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»

«Поиск кратчайших путей из одной вершины. Алгоритм Дейкстры»

Выполнила:

студентка группы 1608

Богатова М. Д.

Проверил:

Доцент кафедры МОиСТ, к.т.н.

А.В. Сысоев

Нижний Новгород

2019

Содержание

1. Введение 3
2. Постановка задачи 4
3. Описание алгоритма 5
3.1. Сведения и определения из теории графов 5
3.2. Неформальное объяснение 5
3.3. Алгоритм 6
3.3.1. Вспомогательная информация 6
3.3.2. Поиск кратчайшего расстояния от заданной вершины до всех остальных 7
3.3.3. Восстановление кратчайших путей 8
3.3.4. Асимптотическая сложность 8
3.3.5. Программная реализация последовательное версии алгоритма 9
3.3.6. Проверка корректности 11
4. Схема распараллеливания 12
5. Параллельная реализация алгоритма Дейкстры с помощью OpenMP 13
5.1. Описание реализации с помощью OpenMP предложенной схемы распараллеливания 13
5.2. Изменения в программной реализации 14
6. Параллельная реализация алгоритма Дейкстры с помощью TBB 15
6.1. Описание реализации с помощью TBB предложенной схемы распараллеливания 15
6.2. Изменения в программной реализации 16
7. Проверка корректности 17
8. Результаты экспериментов 18
8.1. Характеристики вычислительной машины 18
8.2. Данные экспериментов с параллельной реализацией алгоритма Дейкстры с помощью OpenMP 18

8.3. Данные экспериментов с параллельной реализацией алгоритма Дейкстры с помощью TBB 19
8.4. Выводы из результатов экспериментов 21
9. Заключение 23
10. Литература и источники 24

1. Введение

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. Сегодня известно множество алгоритмов для её решения, например, алгоритм Дейкстры, Флойда-Уоршелла и другие Значимость этой задачи определяется ее различными практическими применениями.

- Например, в GPS-навигаторах, где осуществляется поиск кратчайшего пути между двумя перекрестками. В качестве вершин выступают перекрестки, а дороги являются ребрами, которые лежат между ними. Сумма расстояний всех дорог между перекрестками должна быть минимальной, тогда найден самый короткий путь.

2. Постановка задачи

Дан взвешенный неориентированный граф *G* с множеством вершин *V* и с множеством ребер *E*, где E не содержит ребер отрицательного веса. *Необходимо* реализовать алгоритм, находящий кратчайшие пути от некоторой заданной вершины *start* до всех остальных вершин $vertex \in V$. Под выражением «кратчайший путь до вершины *vertex*» понимается сумма весов ребер *w*, которые необходимо использовать, чтобы посетить вершину *vertex* двигаясь из вершины *start.*

Поставленная задача называется "задачей о кратчайших путях с единственным источником" *(single-source shortest paths problem).*

На вход алгоритму поступает взвешенная матрица смежности вершин графа и стартовая вершина, на выходе – два вектора, первый из которых содержит кратчайшие расстояния до остальных вершин графа, а второй - кратчайшие пути до них.

Таким образом, сформулируем задачи, которые необходимо решить в рамках лабораторной работы:

1. В качестве алгоритма, решающего поставленную задачу, использовать алгоритм Дейкстры.
2. Предоставить программную реализацию на языке C++ трех версий обозначенного алгоритма: последовательной, параллельной с использованием OpenMP, параллельной с использованием TBB.
3. Для параллельных версий предложить корректную схему распараллеливания.
4. Произвести проверку корректности результатов выполнения параллельных версий
5. Провести эксперименты и замерить ускорение параллельных версий по сравнению с последовательной.
6. Оценить ускорение и масштабируемость параллельных реализаций алгоритма.
7. Сделать выводы о целесообразности распараллеливания алгоритма Дейкстры.

3. Описание алгоритма

3.1. Сведения и определения из теории графов

При описании графа будут использованы термины из теории графов. Для ясности и однозначности ниже приведены определения необходимых терминов.

- *Граф, или неориентированный граф* ***G*** — это упорядоченная ***G: = (V, E)***, где ***V***— непустое множество вершин или узлов, а ***E*** — множество неупорядоченных пар вершин, называемых рёбрами.
- *Взвешенный граф* — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра).
- *Вес ребра* — значение, поставленное в соответствие данному ребру взвешенного графа. Обычно вес — вещественное число, в таком случае его можно интерпретировать как «длину» ребра.
- *Путь* — последовательность ребер, такая, что конец одного ребра является началом другого ребра.
- *Расстояние между вершинами* — длина кратчайшего пути, соединяющей заданные вершины. Если такого пути не существует, расстояние полагается равным бесконечности.
- *Взвешенная матрица смежности графа* — матрица, значения элементов которой характеризуются смежностью вершин графа. Введем ее формальное определение: пусть $a_{i,j}$ – элемент матрицы, стоявший на пересечении i-ой строчки и j-го столбца. Тогда для определения матрицы A нужно определить $a_{i,j}$ для $i, j = 1, n$.

$$a_{i,j} = \begin{cases} 0, & i = j \\ w_{i,j} = w_{j,i}, & \text{если существует ребро между вершинами } i, j \\ -1, & \text{если ребра } i, j \text{ не существует} \end{cases}$$

3.2. Неформальное объяснение

Пусть граф **G*: = (V, E),*** представлен в виде взвешенной матрицы смежности и не имеет ребер отрицательного веса. Каждой вершине из ***V*** сопоставим **метку** — минимальное *известное* расстояние от этой вершины до ***a***.

Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация. Метка самой вершины ***a*** полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от ***a*** до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

Шаг алгоритма. Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина ***u***, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых ***u*** является предпоследним пунктом. Вершины, в которые ведут рёбра из ***u***, назовём *соседями* этой вершины. Для каждого соседа вершины ***u***, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки ***u*** и длины ребра, соединяющего ***u*** с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину ***u*** как посещённую и повторим шаг алгоритма.

3.3. Алгоритм

3.3.1. Вспомогательная информация

Для простоты описания алгоритма введем следующие обозначения и определения:

- *start* – вершина, от которой необходимо найти кратчайшие расстояния до всех остальных вершин графа
- *vertex* – любая вершина графа, $vertex \in V$
- *num_vertex* – общее количество вершин в графе
- *distance[]* – вектор, в котором для каждой *vertex* храниться текущая длина кратчайшего пути *distance[vertex]* из *start* в *vertex*
- *distance_queue* - структура данных, которая будет хранить непройденные вершины, упорядоченные по величинам их меток (по возрастанию). Наличие этой структуры позволяет хранить информацию о том, какие вершины уже были рассмотрены и на каждой итерации предоставлять непройденную вершину с наименьшей величиной *distance[vertex]*.
- *parent[]* – вектор, в котором для каждой вершины $vertex \neq start$ хранится номер вершины *parent[vertex]*, являющейся предпоследней в кратчайшем пути до вершины *vertex*.

- *Релаксация вдоль ребра* — это попытка улучшить значение *distance[b]* значением *distance[a] + len*. Фактически это значит, что мы пытаемся улучшить расстояние для вершины *b*, пользуясь ребром *(a, b)* и текущим кратчайшим расстоянием для вершины a.

3.3.2. Поиск кратчайшего расстояния от заданной вершины до всех остальных

Инициализация. Проинициализируем значение $distance[start] = 0$, а для всех остальных вершин зададим эту длину равной бесконечности, интерпретируемой как *-1*: $distance[vertex] = -1, vertex \neq start$. Во множество непройденных вершин *distance_queue* помещаем вершину *start*, так как именно от нее необходимо начинать рассматривать соседние вершины.

Шаг алгоритма.

На каждой итерации алгоритма выбирается первая вершина *vertex* из *distance_queue* и на текущей итерации из этой вершины проводятся релаксации: просматриваются все рёбра *(vertex, to)*, исходящие из вершины *vertex*, и для каждой такой вершины *to* алгоритм пытается улучшить значение *distance[to]*.

Улучшение значения *distance[to]* происходит по следующему правилу. Пусть длина текущего ребра *(vertex, to)* равна *len*, тогда *условие для релаксации* выглядит следующим образом:

$$distance[to] = \min(ditance[to], distance[vert] + len)$$

На этом текущая итерация заканчивается, алгоритм переходит к следующей итерации (снова выбирается вершина с наименьшей величиной *distance[vertex]*, из неё производятся релаксации, и т.д.). Итерации выполняются пока в *distance_queue* есть хотя бы одна вершина.

Замечание: вершины не будут повторно попадать в *distance_queue*, так как если вершина была рассмотрена, значит для нее уже было найдено кратчайшее расстояние, следовательно, не будет выполнено условие для релаксации расстояния до этой вершины. Таким образом, гарантируется, что алгоритм выполнит ровно *num_vertex* таких итераций.

Утверждается, что найденные значения *distance[vertex]* и есть искомые длины кратчайших путей из *start* в *vertex*. Если не все вершины графа достижимы из вершины *start*, то значения *distance[vertex]* для них так и останутся бесконечными.

3.3.3. Восстановление кратчайших путей

Для сохранения информации, достаточной для восстановления кратчайшего пути от вершины start достаточно формирования **вектора предков** *parent[]*.

При формировании вектора используется тот факт, что если из кратчайшего пути до некоторой вершины *vertex* удалить последнюю вершину, то получится путь, оканчивающийся некоторой вершиной *parent[vertex]*, и этот путь будет кратчайшим для вершины *parent[vertex]*.

Имея вектор предков, кратчайший путь можно будет восстановить следующим образом: последовательно брать предка от текущей вершины, пока не придём в вершину *start* — так получится искомый кратчайший путь, но записанный в обратном порядке. Таким образом, кратчайший путь *path* до вершины *start* равен:

$$path = (start, \dots, parent[parent[vertex]], parent[vertex], vertex)$$

Вектор предков строится так: при каждой успешной релаксации, т.е. когда из выбранной вершины *vertex* происходит улучшение расстояния до некоторой вершины *to*, запоминается, что предком вершины *to* является вершина *vertex*:

$$parent[to] = vert$$

3.3.4. Асимптотическая сложность

Итак, алгоритм Дейкстры представляет собой n итераций, на каждой из которых выбирается непомеченная вершина с наименьшей величиной *distance [vertex]*, эта вершина помечается, и затем просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра делается попытка улучшить значение *distance* [] на другом конце ребра.

