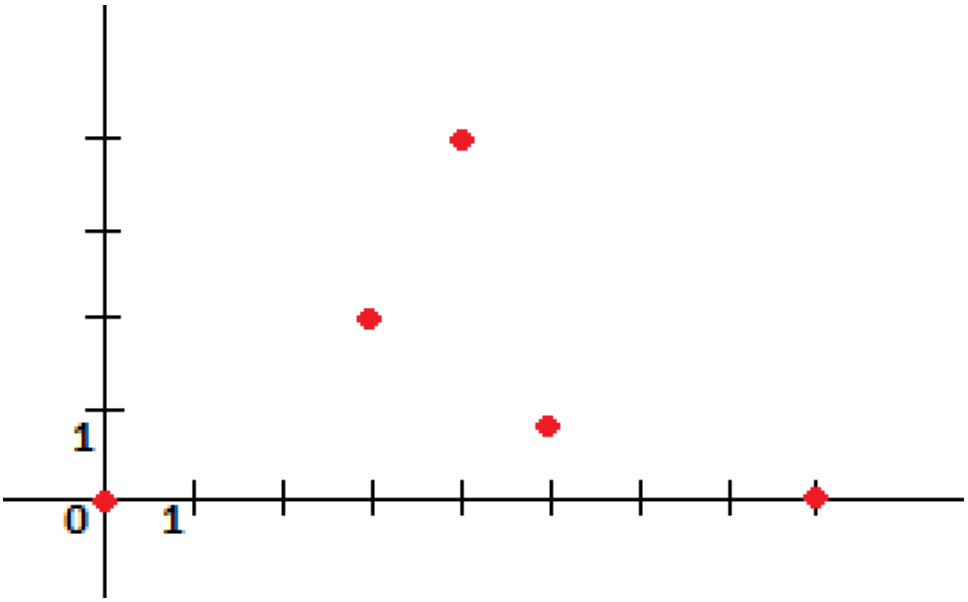


Рисунок 1. Пример работы алгоритма Джарвиса, входные данные

Таблица с координатами точек:

№ точки	1	2	3	4	5
X	0	8	4	3	5
Y	0	0	4	2	1

Зелеными будем отображать точки, входящие в выпуклую оболочку. Если точки не входят в выпуклую оболочку, либо на данном этапе мы не знаем, входят ли они в выпуклую оболочку, они останутся красными.

Найдем начальную точку выпуклой оболочки.

Имеются две точки с наименьшей координатой Y: точки 1 и 2. У точки 1 координата X меньше, чем у точки 2. Точка 1 - начальная точка.

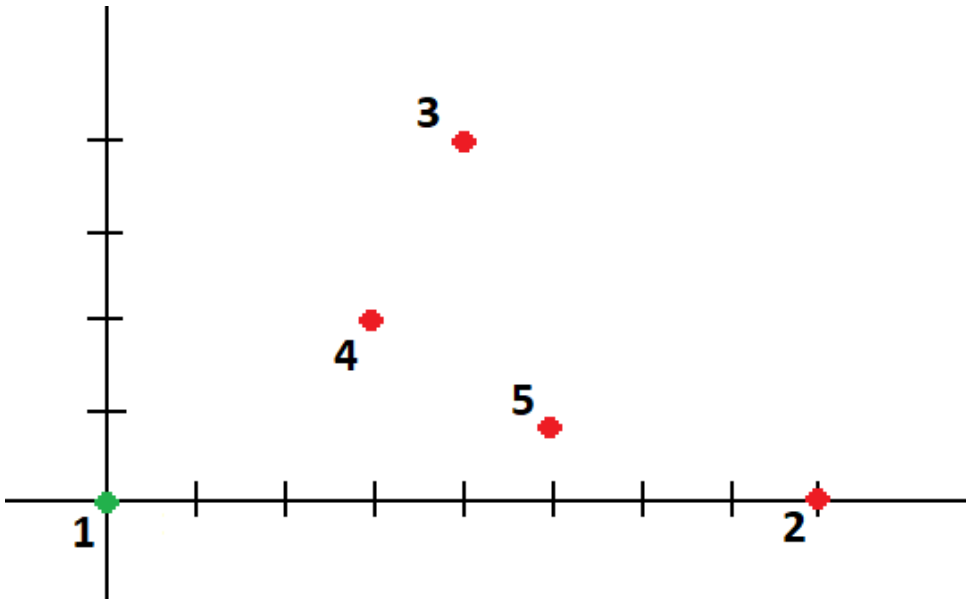


Рисунок 2. Пример работы алгоритма Джарвиса, шаг 1

Проведем через точку 1 горизонтальную прямую, она совпадет с осью абсцисс. Построим отрезки от точки 1 к каждой точке и найдем тот, угол между осью абсцисс и которым минимален (угол считается против часовой). Таким является отрезок 1-2, угол между прямой и отрезком нулевой. Точка 2 - следующая точка выпуклой оболочки.

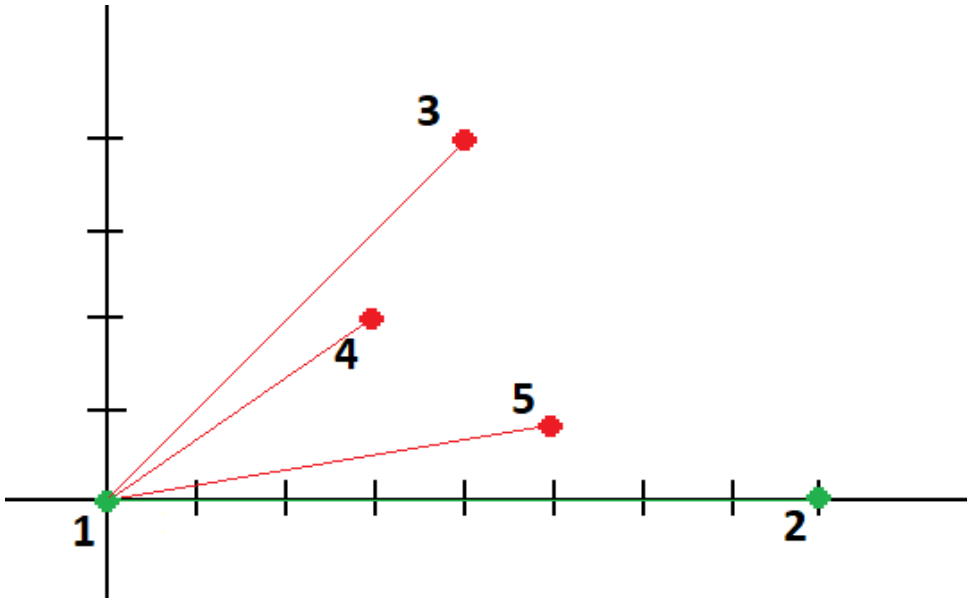


Рисунок 3. Пример работы алгоритма Джарвиса, шаг 2

Построим отрезки от точки 2 к каждой точке, найдем тот, что лежит под наименьшим углом к продолжению отрезка 1-2. Точка 3 - третья точка выпуклой оболочки.

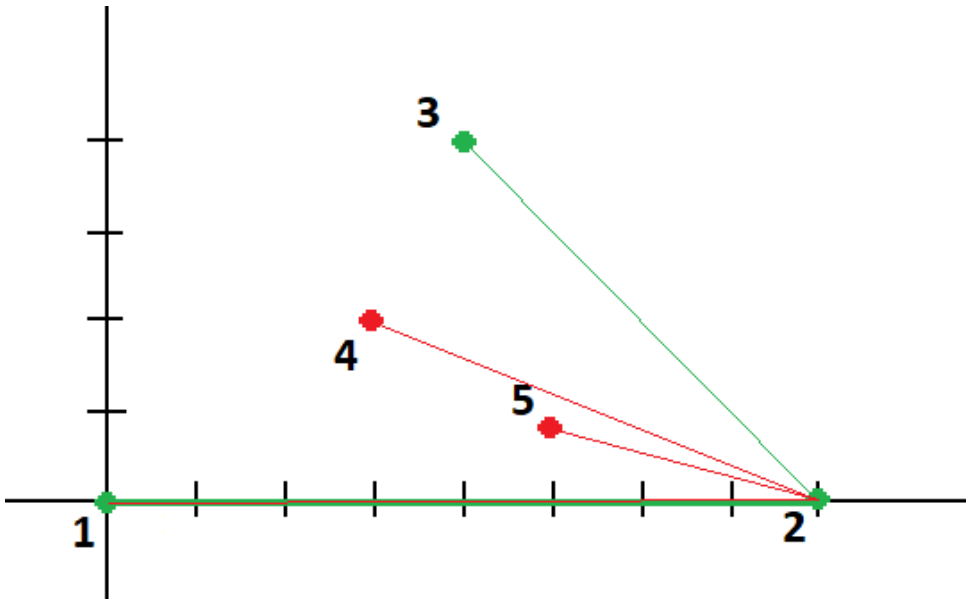


Рисунок 4. Пример работы алгоритма Джарвиса, шаг 3

Проделаем то же самое с точкой 3 и отрезком 2-3. Результатом становится точка 1, также являющаяся начальной.

