МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра программной инженерии**

Направление подготовки: «Программная инженерия»

Профиль подготовки: «Разработка программно-информационных систем»

**ОТЧЕТ**

по лабораторной работе №1   
по дисциплине «Параллельное программирование»

тема:

**«Построение выпуклой оболочки – проход Джарвиса»**

**Выполнил:** студент группы 381508

Бакалин Андрей Евгеньевич

**Проверил:**

Доцент, к.т.н. Сысоев А. В.

Нижний Новгород  
2018

Введение

Вычислительная геометрия – это раздел информатики, изучающий алгоритмы, предназначенные для решения геометрических задач. В современных инженерных и математических расчетах вычислительная геометрия, в числе других областей знаний, применяется в машинной графике, в робототехнике, при автоматизированном проектировании и т.д.

Построение выпуклой оболочки – одна из задач вычислительной геометрии. В данной работе будет рассмотрен один из алгоритмов ее построения – алгоритм Джарвиса.

**Постановка задачи**

Необходимо реализовать алгоритм Джарвиса построения выпуклой оболочки в последовательной и параллельных версиях (с использование технологий OpenMP и TBB). Подготовить задачу для автоматизированной проверки.

Для выполнения лабораторной работы необходимо:

1. Изучить метод решения.
2. Разработать тестовую версию, включающую в себя:

- решение задачи последовательным алгоритмом.

- программу для генерации набора тестовых данных и их сохранения в бинарные файлы.

- программа для проверки корректности параллельных версий.

- набор тестов для проверки корректности параллельных версий.

1. Разработать OpenMP версию.
2. Разработать TBB версию.
3. Провести эксперименты на 1, 2, 4, 8 потоках.
4. Проанализировать полученные результаты.

**Метод решения**

**Суть алгоритма**

Пусть дано множество точек P = {p1, p2, … , pn}. В качестве начальной берется самая левая нижняя точка p1 (ее можно найти за O(n) обычным проходом по всем точкам), она точно является вершиной выпуклой оболочки. Следующей точкой (p2) берем такую точку, которая имеет наименьший положительный полярный угол относительно точки p1 как начала координат. После этого для каждой точки pi (2<i<=|P|) против часовой стрелки ищется такая точка pi+1, путем нахождения за O(n) среди оставшихся точек (+ самая левая нижняя), в которой будет образовываться наибольший угол между прямыми pi-1pi и pipi+1. Он будет следующей вершиной выпуклой оболочки. Нахождение вершин выпуклой оболочки продолжается до тех пор, пока pi+1 не будет равно p1. В тот момент, когда следующая точка в выпуклой оболочке совпала с первой, алгоритм останавливается — выпуклая оболочка построена.

При этом сам угол искать не надо, достаточно найти только косинус, пользуясь формулой скалярного произведения.

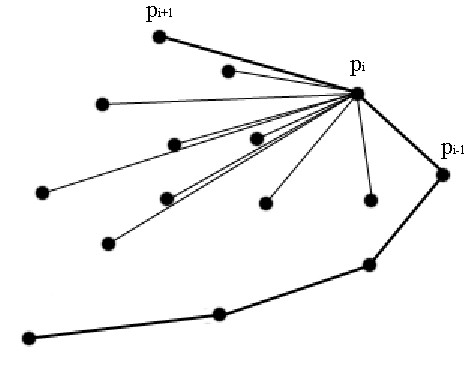


Рис 1. Иллюстрация алгоритма.

**Последовательная версия алгоритма**

В целом, программная реализация алгоритма мало отличается от того, что было изложено выше. В качестве объекта «точка» используется структура из двух double, для хранения координат x и y соответственно. Также необходимо запоминать индексы точек, которые составляют выпуклую оболочку и хранить флаги о том, включена ли данная точка в выпуклую оболочку (чтобы исключить ее проверку в дальнейшем).

Первые два цикла занимаются отысканием первой и второй точки. Затем используются вложенные циклы для отыскания оставшейся части точек.

**Параллельная версия алгоритма**

Для отыскания третьей и последующих точек выпуклой оболочки используются вложенные циклы (цикл for, занимающийся поиском следующей точки находится в цикле do, проверяющем, что очередная найденная точка не совпадает с первой найденной), поэтому здесь заключена основная вычислительная часть алгоритма. Для значительного ускорения достаточно распараллелить внутренний цикл for. Это легко сделать, поскольку необходимо лишь разделить массив точек по потокам, а затем провести редукцию, отыскав среди «локальных» минимумов глобальный, то есть производится распараллеливание по данным.

**Особенности реализации OpenMP версии**

Благодаря возможностям и простоте использования OpenMP, задача распараллеливания цикла свелась в написание одной директивы. Единственная трудность, с которой пришлось столкнуться при реализации – необходимость создания собственной операции редукции (такая возможность поддерживается, начиная с 4.0 версии).

**Особенности реализации TBB версии**

Задача распараллеливания цикла в TBB оказалась не менее легкой задачей. Почти всю часть работы по оптимизации вычислений берет на себя планировщик библиотеки, достаточно только «аккуратно» перенести логику из тела цикла в соответствующий функтор.

**Элементы автоматизированной проверки**

**Typer и Viewer**

Это программы, созданные для вспомогательных целей, занимаются переводом текстовой информации в бинарную и наоборот.

**Generator**

Это программа, занимающаяся генерацией входных данных для алгоритма. Основная ее суть заключается в наполнении массива случайными точками. В качестве области возможных значений используется квадрат 2000х2000. Возможные размеры массива предопределены и составляют тестовый набор: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 100, 200, 500, 1000, 2000, 5000, 10000, 15000, 20000, 25000, 50000, 100000, 500000, 800000, 1000000, 1500000.

**Checker**

Это программа, занимающаяся проверкой правильности алгоритма. Она принимает на вход два бинарника с результатами работы последовательной и параллельной версии соответственно и проверяет их на соответствие. Сначала проверяется размер выпуклой оболочки, а уже потом сами точки.

**Результаты экспериментов**

В качестве исходных данных был взят самый «большой» тест – 1,5 миллиона точек, поскольку здесь время работы измеряется в секундах.

Время работы параллельных версий будет замеряться при 1, 2, 4 потоках на системе с 6 ядрами.

Тест отдельно генерировался для OpenMP и TBB, поэтому обратить внимание нужно именно на ускорение.

Также стоит заметить, что компиляция проводилась на разных компиляторах (OpenMP компилировалась на MinGW в силу наличия там 4 версии и по некоторым причинам работает существенно быстрее, в то время как TBB компилировалась в Visual Studio).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OpenMP | | | TBB | | |
| 1 поток | 2 потока | 4 потока | 1 поток | 2 потока | 4 потока |
| 4.09 | 2.178 (1.87) | 1.229 (3.32) | 7,6 | 4.1 (1.85) | Необъяснимое поведение. |

Также было замечено странное поведение программы на TBB при установлении потоков больше 2.

**Вывод**

Реализовав последовательную, а также OpenMP и TBB версии, было выяснено, что в данной задаче преимущество на стороне OpenMP в силу простоты реализации (почти все преимущества, которые имеет TBB перед OpenMP в алгоритме Джарвиса не требуются, и, соответственно, работа планировщика идет только во вред времени работы программы).