Время работы алгоритма зависит от времени выполнения двух основных операций:

- n раз производится поиск вершины с наименьшей величиной *distance [vertex]* среди всех непомеченных вершин, т.е. среди *O(n)* вершин
- m раз производится попытка совершения релаксации, т.е. изменения величины *distance [to]*.

Так как в описании алгоритма упомянута некоторая структура данных *distance_queue,* предполагается, что эта структура данных позволяет производить операции первого и второго вида за $O(\log n)$. Тогда время работы алгоритма Дейкстры составит:

$$O(n \log n + m \log n) = O(m \log n)$$

При реализации алгоритма на C++ в качестве такой структуры данных можно использовать стандартный контейнер *set* или *priority_queue* библиотеки STL. Первый основан на красно-чёрном дереве, второй — на бинарной куче.

3.3.5. Программная реализация последовательное версии алгоритма

Приведем программную реализацию последовательной версии алгоритма для указания некоторых ее особенностей.

```
void FindDistance(int start, std::size_t n, const GraphType& d,
            std::vector<DstType>*dst, std::vector<int>* prev_v) {
    if (start < 1 || start > static_cast<int>(n))
        throw std::out_of_range("Incorrect start or finish vertex");
    (*dst).assign(n, -1);
    (*prev_v).assign(n, -1);

    /* Lambda function that returns true if the distance to the
    vertex a is less than the distance to the vertex b    */
    auto set_function = [dst](DstType a, DstType b) {
        return ((*dst)[a] <(*dst)[b]) ||
            (((*dst)[a] == (*dst)[b]) && (a < b));
    };

    // Set of unvisited vertices, ordered by non-decreasing distances to them
    std::set<int, decltype(set_function) > dst_queue(set_function);

    (*dst)[start - 1] = 0;
    dst_queue.insert(start - 1);

    while (!dst_queue.empty()) {
        int next_vertex = *dst_queue.begin();
        dst_queue.erase(dst_queue.begin());
        for (int i = 0; i < static_cast<int>(n); ++i) {
            if (!d[next_vertex][i] || d[next_vertex][i] == -1)
                continue;
            if ((*dst)[i] >(*dst)[next_vertex] + d[next_vertex][i]
                                                || (*dst)[i] == -1) {
                (*prev_v)[i] = next_vertex;
                dst_queue.erase(i);
                (*dst)[i] = (*dst)[next_vertex] + d[next_vertex][i];
                dst_queue.insert(i);
            }
        }
    }
}
```

Листинг 1. Поиск кратчайшего расстояния от заданной вершины до всех остальных

Первое, что хотелось бы заметить, это *distance_queue* есть *std::set<>* который упорядочивает значения согласно принимаемой лямбда-функции. Второе замечание – при проведении релаксации сначала необходимо удалить изменяемую вершину из *distance_queue,* изменить значение ее метки и заново добавить в *distance_queue.* Это необходимо для того, чтобы порядок вершины в множестве изменился.

В листинге 2 представлен алгоритм восстановления кратчайшего пути. Так как далее в программе этот путь нигде не используется, а только выводится на экран, было принято решение реализовать функцию, которая рекурсивно выводит вершины, которые содержит кратчайший путь.

```
void PrintPathToVertex(
    const std::vector<int>& prev, int source_vertex, int curr_v) {
    if (curr_v != source_vertex) {
        PrintPathToVertex(prev, source_vertex, prev[curr_v]);
    }
    std::cout << curr_v + 1 << " ";
}
```

Листинг 2. Восстановление кратчайшего пути

3.3.6. Проверка корректности

Для подтверждения корректности работы описанного алгоритма была реализована функция, которая запускает этот алгоритм на простейшем графе. Для этого графа можно самостоятельно и без особых усилий вычислить кратчайшие пути от некоторой вершины до всех остальных.

Граф представлен на рис. 1. Взвешенная матрица смежности для него выглядит следующим образом:

0	7	9	-1	-1	14
7	0	10	15	-1	-1
9	10	0	11	-1	2
-1	15	11	0	6	-1
-1	-1	-1	6	0	9
14	-1	2	-1	9	0