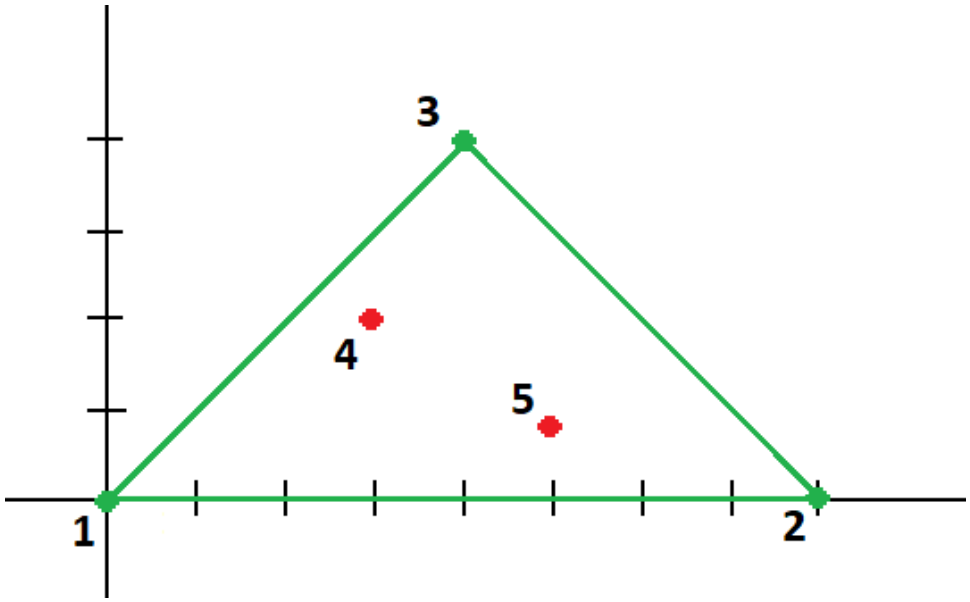


Рисунок 5. Пример работы алгоритма Джарвиса, завершение

Теперь выпуклая оболочка завершена. Она формируется точками 1-2-3; результат можно представить в виде упорядоченной последовательности точек.

4. Схема распараллеливания

Существует достаточно простая и эффективная схема распараллеливания алгоритма Джарвиса, которая заключается в распараллеливании вычислений отдельно для каждого шага алгоритма.

Чтобы распараллелить поиск начальной точки, то есть точки с наименьшими координатами x, y, выполним следующие шаги:

- Разделим точки на n групп примерно одинакового размера и передадим каждому потоку свою группу точек.
- Каждый поток найдет в своей группе точку с наименьшими координатами.
- Мастер-поток соберет результаты каждого потока. Среди полученных точек мастер-поток найдет точку с наименьшими координатами, эта точка - результат поиска.

Чтобы распараллелить поиск точки *c*, такой, что угол между продолжением заданного отрезка *a-b* и отрезком *b-c* является наименьшим, в дальнейшем называемой точкой с наименьшим углом, выполним следующие шаги:

- Разделим точки на n групп примерно одинакового размера и передадим каждому потоку свою группу точек.
- Каждый поток найдет в своей группе точку с наименьшим углом.
- Мастер-поток соберет результаты каждого потока. Среди полученных точек мастер-поток найдет точку с наименьшим углом, эта точка - результат поиска.

Стоит отметить, что существуют способы распараллелить алгоритм Джарвиса, а не его части, но на практике они менее эффективны и более сложны в реализации.

5. Метод решения

5.1. Последовательная реализация

Последовательная реализация алгоритма является основной для разработки программы, она включена во все последующие реализации с целью проверки корректности результатов.

Последовательную реализацию можно разбить на несколько этапов:

- Ввод входных параметров

  При запуске программы возможно указать некоторые параметры задачи: число точек и минимальное/максимальное значения их координат. Параметры вводить необязательно, тогда будут использованы значения по умолчанию.
- Генерация множества случайных точек

  Создаются массивы X_coord и Y_coord размера Size для значений координат точек. Координаты точки p - (X_coord[p], Y_coord[p]). Генерация множества случайных точек осуществляется с помощью метода RandomizeArray(double* rarray, const int& Size, const int& min, const int& max)

  Параметры функции - указатель на массив, заполняемый случайными значениями, его размер, ограничения на значения массива.

  После заполнения массивов случайными значениями координаты точек выводятся на экран.
- Поиск выпуклой оболочки

  Если Size <3, то каждая из имеющихся точек содержится в выпуклой оболочке. Иначе выполняется алгоритм Джарвиса.

  Создается массив для результатов алгоритма Envelope, число элементов в нем заранее неизвестно. PNum - число известных элементов выпуклой оболочки, увеличивающееся на каждом шаге.

  Поиск начальной точки осуществляется с помощью функции FindBLPoint(double* X, double* Y, const int& Size), параметры функции - указатели на массивы координат точек и их размер.

  В дальнейшем поиск точек оболочки осуществляется с помощью функции FindPWithMinAngle(double* X, double* Y, const int& Size, const double& x1, const

double& y1, const double& x2, const double& y2), параметры функции - указатели на массивы координат точек, их размер и координаты последних двух точек оболочки. При поиске второй точки оболочки последние четыре параметра - (x-1, y, x, y), x и y - координаты первой точки, так можно эмулировать горизонтальную прямую.
Функция находит точку с наименьшим углом, то есть функцию с наибольшим косинусом угла. Для нахождения косинуса угла 3-2-1 используется функция GetCos(const double& x1, const double& y1, const double& x2, const double& y2, const double& x3, const double& y3). Параметры функции - координаты двух предыдущих точек оболочки и рассматриваемой точки. Рассматриваемый угол - внешний угол к углу 3-2-1, потому косинус необходимо умножить на -1.
Функция GetCos находит косинус угла 3-2-1 как частное скалярного произведения векторов 2-3 и 2-1 и их модулей, то есть, длин отрезков. Длины отрезков вычисляются с помощью функции FindDist(const double& x1, const double& y1, const double& x2, const double& y2). Параметры - координаты точек. Результат вычисляется как корень суммы квадратов разностей координат x и y точек.
Проход Джарвиса выполняется до тех пор, пока начальная и текущая точки оболочки не совпадут.
Наконец, из точек выпуклой оболочки будут исключены лишние точки (точки, лежащие на прямых между соседними точками) с помощью функции ElimPointsOnLines(double* X, double* Y, int* Envelope, int* Size), где входные параметры - указатели на массивы координат всех точек, указатель на массив индексов точек выпуклой оболочки и его размер.

- Вывод результатов и освобождение памяти
  Результаты прохода Джарвиса выводятся на экран. Также выводится на экран время работы программы и время выполнения самого прохода Джарвиса. Освобождается память, выделенная массивам.

Пример выходных данных программы:

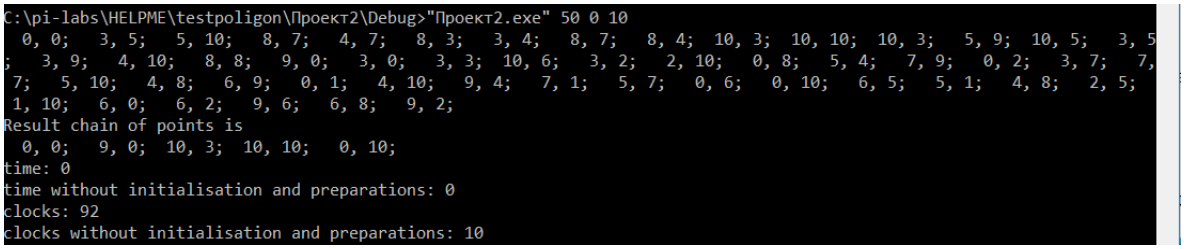


Рисунок 6. Пример выходных данных последовательной программной реализации.

5.2. Параллельная реализация с использованием OpenMP

Вторая программная реализация алгоритма Джарвиса содержит параллельную реализацию алгоритма с использованием OpenMP. Она также имеет отличия от первой программной реализации:

- Новый входной параметр - число потоков. При отсутствии заданного числа потоков OpenMP сам задает число потоков, на которых программа реализуется.
- Начальные данные алгоритма (множество случайных точек) больше не выводятся на экран, их вывод на экран затрудняет проведение экспериментов на больших объемах данных.
- Время теперь подсчитывается с помощью функций omp_get_wtime().
- Программная реализация включает в себя и последовательную, и параллельную версию алгоритма. Время их выполнения подсчитывается отдельно, результаты сравниваются с целью проверки корректности, выводятся дополнительные характеристики реализации алгоритма - ускорение и эффективность.