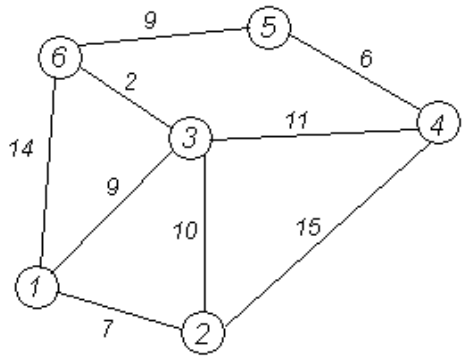

Рисунок 1

Замечание: значение -1 означает отсутствие ребра между соответствующими вершинами или расстояние между ними равняется бесконечности Программа, реализующая этот алгоритм, имеет вывод, описанный на рис. 2. Нетрудно заметить, что найденный кратчайшие пути и вправду являются таковыми (рис. 3).

```
Shortest distance between 1 and 1 is    0
Path: 1
Shortest distance between 1 and 2 is    7
Path: 1 2
Shortest distance between 1 and 3 is    9
Path: 1 3
Shortest distance between 1 and 4 is    20
Path: 1 3 4
Shortest distance between 1 and 5 is    20
Path: 1 3 6 5
Shortest distance between 1 and 6 is    11
Path: 1 3 6
```

Рисунок 2

4. Схема распараллеливания

Первый цикл, который осуществляет выбор вершины для проведения из нее релаксаций, распараллеливанию не подлежит. Это объясняется тем, что предварительно неизвестно, в каком порядке будут рассматриваться вершины, так как этот фактор определяет порядок их поступления в множество и величина их метки. В таком случае, предлагается организовать распараллеливание той части алгоритма, которая на каждой итерации делает попытки релаксации расстояния от рассматриваемой вершины до всех «соседей». Так как это есть цикл *for()* в стандартном виде, со счетчиком, следовательно, необходимо организовать распараллеливание цикла с известным числом итераций.

Однако, необходимо учесть, что внутри этого цикла есть код, который изменяет данные, являющиеся общими для всех потоков (*distance_queue, см 3.2.*). В таком случае возникает *состояние гонки*: когда результат программы зависит от темпа выполнения потоками своей части вычислений. Для исключения гонки необходимо обеспечить, чтобы изменение общих данных осуществлялось в каждый момент времени только одним единственным потоком – иными словами необходимо обеспечить *взаимное исключение* (*mutual exclusion*) потоков при работе с общими данными.

На рис. 4 выделена часть алгоритма, которая поддается распараллеливанию.

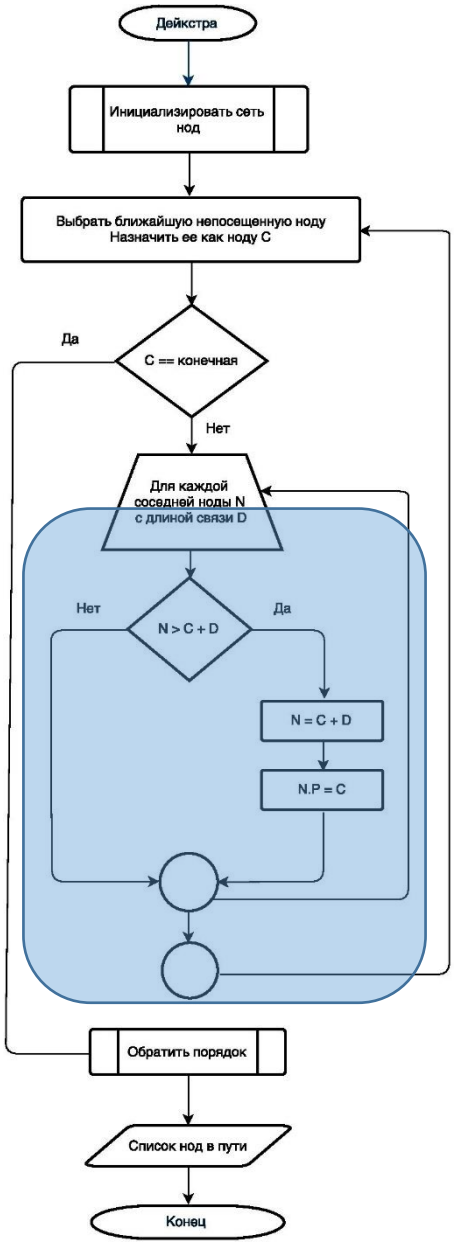

Рисунок 4

5. Параллельная реализация алгоритма Дейкстры с помощью OpenMP

5.1. Описание реализации с помощью OpenMP предложенной схемы распараллеливания

Для предложенной схемы распараллеливания необходим инструмент, который позволяет реализовать распараллеливание цикла с известным числом повторений. Таким инструментом в OpenMP является директива препроцессора

#pragma omp parallel for < параметры >

Упомянутая директива имеет один из варьируемых параметров, который может в лучшую сторону повлиять на ускорение при распараллеливании алгоритма Дейкстры: это параметр для управления распределением итераций в распараллеливаемом цикле *schedule(type, chunk).*

Возникает вопрос: какой же тип распределения итераций между потоками использовать? Ответ был получен экспериментально: программа запускалась с разными типами и наилучший результат показал алгоритм с статическим распределением итераций (см. таб. 1). Результат очевиден: распараллеливаемый цикл имеет довольно малое количество вычислений и итерации разной длины. Поэтому невыгодно использовать распределение, отличное от статического.

Тип	static	Dynamic	guided
Ускорение	1.7	0,0889	1,2

Таблица 1.