Сама параллельная реализация алгоритма была разработана по схеме распараллеливания, рассмотренной выше (см. Пункт 4, Схема распараллеливания). Вместо функций FindBLPoint и FindPWithMinAngle используются их распараллеленные аналоги FindBLPointParallel и FindPWithMinAngleParallel. Их распараллеливание осуществляется с помощью директив OpenMP: #pragma omp parallel и #pragma omp for.

Для достижения максимальной эффективности была найдена оптимальная стратегия schedule распределений итераций цикла for по потокам. В процессе поиска оптимальной

стратегии многократно запускалось выполнение второй программной реализации на 2 потоках при различных значениях параметров schedule: static, dynamic, guided.

Параметры запуска: 100000 точек, значения координат от 0 до 10000, chunk = 2500. Опыты проводились на ПК с 2-хядерным процессором, 4 лог. процессорами и 6 ГБ ОП, измерения повторялись 5 раз. Результаты опытов приведены в таблице:

Параметр schedule	static	dynamic	guided
Среднее время выполнения параллельной части программы (сек)	0,22	0,21	0,21

Результаты округлены до сотых. Среднее время выполнения колебалось; оно зависит от числа точек выпуклой оболочки, и это число случайно. В программной реализации используется параметр dynamic.

Пример выходных данных программы:

```
C:\pi-labs\MPI2019wip\OMPn\Debug>OMP.exe 100000 0 10000 2

2 threads are working
Results checked successfully
Result chain of points is
274, 0; 9738, 0; 9903, 5; 9967, 9; 9980, 28; 9991, 132; 10000, 503; 10000, 8631; 9999, 9920; 9993, 9951; 9985, 9968; 989
6, 9994; 9513, 10000; 2038, 10000; 233, 9999;  31, 9996;  16, 9956;   1, 9714;   0, 8474;   0, 197;   5, 157;  28, 9; 17
9, 2;
time: 0.666401
time without initialisation and preparations: 0.199116
time of sequential execution of the same task: 0.405282
Acceleration: 2.03512
Efficiency: 1.01756
```

Рисунок 7. Пример выходных данных программной реализации с OpenMP

5.3. Параллельная реализация с использованием TBB

Третья программная реализация алгоритма Джарвиса содержит параллельную реализацию алгоритма с использованием TBB. Она также имеет отличия от первой программной реализации:

- Новый входной параметр - число потоков. При отсутствии заданного числа потоков TBB сам задает число потоков, на которых программа реализуется. Число потоков устанавливается с помощью метода tbb::task_scheduler_init.initialize.

- Начальные данные алгоритма (множество случайных точек) больше не выводятся на экран, их вывод на экран затрудняет проведение экспериментов на больших объемах данных.
- Время теперь хранится в параметрах типа tbb::tick_count, фиксируется с помощью tbb::tick_count::now() и подсчитывается с помощью метода tbb::tick_count.seconds().
- Программная реализация включает в себя и последовательную, и параллельную версию алгоритма. Время их выполнения подсчитывается отдельно, результаты сравниваются с целью проверки корректности, выводятся дополнительные характеристики реализации алгоритма - ускорение и эффективность.

Сама параллельная реализация алгоритма была разработана по схеме распараллеливания, рассмотренной выше (см. Пункт 4, Схема распараллеливания). Вместо функций FindBLPoint и FindPWithMinAngle используются классы MinPointSearch и MinAngleSearch с реализованными функтором и методом редукции, а также метод tbb::parallel_reduce библиотеки TBB.

Пример выходных данных программы:

```
C:\pi-labs\mpi2019correctcopy\bin>churakov_s_jarvis_march_tbb.exe 100000 0 10000 2

4 threads are working by default
Number of threads was set to 2
Results checked successfully
Result chain of points is
292, 0; 9988, 0; 9995, 92; 9998, 318; 10000, 784; 10000, 9668; 9997, 9867; 9962, 9966; 9803, 9998; 8839, 10000; 2083, 10
000; 260, 9999; 170, 9998;  39, 9977;   7, 9928;   5, 9781;   0, 9043;   0, 1338;   8, 77;  14, 49;  28, 28;  58, 17;  7
4, 13; 133, 2;
time: 0.655742
time without initialisation and preparations: 0.204839
time of sequential execution of the same task: 0.385997
Acceleration: 1.88434
Efficiency: 0.942169
```

Рисунок 8. Пример выходных данных программной реализации с TBB

6. Проверка корректности

С целью проверки корректности последовательной реализации алгоритма Джарвиса была добавлена возможность вывести на экран графическое представление полученной выпуклой оболочки. Предполагается, что она запускается при малом размере интервала допустимых значений координат, при чем значения координат должны быть целыми числами.

Графическое представление - таблица размера s*s, где s = max – min, max и min - ограничения на значения координат. Таблица содержит данные типа char. Обозначения:

- ‘-’ - нет точки с данной координатой.
- ‘+’ - по данным координатам расположена точка, не входящая в выпуклую оболочку.
- ‘*’ - по данным координатам расположена точка, входящая в выпуклую оболочку.

Нижний левый элемент таблицы соответствует координатам (min, min); правый верхний - координатам (max, max).

На данном примере можно убедиться, что последовательная реализация корректно находит выпуклую оболочку множества точек:

```
C:\pi-labs\HELPME\testpoligon\Проект2\Debug>"Проект2.exe" 10 0 10
  5, 1;   9, 7;   3, 0;   0, 1;   7, 8;   0, 5;   2, 9;   7, 9;   4, 6;   5, 2;
Result chain of points is
  3, 0;   5, 1;   9, 7;   7, 9;   2, 9;   0, 5;   0, 1;
- - - - - - - - - -
- - * - - - - * - - -
- - - - - - - + - - -
- - - - - - - - - * -
- - - - + - - - - - -
* - - - - - - - - - -
- - - - - - - - - - -
- - - - - - - - - - -
- - - - - + - - - - -
* - - - - * - - - - -
- - - * - - - - - - -

time: 0
time without initialisation and preparations: 0
clocks: 43
clocks without initialisation and preparations: 36
```

Рисунок 8. Проверка корректности последовательной реализации.

Проверка корректности параллельных реализаций алгоритма осуществляется сравнением их результатов с результатами последовательной реализации.

7. Результаты экспериментов

Эксперименты проводились на ПК, характеристики приведены ниже.

- Windows 10, x64
- Процессор Intel Core i5-6200U, 2.3 ГГц с возможностью ускорения до 2.8 ГГц; 2 ядра, 4 лог. Процессора
- 6 ГБ ОП

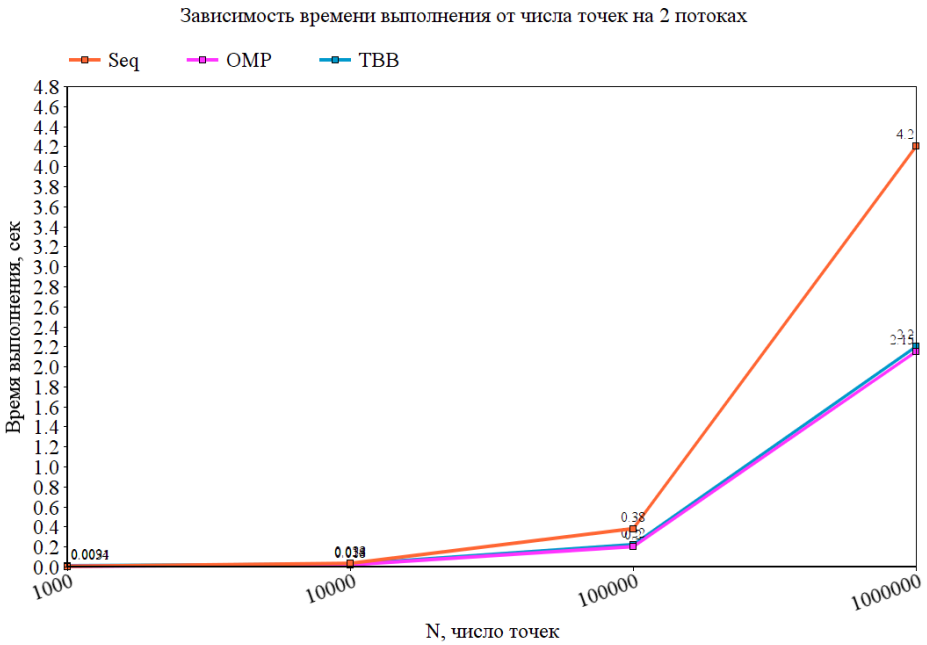
Исследование зависимости времени выполнения последовательной реализации от числа точек:

	Время выполнения, сек
N, число точек	Seq
1000	0,0031
10000	0,033
100000	0,38
1000000	4,2

Зависимость близка к линейной; сложность алгоритма линейно зависит от N.

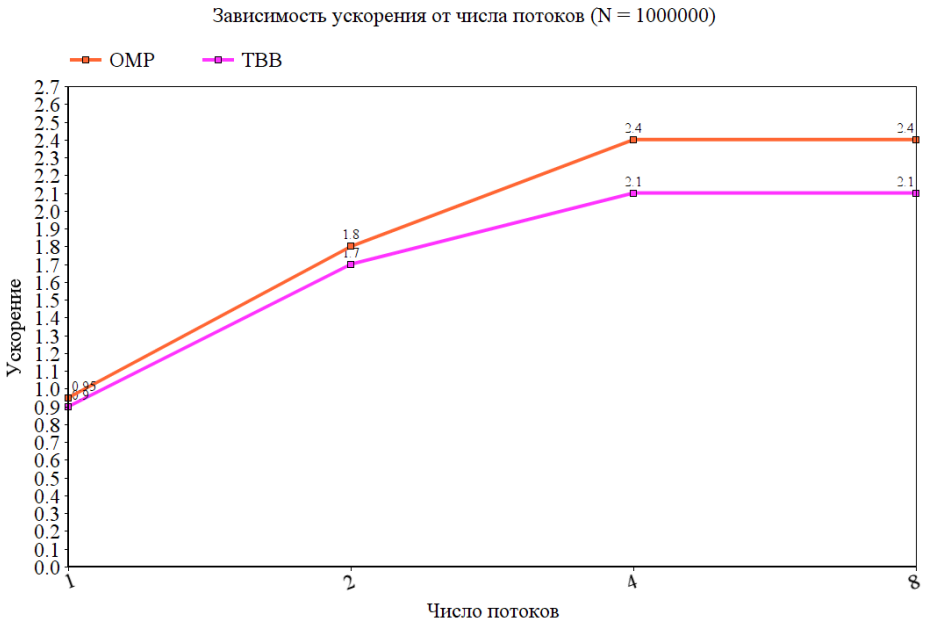
Исследование зависимости времени выполнения от числа точек на 2 потоках (последовательная реализация для сравнения, OMP, TBB):

	Время выполнения, сек		
N, число точек	Seq	OMP	TBB
1000	0,0031	0,0034	0,009
10000	0,033	0,18	0,024
100000	0,38	0,2	0,22
1000000	4,2	2,2	2,2



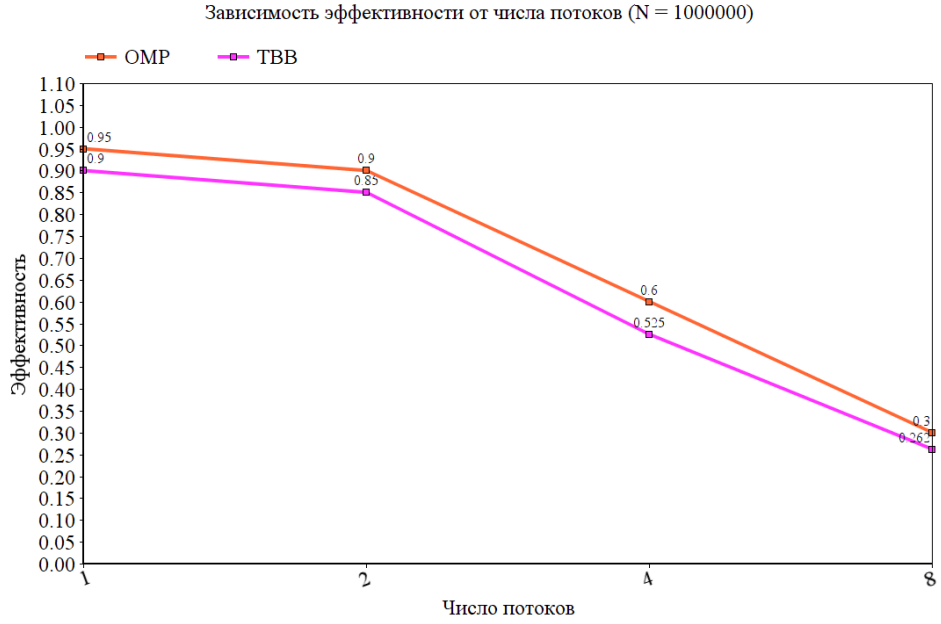
Исследование зависимости ускорения от числа потоков (N = 1000000)

	Ускорение			
Технология распараллеливания	1	2	4	8
OMP	0,95	1,8	2,4	2,4
TBB	0,9	1,7	2,1	2,1



Исследование зависимости эффективности от числа потоков

	Эффективность			
Технология распараллеливания	1	2	4	8
OMP	0,95	0,9	0,6	0,3
TBB	0,9	0,85	0,525	0,262



Когда используется больше 2 потоков, эффективность падает, а ускорение перестает увеличиваться линейно. Это объясняется тем, что выполняется программа на 2 ядрах. Наличие 4 логических ядер и использование технологии гиперпоточности (при кеш-промахах, неправильном предсказании условия, прочих простоях планировщик отдает процессор другому потоку) объясняет 10-30% рост ускорения на 4 и более потоках.

Результат удовлетворяет ожиданиям и на OMP, и на TBB реализациях, ведь показатель эффективности ~0,9 достаточно хорош. Часть времени теряется на распараллеливание задач, часть алгоритма выполняется последовательно, потому эффективность меньше 1.

Ускорение и эффективность реализаций, использующих OMP и TBB, близки, но можно заметить, что реализация на OMP немного эффективнее (особенно на малых N). OMP и TBB, очевидно, имеют различия между собой, и для этой конкретной задачи лучшим инструментом распараллеливания оказался OMP.

8. Заключение

Выводы, полученные, исходя из вышеизложенных исследований:

- Сложность алгоритма Джарвиса - O(n*M), где n - число точек всего множества, M - число точек выпуклой оболочки. При равномерном случайном распределении точек на плоскости (именно так задается множество точек в программной реализации) M мало (обычно, не больше 30), сложность алгоритма при этих условиях близка к линейной.
- Технологии OMP и TBB позволяют распараллелить программу, что позволяет в разы ускорить ее выполнение на многопроцессорных/многоядерных системах.
- Распараллеливание влечет накладные расходы на организацию параллелизма, и не все участки кода можно выполнять параллельно; для минимизации доли накладных расходов можно увеличивать объем обрабатываемых данных, но даже тогда не в каждой задаче достижима эффективность >= 1.
- Ускорение программы ограничено числом потоков и числом процессоров/ядер. Чтобы ускорить программу в ~4 раза, необходимо иметь 4 процессора/ядра и запустить программу на 4 потоках.
- Технология гиперпоточности может увеличить ускорение некоторых программ на 10-30%.
- Для данного алгоритма технологии OMP и TBB одинаково эффективны; OMP немного эффективнее.

9. Литература и источники

I) Википедия - свободная энциклопедия. Алгоритм Джарвиса [Электронный ресурс] https://ru.wikipedia.org/wiki/Алгоритм_Джарвиса

II) Википедия - свободная энциклопедия. Выпуклая оболочка [Электронный ресурс] https://ru.wikipedia.org/wiki/Выпуклая_оболочка

III) Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ = Introduction to Algorithms. — 2-е изд. — “Вильямс”, 2005. — 1296 с.