Как отмечалось в разделе 4, имеет место состояние гонки и необходимо испоьзовать механизм взаимоисключения. В OpenMP взаимоисключение может быть организовано при помощи механизма *критических секций* (*critical sections*). Это позволяет гарантировать, в каждый момент времени в критической секции будет находится ровно один поток.

#pragma omp critical [(name)]

5.2. Изменения в программной реализации

Одним из преимуществ использования OpenMP является минимальные изменения исходного кода программ. Для реализации параллельной версии алгоритма Дейкстры в код были добавлены указанные в 5.1. директивы (см листинг 3)

```
void FindDistanceOMP(int start, std::size_t n, const GraphType& d,
            std::vector<DstType>* dst, std::vector<int>* prev_v) {
    if (start < 1 || start > static_cast<int>(n))
        throw std::out_of_range("Incorrect start or finish vertex");
    dst->assign(n, -1);
    prev_v->assign(n, -1);
    /* Lambda function that returns true if the distance to the
    vertex a is less than the distance to the vertex b    */
    auto set_function = [dst](DstType a, DstType b) {
        return ((*dst)[a] < (*dst)[b]) ||
            (((*dst)[a] == (*dst)[b]) && (a < b));
    };

    // Set of unvisited vertices, ordered by non-decreasing distances to them
    std::set<int, decltype(set_function) > dst_queue(set_function);

    (*dst)[start - 1] = 0;
    dst_queue.insert(start - 1);

    int i;
    while (!dst_queue.empty()) {
        int next_vertex = *dst_queue.begin();
         dst_queue.erase(dst_queue.begin());
#pragma omp parallel for private(i)
        for (i = 0; i < static_cast<int>(n); ++i) {
            if (!d[next_vertex][i] || d[next_vertex][i] == -1)
                continue;
            if ((*dst)[i] > (*dst)[next_vertex] + d[next_vertex][i]
                                                || (*dst)[i] == -1) {
                (*prev_v)[i] = next_vertex;
#pragma omp critical
                {
                dst_queue.erase(i);
                (*dst)[i] = (*dst)[next_vertex] + d[next_vertex][i];
                dst_queue.insert(i);
                }
            }
        }
    }
}
```

Листинг 3. Поиск кратчайшего расстояния от заданной вершины до всех остальных. Версия OpenMP

6. Параллельная реализация алгоритма Дейкстры с помощью TBB

6.1. Описание реализации с помощью TBB предложенной схемы распараллеливания

Для распараллеливания цикла с известным количеством повторений используется шаблонная функция библиотеки TBB, которая принимает 2 параметра: итерационное пространство и функтор.

```
template < typename Range, typename Body >
void parallel_for(const Range& range, const Body& body);
```

В реализации используется одномерное итерационное пространство *blocked_range<…>*, которое реализовано в библиотеке TBB. Это итерационное пространство имеет 3 основных параметра: *begin*, *end и grainsize*. *Grainsize* означает минимальный размер итерационного пространства Изменяя его значение можно влиять на ускорение программы. Для подбора оптимального значения grainsize были произведены испытания по алгоритму, изложенному ниже.

1. Установите значение **grainsize** достаточно большим, например равным размеру итерационного пространства в случае использования класса **blocked_range**.
2. Запустите приложение в один поток, замерьте время его выполнения.
3. Установите значение **grainsize** в 2 раза меньше, запустите приложение по-прежнему в один поток и оцените замедление по отношению к шагу 2. Если приложение замедлилось на 5-10%, это хороший результат. Продолжайте уменьшение **grainsize** до тех пор, пока замедление не превысит 5-10%.

Испытания проводились для полного графа с количеством вершин n = 100

Размер grainsize	100	50	25	13	20
Время работы	0.000355808	0.000385187	0.000422027	0.000521821	0.000522753

Таблица 2. Подбор значение grainsize

Следовательно, оптимальным значением grainsize для n = 100 является grainsize = n/4 = 25. Поэтому в программной реализации был выбран grainsize = n/4

Взаимоисключение в TBB организуется с помощью механизма мьютексов ОС.