10. Приложения

10.1. Последовательная программная реализация

```
//  Copyright 2019 Churakov Sergey
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>
#include <cmath>
#define step 1000

void RandomizeArray(double* rarray, const int& size, const int& min = -50, const int& max =
50) {
   for (int i = 0; i < size; i++) {
      *(rarray + i) = std::rand() % (max - min + 1) + min;
   }
}


void PrintPoints(double* X, double* Y, const int& Size) {
   for (int i = 0; i < Size; i++) {
      std::cout << std::setw(3) << X[i] << ", " << Y[i] << "; ";
   }
}


void ReinitEqPoints(double* X, double* Y, const int& Size, const int& min = -50, const int&
max = 50) {
   for (int i = 0; i < Size - 1; i++)
      for (int j = i + 1; j < Size; j++) {
         if (X[i] == X[j] && Y[i] == Y[j]) {
            int flag = 1;
            while (flag) {
               flag = 0;
               X[i] = std::rand() % (max - min + 1) + min;
               Y[i] = std::rand() % (max - min + 1) + min;
               for (int k = 0; k < Size; k++) {
                  if (X[i] == X[k] && Y[i] == Y[k] && i != k) {
                     flag = 1;
                     break;
                  }
               }
            }
            break;
         }
```

```
        }
}

int FindBLPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
    for (int i = 1; i < Size; i++) {
        if (Y[i] < Y[pointInd]) {
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] < X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}

int FindTRPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
    for (int i = 1; i < Size; i++) {
        if (Y[i] > Y[pointInd]) {
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] > X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}

double FindDist(const double& x1, const double& y1, const double& x2, const double& y2) {
    return sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
}

double GetCos(const double& x1, const double& y1, const double& x2,
const double& y2, const double& x3, const double& y3) {
    return ((x2 - x1)*(x3 - x1) + (y2 - y1)*(y3 - y1)) / (FindDist(x1, y1, x2, y2)*FindDist(x1, y1,
x3, y3));
}


int FindPWithMinAngle(double* X, double* Y, const int& Size,
const double& x1, const double& y1, const double& x2, const double& y2) {
    double maxCos = -1.5;
    int NextPointInd = -1;
    double tmp;
```

```
    for (int i = 0; i < Size; i++) {
        if (!((X[i] == x1 && Y[i] == y1) || (X[i] == x2 && Y[i] == y2))) {
            if (((x2 - x1)*(Y[i] - y2) - (X[i] - x2)*(y2 - y1)) >= 0) {
                tmp = (-1)*GetCos(x2, y2, x1, y1, X[i], Y[i]);
                if (tmp > maxCos) {
                    maxCos = tmp;
                    NextPointInd = i;
                }
            }
        }
    }
    return NextPointInd;
}

void ElimPointsOnLines(double* X, double* Y, int* Envelope, int* Size) {
    int i = 1;
    while (i != *Size) {
        if (static_cast<double>(static_cast<double>(X[Envelope[i]] - X[Envelope[i - 1]])*
            static_cast<double>(Y[Envelope[i + 1]] - Y[Envelope[i]])) ==
            static_cast<double>(static_cast<double>(X[Envelope[i + 1]] - X[Envelope[i]])*
                static_cast<double>(Y[Envelope[i]] - Y[Envelope[i - 1]]))) {
            int j = 0;
            while (i + j < *Size) {
                Envelope[i + j] = Envelope[i + j + 1];
                j++;
            }
            *Size = *Size - 1;
            i--;
        }
        i++;
    }
}

int main(int argc, char* argv[]) {
    time_t times, time_part, timef;
    clock_t clocks, clock_part, clockf;
    times = time(NULL);
    clocks = clock();
    srand((unsigned int)time(NULL));
    int Size = 100;
    if (argc != 1 && argc != 2 && argc != 4)
        return 1;
    if (argc > 1) {
        Size = atoi(argv[1]);
        if (Size < 1)
            return 1;
    }
```

```
double* X_coord = new double[Size];
double* Y_coord = new double[Size];
if (argc == 4) {
   int minRand, maxRand;
   minRand = atoi(argv[2]);
   maxRand = atoi(argv[3]);
   if (maxRand < minRand)
      return 1;
   RandomizeArray(X_coord, Size, minRand, maxRand);
   RandomizeArray(Y_coord, Size, minRand, maxRand);
   //       if (Size >(maxRand - minRand + 1)*(maxRand - minRand + 1))
   //          return 1;
   //       ReinitEqPoints(X_coord, Y_coord, Size, minRand, maxRand);
} else {
   RandomizeArray(X_coord, Size);
   RandomizeArray(Y_coord, Size);
   //       if (Size > 101 * 101)
   //          return 1;
   //       ReinitEqPoints(X_coord, Y_coord, Size);
}
PrintPoints(X_coord, Y_coord, Size);
std::cout << std::endl;
time_part = time(NULL);
clock_part = clock();
if (Size == 1) {
   std::cout << "Result chain of points is a single point:" << std::endl;
   std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
} else {
   if (Size == 2) {
      std::cout << "Result chain of points is" << std::endl;
      std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
      std::cout << std::setw(3) << X_coord[1] << ", " << Y_coord[1] << "; ";
   } else {
      int dynsize = step;
      int* Envelope = static_cast<int*>(malloc(sizeof(int) * dynsize));
      int PNum = 1;
      int FirstPoint = FindBLPoint(X_coord, Y_coord, Size);
      Envelope[0] = FirstPoint;
      Envelope[1] = FindPWithMinAngle(X_coord, Y_coord, Size, X_coord[FirstPoint] - 1,
         Y_coord[FirstPoint], X_coord[FirstPoint], Y_coord[FirstPoint]);
      while (Envelope[PNum] != FirstPoint) {
         PNum++;
         if (PNum == dynsize) {
            dynsize += step;
            Envelope = static_cast<int*>(realloc(Envelope, sizeof(int) * (dynsize)));
         }
```

```
            Envelope[PNum] = FindPWithMinAngle(X_coord, Y_coord, Size,
X_coord[Envelope[PNum - 2]],
                Y_coord[Envelope[PNum - 2]], X_coord[Envelope[PNum - 1]],
Y_coord[Envelope[PNum - 1]]);
          }
          ElimPointsOnLines(X_coord, Y_coord, Envelope, &PNum);
          if (PNum < 3) {
            std::cout << "Result chain of points is a line" << std::endl;
            int point;
            point = FindBLPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
            point = FindTRPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
          } else {
            std::cout << "Result chain of points is" << std::endl;
            for (int i = 0; i < PNum; i++) {
              std::cout << std::setw(3) << X_coord[Envelope[i]] << ", " << Y_coord[Envelope[i]]
<< "; ";
            }
          }
          delete[] Envelope;
        }
    }
    timef = time(NULL);
    clockf = clock();
    //    times =
    std::cout << std::endl;
    std::cout << "time: " << difftime(timef, times) << std::endl;
    std::cout << "time without initialisation and preparations: " << difftime(time_part, timef) <<
std::endl;
    std::cout << "clocks: " << clockf - clocks << std::endl;
    std::cout << "clocks without initialisation and preparations: " << clockf - clock_part <<
std::endl;
    delete[] X_coord;
    delete[] Y_coord;
    return 0;
}
```

10.2. Программная реализация с использованием OpenMP

```
//  Copyright 2019 Churakov Sergey
#include <omp.h>
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>
#include <cmath>
#define step 1000
#define chunk 2500


void RandomizeArray(double* rarray, const int& size, const int& min = -50, const int& max =
50) {
   for (int i = 0; i < size; i++) {
      int part1 = std::rand();
      int part2 = part1 << (sizeof(int) );
      part2 |= std::rand();
      *(rarray + i) = part2 % static_cast<int>(max - min + 1) + min;
   }
}



void PrintPoints(double* X, double* Y, const int& Size) {
   for (int i = 0; i < Size; i++) {
      std::cout << std::setw(3) << X[i] << ", " << Y[i] << "; ";
   }
}



void ReinitEqPoints(double* X, double* Y, const int& Size, const int& min = -50, const int&
max = 50) {
   for (int i = 0; i < Size - 1; i++)
      for (int j = i + 1; j < Size; j++) {
         if (X[i] == X[j] && Y[i] == Y[j]) {
            int flag = 1;
```

```
            while (flag) {
                flag = 0;
                X[i] = std::rand() % (max - min + 1) + min;
                Y[i] = std::rand() % (max - min + 1) + min;
                for (int k = 0; k < Size; k++) {
                    if (X[i] == X[k] && Y[i] == Y[k] && i != k) {
                        flag = 1;
                        break;
                    }
                }
            }
            break;
        }
    }
}

int FindBLPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
    for (int i = 1; i < Size; i++) {
        if (Y[i] < Y[pointInd]) {
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] < X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}


int FindBLPointParallel(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
```

```
    int i;
    int threads = omp_get_max_threads();
    int* pointIndOwn = new int[threads];
    for (int i = 0; i < threads; i++) {
        pointIndOwn[i] = 0;
    }
    #pragma omp parallel private(i)
    {
        int cur_thread = omp_get_thread_num();
        #pragma omp for schedule(dynamic, chunk)
        for (i = 0; i < Size; i++) {
            if (Y[i] < Y[pointIndOwn[cur_thread]]) {
                pointIndOwn[cur_thread] = i;
            } else {
                if (Y[i] == Y[pointIndOwn[cur_thread]] && X[i] < X[pointIndOwn[cur_thread]])
                    pointIndOwn[cur_thread] = i;
            }
        }
    }
    pointInd = pointIndOwn[0];
    for (int i = 1; i < threads; i++) {
        if (Y[pointIndOwn[i]] < Y[pointInd]) {
            pointInd = pointIndOwn[i];
        } else {
            if (Y[pointIndOwn[i]] == Y[pointInd] && X[pointIndOwn[i]] < X[pointInd])
                pointInd = pointIndOwn[i];
        }
    }
    return pointInd;
}

int FindTRPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
```

```
    for (int i = 1; i < Size; i++) {
        if (Y[i] > Y[pointInd]) {
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] > X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}

double FindDist(const double& x1, const double& y1, const double& x2, const double& y2) {
    return sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
}

double GetCos(const double& x1, const double& y1, const double& x2,
    const double& y2, const double& x3, const double& y3) {
    return ((x2 - x1)*(x3 - x1) + (y2 - y1)*(y3 - y1)) / (FindDist(x1, y1, x2, y2)*FindDist(x1, y1,
x3, y3));
}



int FindPWithMinAngle(double* X, double* Y, const int& Size,
    const double& x1, const double& y1, const double& x2, const double& y2) {
    double maxCos = -1.5;
    int NextPointInd = -1;
    double tmp;
    for (int i = 0; i < Size; i++) {
        if (!((X[i] == x1 && Y[i] == y1) || (X[i] == x2 && Y[i] == y2))) {
            if (((x2 - x1)*(Y[i] - y2) - (X[i] - x2)*(y2 - y1)) >= 0) {
                tmp = (-1)*GetCos(x2, y2, x1, y1, X[i], Y[i]);
                if (tmp > maxCos) {
                    maxCos = tmp;
                    NextPointInd = i;
                }
```

```
            }
        }
    }
    return NextPointInd;
}

int FindPWithMinAngleParallel(double* X, double* Y, const int& Size,
    const double &x1, const double& y1, const double& x2, const double& y2) {
    int NextPointInd = -1;
    int i;
    double tmp;
    int threads = omp_get_max_threads();
    int* NextPointIndOwn = new int[threads];
    double* maxCosOwn = new double[threads];
    for (int i = 0; i < threads; i++) {
        NextPointIndOwn[i] = -1;
        maxCosOwn[i] = -1.5;
    }
    #pragma omp parallel private(i, tmp)
    {
        int cur_thread = omp_get_thread_num();
        #pragma omp for schedule(dynamic, chunk)
        for (i = 0; i < Size; i++) {
            if (!((X[i] == x1 && Y[i] == y1) || (X[i] == x2 && Y[i] == y2))) {
                if (((x2 - x1)*(Y[i] - y2) - (X[i] - x2)*(y2 - y1)) >= 0) {
                    tmp = (-1)*GetCos(x2, y2, x1, y1, X[i], Y[i]);
                    if (tmp > maxCosOwn[cur_thread]) {
                        maxCosOwn[cur_thread] = tmp;
                        NextPointIndOwn[cur_thread] = i;
                    }
                }
            }
        }
    }
```

```
    tmp = maxCosOwn[0];
    NextPointInd = NextPointIndOwn[0];
    for (int i = 1; i < threads; i++) {
        if (tmp < maxCosOwn[i]) {
            tmp = maxCosOwn[i];
            NextPointInd = NextPointIndOwn[i];
        }
    }
    return NextPointInd;
}

void ElimPointsOnLines(double* X, double* Y, int* Envelope, int* Size) {
    int i = 1;
    while (i != *Size) {
        if (static_cast<double>(static_cast<double>(X[Envelope[i]] - X[Envelope[i - 1]])*
            static_cast<double>(Y[Envelope[i + 1]] - Y[Envelope[i]])) ==
            static_cast<double>(static_cast<double>(X[Envelope[i + 1]] - X[Envelope[i]])*
                static_cast<double>(Y[Envelope[i]] - Y[Envelope[i - 1]]))) {
            int j = 0;
            while (i + j < *Size) {
                Envelope[i + j] = Envelope[i + j + 1];
                j++;
            }
            *Size = *Size - 1;
            i--;
        }
        i++;
    }
}

int main(int argc, char* argv[]) {
    double times = 0, time_part = 0, time_part_fin = 0, timef = 0;
    double time_part_seq = 0, time_part_seq_fin = 0;
    times = omp_get_wtime();
```

```
srand((unsigned int)time(NULL));
int Size = 100;
if (argc < 1 || argc > 5)
   return 1;
if (argc > 1) {
   Size = atol(argv[1]);
   if (Size < 1)
      return 1;
}
double* X_coord = new double[Size];
double* Y_coord = new double[Size];
if (argc == 4 || argc == 5) {
   int minRand, maxRand;
   minRand = atol(argv[2]);
   maxRand = atol(argv[3]);
   if (maxRand < minRand)
      return 1;
   RandomizeArray(X_coord, Size, minRand, maxRand);
   RandomizeArray(Y_coord, Size, minRand, maxRand);
   //       if (Size >(maxRand - minRand + 1)*(maxRand - minRand + 1))
   //          return 1;
   //       ReinitEqPoints(X_coord, Y_coord, Size, minRand, maxRand);
} else {
   RandomizeArray(X_coord, Size);
   RandomizeArray(Y_coord, Size);
   //       if (Size > 101 * 101)
   //          return 1;
   //       ReinitEqPoints(X_coord, Y_coord, Size);
}
if (argc == 3) {
   int numt = atoi(argv[2]);
   if (numt > 0 && numt < 65) {
      omp_set_num_threads(numt);
   } else {
```

```
            return 1;
        }
    }
    if (argc == 5) {
        int numt = atoi(argv[4]);
        if (numt > 0 && numt < 65) {
            omp_set_num_threads(numt);
        } else {
            return 1;
        }
    }
//  PrintPoints(X_coord, Y_coord, Size);
    std::cout << std::endl;
    std::cout << omp_get_max_threads() << " threads are working" << std::endl;
    if (Size == 1) {
        std::cout << "Result chain of points is a single point:" << std::endl;
        std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
    } else {
        if (Size == 2) {
            std::cout << "Result chain of points is" << std::endl;
            std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
            std::cout << std::setw(3) << X_coord[1] << ", " << Y_coord[1] << "; ";
        } else {
            int dynsize = step;
            int* Envelope = static_cast<int*>(malloc(sizeof(int) * dynsize));
            int PNum = 1;
            time_part = omp_get_wtime();
            int FirstPoint = FindBLPointParallel(X_coord, Y_coord, Size);
            Envelope[0] = FirstPoint;
            Envelope[1] = FindPWithMinAngleParallel(X_coord, Y_coord, Size, X_coord[FirstPoint]
- 1,
                Y_coord[FirstPoint], X_coord[FirstPoint], Y_coord[FirstPoint]);
            if (Envelope[1] == -1) {
                std::cout << "Result may be a single point or error" << std::endl;
                std::cout << "Result point may be: ";
```

```
            std::cout << X_coord[Envelope[0]] << " " << Y_coord[Envelope[0]] << std::endl;
            free(Envelope);
            delete[] X_coord;
            delete[] Y_coord;
            return 1;
        }
        while (Envelope[PNum] != FirstPoint && ((X_coord[FirstPoint] !=
X_coord[Envelope[PNum]])
            ||(Y_coord[FirstPoint] != Y_coord[Envelope[PNum]]))) {
            PNum++;
            if (PNum == dynsize) {
                dynsize += step;
                Envelope = static_cast<int*>(realloc(Envelope, sizeof(int) * (dynsize)));
            }
            Envelope[PNum] = FindPWithMinAngleParallel(X_coord, Y_coord, Size,
X_coord[Envelope[PNum - 2]],
                Y_coord[Envelope[PNum - 2]], X_coord[Envelope[PNum - 1]],
Y_coord[Envelope[PNum - 1]]);
        }
        ElimPointsOnLines(X_coord, Y_coord, Envelope, &PNum);
        time_part_fin = omp_get_wtime();
        dynsize = step;
//      Sequential implementation
        int* EnvelopeForCheck = static_cast<int*>(malloc(sizeof(int) * dynsize));
        int PNumForCheck = 1;
        int Correct = 1;
        time_part_seq = omp_get_wtime();
        int FirstPointForCheck = FindBLPoint(X_coord, Y_coord, Size);
        EnvelopeForCheck[0] = FirstPointForCheck;
        EnvelopeForCheck[1] = FindPWithMinAngle(X_coord, Y_coord, Size,
X_coord[FirstPointForCheck] - 1,
            Y_coord[FirstPointForCheck], X_coord[FirstPointForCheck],
Y_coord[FirstPointForCheck]);
        if (Envelope[1] == -1) {
            std::cout << "Result may be a single point or error" << std::endl;
            std::cout << "Result point may be: ";
```

```
            std::cout << X_coord[Envelope[0]] << " " << Y_coord[Envelope[0]] << std::endl;
            free(EnvelopeForCheck);
            free(Envelope);
            delete[] X_coord;
            delete[] Y_coord;
            return 1;
        }
        while (EnvelopeForCheck[PNumForCheck] != FirstPointForCheck && Correct &&
            ((X_coord[FirstPointForCheck] != X_coord[EnvelopeForCheck[PNumForCheck]])
            || (Y_coord[FirstPointForCheck] != Y_coord[EnvelopeForCheck[PNumForCheck]]))) {
            PNumForCheck++;
            if (PNumForCheck == dynsize) {
                dynsize += step;
                EnvelopeForCheck = static_cast<int*>(realloc(EnvelopeForCheck, sizeof(int) *
(dynsize)));
            }
            EnvelopeForCheck[PNumForCheck] = FindPWithMinAngle(X_coord, Y_coord, Size,
                X_coord[EnvelopeForCheck[PNumForCheck - 2]],
                Y_coord[EnvelopeForCheck[PNumForCheck - 2]],
                X_coord[EnvelopeForCheck[PNumForCheck - 1]],
                Y_coord[EnvelopeForCheck[PNumForCheck - 1]]);
        }
        ElimPointsOnLines(X_coord, Y_coord, EnvelopeForCheck, &PNumForCheck);
        time_part_seq_fin = omp_get_wtime();
        if (PNum != PNumForCheck) {
            Correct = 0;
        } else {
            for (int i = 0; i < PNum; i++)
                if (X_coord[Envelope[i]] != X_coord[EnvelopeForCheck[i]] &&
                    Y_coord[Envelope[i]] != Y_coord[EnvelopeForCheck[i]]) {
                    Correct = 0;
                    break;
                }
        }
        if (Correct)
```

```
            std::cout << "Results checked successfully" << std::endl;
        else
            std::cout << "Results dont match" << std::endl;
        if (PNum < 3) {
            std::cout << "Result chain of points is a line" << std::endl;
            int point;
            point = FindBLPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
            point = FindTRPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
        } else {
            std::cout << "Result chain of points is" << std::endl;
            for (int i = 0; i < PNum; i++) {
                std::cout << std::setw(3) << X_coord[Envelope[i]] << ", " << Y_coord[Envelope[i]]
<< "; ";
            }
        }
        timef = omp_get_wtime();
        std::cout << std::endl;
        std::cout << "time: " << timef - times << std::endl;
        std::cout << "time without initialisation and preparations: " << time_part_fin - time_part
<< std::endl;
        std::cout << "time of sequential execution of the same task: " << time_part_seq_fin -
time_part_seq;
        std::cout << std::endl;
        double acc = (time_part_seq_fin - time_part_seq) / (time_part_seq - time_part);
        std::cout << "Acceleration: " << acc << std::endl;
        std::cout << "Efficiency: " << acc / omp_get_max_threads() << std::endl;
        free(Envelope);
        delete[] EnvelopeForCheck;
      }
   }
   delete[] X_coord;
   delete[] Y_coord;
   return 0;
```

```
}
```