6.2. Изменения в программной реализации

В случае с использованием TBB, изменения программы также не особо большие: добавлено объявление, захват и освобождение мьютекса, а также необходимая функция *tbb::parallel_for()*, одним из аргументов которой является лямбда-функция, которая и реализует вычисления (проведения релаксаций)

```
void FindDistanceTBB(int start, std::size_t n, const GraphType& d,
    std::vector<DstType>* dst, std::vector<int>* prev_v) {
    if (start < 1 || start > static_cast<int>(n))
        throw std::out_of_range("Incorrect start or finish vertex");
    dst->assign(n, -1);
    prev_v->assign(n, -1);
    /* Lambda function that returns true if the distance to the
    vertex a is less than the distance to the vertex b    */
    auto set_function = [dst](DstType a, DstType b) {
        return ((*dst)[a] < (*dst)[b]) ||
            (((*dst)[a] == (*dst)[b]) && (a < b));
    };

    // Set of unvisited vertices, ordered by non-decreasing distances to them
    std::set<int, decltype(set_function) > dst_queue(set_function);

    (*dst)[start - 1] = 0;
    dst_queue.insert(start - 1);
    tbb::mutex m;

    while (!dst_queue.empty()) {
        int next_vertex = *dst_queue.begin();
        dst_queue.erase(dst_queue.begin());

        tbb::parallel_for(tbb::blocked_range<int>(0, n, n/4),
            [&](const tbb::blocked_range<int> &r) {
                int i;
                for (i = r.begin(); i != r.end(); ++i) {
                    if (!d[next_vertex][i] || d[next_vertex][i] == -1)
                        continue;
                    if ((*dst)[i] > (*dst)[next_vertex] + d[next_vertex][i]
                        || (*dst)[i] == -1) {
                        m.lock();
                        (*prev_v)[i] = next_vertex;
                            dst_queue.erase(i);
                            (*dst)[i] = (*dst)[next_vertex] + d[next_vertex][i];
                            dst_queue.insert(i);
                        m.unlock();
                    }
                }
            });
    }
```

Листинг 4. Поиск кратчайшего расстояния от заданной вершины до всех остальных. Версия TBB

7. Проверка корректности

Для подтверждения корректности в программе осуществляется сравнение результатов, полученных параллельным путем и последовательным. После выполнения выводится соответствующее сообщение: в случае совпадения **«true»**, в случае ошибки **«false».**

Запустим параллельные программы для случайно сгенерированного графа с 1000 вершинами. Видим, что результаты одинаковы (рис. 5, рис. 6), а это значит, что алгоритм работает корректно.

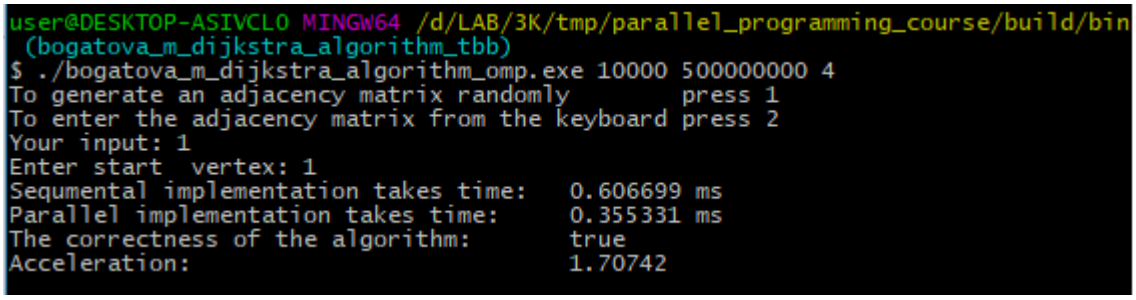

Рисунок 5

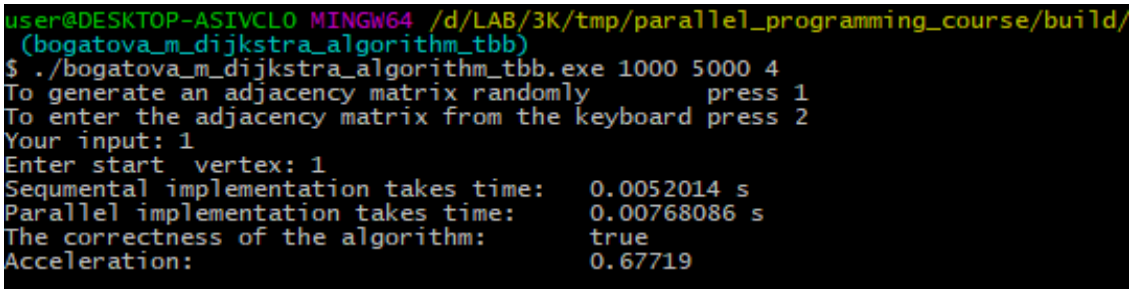

Рисунок 6

8. Результаты экспериментов

8.1. Характеристики вычислительной машины

- Windows 10, x64
- Процессор: AMD A8-7410, 2,20 ГГц
- Логических процессоров: 4
- Оперативная память: 6 Гб

Тестирования проводились на полном графе, то есть количество ребер $m = n*(n-1)/2$, где n – количество вершин.

8.2. Данные экспериментов с параллельной реализацией алгоритма Дейкстры с помощью OpenMP

Число вершин / Количество потоков	100	1000	10000
1	0.892697	0.873629	1.04056
2	0.302092	1.03407	1.46933
4	0.286521	0.708438	2.05401
8	0.184642	0.509636	1.97283
16	0.0869626	0.386916	1.57286
32	0.0202944	0.188538	0.90198

Таблица 3. Ускорение параллельной программы с использованием OpenMP

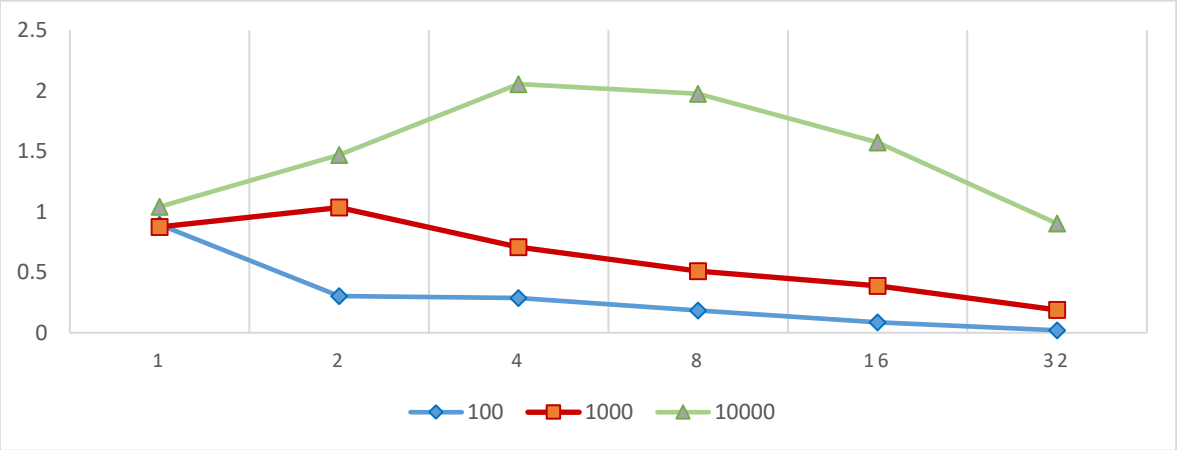

График 1. Зависимость ускорения программы от числа потоков

8.3. Данные экспериментов с параллельной реализацией алгоритма Дейкстры с помощью TBB

Число вершин / Количество потоков	100	1000	10000
1	0.686869	0.791568	0.904583
2	0.631167	0.926339	1.36375
4	0.515428	0.863854	1.73923
8	0.529872	0.741204	1.18292
16	0.534676	0.709592	1.16665
32	0.526539	0.63675	1.13275

Таблица 4. Ускорение параллельной программы с использованием TBB (grainsize = n/4)

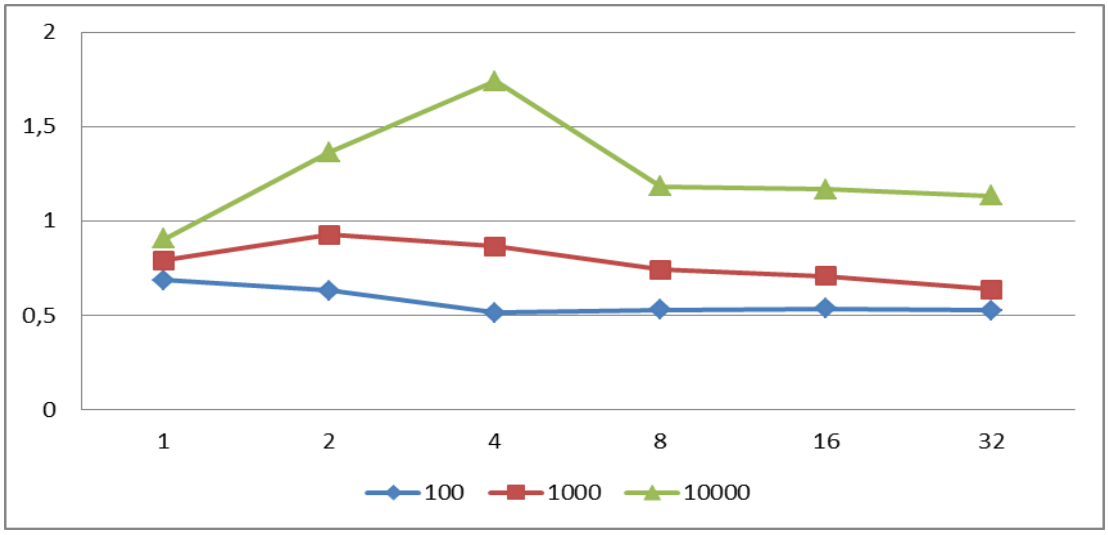

График 2. Зависимость ускорения программы от числа потоков

(grainsize = n/4)

Число вершин / Количество потоков	100	1000	10000
1	0.878505	0.873909	0.904583
2	0.613775	0.842534	1.39375
4	0.513837	0.792168	1.80463
8	0.394342	0.741204	1.47566
16	0.311163	0.532756	1.37005
32	0.24567	0.406121	1.13352

Таблица 5. Ускорение параллельной программы с использованием TBB (grainsize = n/num_threads)