10.3. Программная реализация с использованием TBB

```
//  Copyright 2019 Churakov Sergey
#include <tbb/tbb.h>
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>
#include <cmath>
#define step 1000

void RandomizeArray(double* rarray, const int& size, const int& min = -50, const int& max =
50) {
   for (int i = 0; i < size; i++) {
      int part1 = std::rand();
      int part2 = part1 << (sizeof(int)*4);
      part2 |= std::rand();
      *(rarray + i) = part2 % static_cast<int>(max - min + 1) + min;
   }
}



void PrintPoints(double* X, double* Y, const int& Size) {
   for (int i = 0; i < Size; i++) {
      std::cout << std::setw(3) << X[i] << ", " << Y[i] << "; ";
   }
}



void ReinitEqPoints(double* X, double* Y, const int& Size, const int& min = -50, const int&
max = 50) {
   for (int i = 0; i < Size - 1; i++)
      for (int j = i + 1; j < Size; j++) {
         if (X[i] == X[j] && Y[i] == Y[j]) {
            int flag = 1;
            while (flag) {
```

```
                flag = 0;
                X[i] = std::rand() % (max - min + 1) + min;
                Y[i] = std::rand() % (max - min + 1) + min;
                for (int k = 0; k < Size; k++) {
                    if (X[i] == X[k] && Y[i] == Y[k] && i != k) {
                        flag = 1;
                        break;
                    }
                }
            }
            break;
        }
    }
}

int FindBLPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
    for (int i = 1; i < Size; i++) {
        if (Y[i] < Y[pointInd]) {
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] < X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}

int FindTRPoint(double* X, double* Y, const int& Size) {
    if (Size < 1) { return -1; }
    int pointInd = 0;
    for (int i = 1; i < Size; i++) {
        if (Y[i] > Y[pointInd]) {
```

```
            pointInd = i;
        } else {
            if (Y[i] == Y[pointInd] && X[i] > X[pointInd])
                pointInd = i;
        }
    }
    return pointInd;
}

double FindDist(const double& x1, const double& y1, const double& x2, const double& y2) {
    return sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
}

double GetCos(const double& x1, const double& y1, const double& x2,
    const double& y2, const double& x3, const double& y3) {
    return ((x2 - x1)*(x3 - x1) + (y2 - y1)*(y3 - y1)) / (FindDist(x1, y1, x2, y2)*FindDist(x1, y1,
x3, y3));
}



int FindPWithMinAngle(double* X, double* Y, const int& Size,
    const double& x1, const double& y1, const double& x2, const double& y2) {
    double maxCos = -1.5;
    int NextPointInd = -1;
    double tmp;
    for (int i = 0; i < Size; i++) {
        if (!((X[i] == x1 && Y[i] == y1) || (X[i] == x2 && Y[i] == y2))) {
            if (((x2 - x1)*(Y[i] - y2) - (X[i] - x2)*(y2 - y1)) >= 0) {
                tmp = (-1)*GetCos(x2, y2, x1, y1, X[i], Y[i]);
                if (tmp > maxCos) {
                    maxCos = tmp;
                    NextPointInd = i;
                }
            }
        }
```

```
    }
    return NextPointInd;
}

void ElimPointsOnLines(double* X, double* Y, int* Envelope, int* Size) {
    int i = 1;
    while (i != *Size) {
        if (static_cast<double>(static_cast<double>(X[Envelope[i]] - X[Envelope[i - 1]])*
            static_cast<double>(Y[Envelope[i + 1]] - Y[Envelope[i]])) ==
            static_cast<double>(static_cast<double>(X[Envelope[i + 1]] - X[Envelope[i]])*
                static_cast<double>(Y[Envelope[i]] - Y[Envelope[i - 1]]))) {
            int j = 0;
            while (i + j < *Size) {
                Envelope[i + j] = Envelope[i + j + 1];
                j++;
            }
            *Size = *Size - 1;
            i--;
        }
        i++;
    }
}

class BLPointSearch {
private:
    double* X;
    double* Y;
    int Result;

public:
    explicit BLPointSearch(double* X_coord, double* Y_coord) {
        X = X_coord;
        Y = Y_coord;
        Result = 0;
```

```
    }

    BLPointSearch(const BLPointSearch& m, tbb::split) {
        X = m.X;
        Y = m.Y;
        Result = 0;
    }

    void operator()(const tbb::blocked_range<int>& r) {
        int begin = r.begin();
        int end = r.end();
        for (int i = begin; i < end; i++) {
            if (Y[i] < Y[Result]) {
                Result = i;
            } else {
                if (Y[i] == Y[Result] && X[i] < X[Result])
                    Result = i;
            }
        }
    }

    void join(const BLPointSearch& m) {
        if (Y[m.Result] < Y[Result]) {
            Result = m.Result;
        } else {
            if (Y[m.Result] == Y[Result] && X[m.Result] < X[Result])
                Result = m.Result;
        }
    }

    int getResult() {
        return Result;
    }
};
```

```
class MinAngleSearch {
private:
    double* X;
    double* Y;
    double x1, y1, x2, y2;
    int Result;
    double maxCos;

public:
    explicit MinAngleSearch(double* X_coord, double* Y_coord, const double& x1,
        const double& y1, const double& x2, const double& y2) {
        X = X_coord;
        Y = Y_coord;
        this->x1 = x1;
        this->y1 = y1;
        this->x2 = x2;
        this->y2 = y2;
        Result = -1;
        maxCos = -1.5;
    }

    MinAngleSearch(const MinAngleSearch& m, tbb::split) {
        X = m.X;
        Y = m.Y;
        x1 = m.x1;
        y1 = m.y1;
        x2 = m.x2;
        y2 = m.y2;
        Result = -1;
        maxCos = -1.5;
    }
```

```
void join(const MinAngleSearch& m) {
    if (m.maxCos > maxCos) {
        Result = m.Result;
        maxCos = m.maxCos;
    }
}

void operator()(const tbb::blocked_range<int>& r) {
    int begin = r.begin();
    int end = r.end();
    double tmp;
    int tempResult = Result;
    double tempmaxCos = maxCos;
    for (int i = begin; i < end; i++) {
        if (!((X[i] == x1 && Y[i] == y1) || (X[i] == x2 && Y[i] == y2))) {
            if (((x2 - x1)*(Y[i] - y2) - (X[i] - x2)*(y2 - y1)) >= 0) {
                tmp = (-1)*GetCos(x2, y2, x1, y1, X[i], Y[i]);
                if (tmp > tempmaxCos) {
                    tempmaxCos = tmp;
                    tempResult = i;
                }
            }
        }
    }
    Result = tempResult;
    maxCos = tempmaxCos;
}

void reinit(const double& x1, const double& y1,
    const double& x2, const double& y2) {
    this->x1 = x1;
    this->y1 = y1;
    this->x2 = x2;
    this->y2 = y2;
```

```
        Result = -1;
        maxCos = -1.5;
    }

    int getResult() {
        return Result;
    }
};

int main(int argc, char* argv[]) {
    tbb::task_scheduler_init init(tbb::task_scheduler_init::automatic);
    tbb::tick_count times, time_part, time_part_fin, timef;
    tbb::tick_count time_part_seq, time_part_seq_fin;
    times = tbb::tick_count::now();
    srand((unsigned int)time(NULL));
    int Size = 100;
    int numt = tbb::task_scheduler_init::default_num_threads();
    if (argc < 1 || argc > 5)
        return 1;
    if (argc > 1) {
        Size = atol(argv[1]);
        if (Size < 1)
            return 1;
    }
    double* X_coord = new double[Size];
    double* Y_coord = new double[Size];
    std::cout << std::endl;
    std::cout << tbb::task_scheduler_init::default_num_threads()