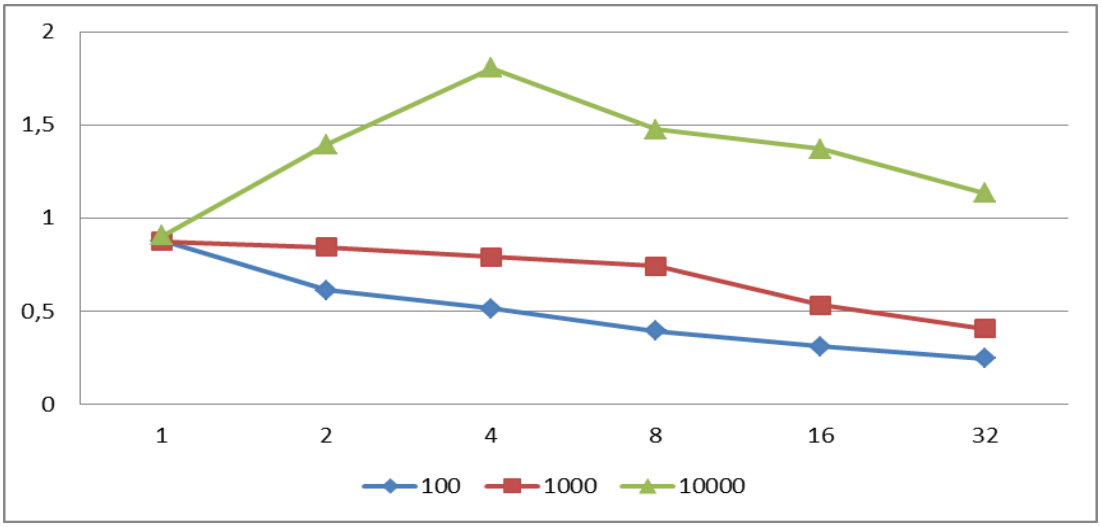

График 3. Зависимость ускорения программы от числа потоков

(grainsize = n/num_threads)

8.4. Выводы из результатов экспериментов

OpenMP версия

Максимальное ускорение параллельной программы было получено только при достаточно большом количество вершин (10000) и при минимальных накладных расходах (количество потоков = число вычислительных узлов компьютера). Из чего можно сделать вывод, что распараллеливание алгоритма Дейкстры имеет смысл только при большом объеме данных.

Также можно сделать следующее наблюдение: при n = 10000 с ростом числа потоков ускорение программы падает относительно медленно. Это можно объяснить тем, что в параллельной OpenMP версии алгоритма основной вычислительной нагрузкой является проверка условий релаксации, и только лишь потом вход в критическую секцию и выполнение самой релаксации. Таким образом потоки блокируют друг друга в критической секции не так часто, как это кажется на первый взгляд.

TBB версия

Были проведены испытание для различных значений grainsize. Сделаны следующие наблюдения

- при *grainsize = n/4* и при количестве потоков более 4 ускорение программы изменялось незначительно. Это говорит о том, что в таком случае вычисления выполняются только на 4-х потоках, а остальные простаивают. Следовательно, не имеет смысла увеличивать далее количество потоков и такой алгоритм обладает плохой масштабируемостью.
- при *grainsize = n/num_threads* прослеживается значительное уменьшение ускорения при количестве потоков более 4-х. В таком случае алгоритм является хорошо масштабируемым.

Вывод из этих наблюдений следующие. Во-первых, чем больше *grainsize,* тем меньше масштабируемость. Во-вторых, минимальные накладные расходы при распараллеливании приходятся на количество потоков, равное числу вычислительных узлов компьютера.

На ряду с этим следует отметить, что максимальное ускорение, полученное в рассматриваемой версии алгоритма, меньше ~ на 0,2, чем в OpenMP версии. Это говорит о том, что в TBB создается больше накладных расходов при создании потоков и распределении вычислений между ними, чем в OpenMP. И эти накладные расходы несравнимы с объемом вычислений.

Общие наблюдения

Из динамики изменения ускорения программы в зависимости от числа потоков можно сделать вывод о масштабируемости алгоритма. К сожалению, протестировать работу алгоритма на вычислительной машине с б`ольшим количеством ядер не удалось, целесообразно рассматривать значения ускорения только для 1/2/4-х потоков. Так как на этом интервале ускорение программы растет, следовательно, приложение обладает неплохой масштабируемостью. Однако следует помнить, что бесконечное увеличение потоков не приведет к увеличению ускорения: все же существуют накладные расходы, и чем больше потоков, тем их больше.

9. Заключение

В лабораторных работах были реализованы последовательная и параллельные версии алгоритма Дейкстры с использованием средств OpenMP и TBB. Также были проведены эксперименты и оценено ускорение параллельных версий алгоритма. На основании результатов лабораторной работы сделаем следующий вывод: *распараллеливание алгоритма Дейкстры целесообразно только при достаточно большом объеме данных*, так как вычислительная нагрузка мала, а накладные расходы велики. Также следует заметить, что оптимальнее для распараллеливания использовать OpenMP, так как соответствующая версия алгоритма показала лучшие результаты.

10. Литература и источники

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е. — М.: Вильямс, 2005. — 1296 с.

2. Википедия - свободная энциклопедия. Алгоритм Дейкстры [Электронный ресурс] // https://ru.wikipedia.org/wiki/Алгоритм_Дейкстры

3. . Иванов М. Нахождение кратчайших путей от заданной вершины до всех остальных вершин алгоритмом Дейкстры для разреженных графов [Электронный ресурс] // https://e-maxx.ru/algo/dijkstra_sparse