        << " threads are working by default" << std::endl;
    if (argc == 4 || argc == 5) {
        int minRand, maxRand;
        minRand = atol(argv[2]);
        maxRand = atol(argv[3]);
        if (maxRand < minRand)
```

```
            return 1;
        RandomizeArray(X_coord, Size, minRand, maxRand);
        RandomizeArray(Y_coord, Size, minRand, maxRand);
    } else {
        RandomizeArray(X_coord, Size);
        RandomizeArray(Y_coord, Size);
    }
    if (argc == 3) {
        numt = atoi(argv[2]);
        if (numt > 0 && numt < 65) {
            init.terminate();
            init.initialize(numt);
            std::cout << "Number of threads was set to " << numt << std::endl;
        } else {
            return 1;
        }
    }
    if (argc == 5) {
        numt = atoi(argv[4]);
        if (numt > 0 && numt < 65) {
            init.terminate();
            init.initialize(numt);
            std::cout << "Number of threads was set to " << numt << std::endl;
        } else {
            return 1;
        }
    }
    if (Size == 1) {
        std::cout << "Result chain of points is a single point:" << std::endl;
        std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
    } else {
        if (Size == 2) {
            std::cout << "Result chain of points is" << std::endl;
            std::cout << std::setw(3) << X_coord[0] << ", " << Y_coord[0] << "; ";
```

```
        std::cout << std::setw(3) << X_coord[1] << ", " << Y_coord[1] << "; ";
    } else {
        int dynsize = step;
        int* Envelope = static_cast<int*>(malloc(sizeof(int) * dynsize));
        int PNum = 1;
        time_part = tbb::tick_count::now();
        BLPointSearch searchBLP(X_coord, Y_coord);
        tbb::parallel_reduce(tbb::blocked_range<int>(0, Size), searchBLP);
        int FirstPoint = searchBLP.getResult();
        Envelope[0] = FirstPoint;
        MinAngleSearch jarvis(X_coord, Y_coord, X_coord[FirstPoint] - 1,
            Y_coord[FirstPoint], X_coord[FirstPoint], Y_coord[FirstPoint]);
        tbb::parallel_reduce(tbb::blocked_range<int>(0, Size), jarvis);
        Envelope[1] = jarvis.getResult();
        if (Envelope[1] == -1) {
            std::cout << "Result may be a single point or error" << std::endl;
            std::cout << "Result point may be: ";
            std::cout << X_coord[Envelope[0]] << " " << Y_coord[Envelope[0]] << std::endl;
            free(Envelope);
            delete[] X_coord;
            delete[] Y_coord;
            return 1;
        }
        while (Envelope[PNum] != FirstPoint && ((X_coord[FirstPoint] !=
X_coord[Envelope[PNum]])
            ||(Y_coord[FirstPoint] != Y_coord[Envelope[PNum]]))) {
            PNum++;
            if (PNum == dynsize) {
                dynsize += step;
                Envelope = static_cast<int*>(realloc(Envelope, sizeof(int) * (dynsize)));
            }
            jarvis.reinit(X_coord[Envelope[PNum - 2]], Y_coord[Envelope[PNum - 2]],
                X_coord[Envelope[PNum - 1]], Y_coord[Envelope[PNum - 1]]);
            tbb::parallel_reduce(tbb::blocked_range<int>(0, Size), jarvis);
            Envelope[PNum] = jarvis.getResult();
```

```
        }
        ElimPointsOnLines(X_coord, Y_coord, Envelope, &PNum);
        time_part_fin = tbb::tick_count::now();
        dynsize = step;
//      Sequential implementation
        int* EnvelopeForCheck = static_cast<int*>(malloc(sizeof(int) * dynsize));
        int PNumForCheck = 1;
        int Correct = 1;
        time_part_seq = tbb::tick_count::now();
        int FirstPointForCheck = FindBLPoint(X_coord, Y_coord, Size);
        EnvelopeForCheck[0] = FirstPointForCheck;
        EnvelopeForCheck[1] = FindPWithMinAngle(X_coord, Y_coord, Size,
X_coord[FirstPointForCheck] - 1,
            Y_coord[FirstPointForCheck], X_coord[FirstPointForCheck],
Y_coord[FirstPointForCheck]);
        if (Envelope[1] == -1) {
            std::cout << "Result may be a single point or error" << std::endl;
            std::cout << "Result point may be: ";
            std::cout << X_coord[Envelope[0]] << " " << Y_coord[Envelope[0]] << std::endl;
            free(EnvelopeForCheck);
            free(Envelope);
            delete[] X_coord;
            delete[] Y_coord;
            return 1;
        }
        while (EnvelopeForCheck[PNumForCheck] != FirstPointForCheck && Correct &&
            ((X_coord[FirstPointForCheck] != X_coord[EnvelopeForCheck[PNumForCheck]])
            || (Y_coord[FirstPointForCheck] != Y_coord[EnvelopeForCheck[PNumForCheck]]))) {
            PNumForCheck++;
            if (PNumForCheck == dynsize) {
                dynsize += step;
                EnvelopeForCheck = static_cast<int*>(realloc(EnvelopeForCheck, sizeof(int) *
(dynsize)));
            }
            EnvelopeForCheck[PNumForCheck] = FindPWithMinAngle(X_coord, Y_coord, Size,
```

```
            X_coord[EnvelopeForCheck[PNumForCheck - 2]],
            Y_coord[EnvelopeForCheck[PNumForCheck - 2]],
            X_coord[EnvelopeForCheck[PNumForCheck - 1]],
            Y_coord[EnvelopeForCheck[PNumForCheck - 1]]);
        }
        ElimPointsOnLines(X_coord, Y_coord, EnvelopeForCheck, &PNumForCheck);
        time_part_seq_fin = tbb::tick_count::now();
        if (PNum != PNumForCheck) {
            Correct = 0;
        } else {
            for (int i = 0; i < PNum; i++)
                if (X_coord[Envelope[i]] != X_coord[EnvelopeForCheck[i]] &&
                    Y_coord[Envelope[i]] != Y_coord[EnvelopeForCheck[i]]) {
                    Correct = 0;
                    break;
                }
        }
        if (Correct)
            std::cout << "Results checked successfully" << std::endl;
        else
            std::cout << "Results dont match" << std::endl;
        if (PNum < 3) {
            std::cout << "Result chain of points is a line" << std::endl;
            int point;
            point = FindBLPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
            point = FindTRPoint(X_coord, Y_coord, Size);
            std::cout << std::setw(3) << X_coord[point] << ", " << Y_coord[point] << "; ";
        } else {
            std::cout << "Result chain of points is" << std::endl;
            for (int i = 0; i < PNum; i++) {
                std::cout << std::setw(3) << X_coord[Envelope[i]] << ", " << Y_coord[Envelope[i]]
<< "; ";
            }
        }
```

```
            timef = tbb::tick_count::now();
            std::cout << std::endl;
            std::cout << "time: " << (timef - times).seconds() << std::endl;
            std::cout << "time without initialisation and preparations: " <<
               (time_part_fin - time_part).seconds() << std::endl;
            std::cout << "time of sequential execution of the same task: " <<
               (time_part_seq_fin - time_part_seq).seconds();
            std::cout << std::endl;
            double acc = (time_part_seq_fin - time_part_seq).seconds() /
               (time_part_seq - time_part).seconds();
            std::cout << "Acceleration: " << acc << std::endl;
            std::cout << "Efficiency: " << acc / numt << std::endl;
            free(Envelope);
            delete[] EnvelopeForCheck;
         }
      }
      delete[] X_coord;
      delete[] Y_coord;
      return 0;
   }
```