

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

Отчет
по лабораторной работе
**Поиск кратчайших путей из одной вершины
(алгоритм Мура)**

Выполнил:
студент группы 381608
Мезина М.Д.

Проверил:
Доцент кафедры МОиСТ
Сысоев А.В.

Нижний Новгород
2019

ОГЛАВЛЕНИЕ

Оглавление

I.	ВВЕДЕНИЕ	3
	Почему теория графов?	3
	Почему алгоритм поиска кратчайшего пути?	3
	Почему алгоритм Мура?	3
	Для чего нужно использовать параллельное программирование?	4
II.	ПОСТАНОВКА ЗАДАЧИ	5
III.	СУТЬ АЛГОРИТМА МУРА	6
	Необходимые понятия теории графов	6
	Задача алгоритма	6
	Алгоритм	7
	Пример работы алгоритма	8
	Оптимизация алгоритма	8
	Число решений алгоритма	9
IV.	СХЕМА РАСПАРАЛЛЕЛИВАНИЯ	10
V.	МЕТОД РЕШЕНИЯ	11
	Входные параметры	11
	Формат хранения графа	11
	Генерация графа	12
	Последовательная часть	12
	Параллельная часть с использованием OpenMP	13
	Параллельная часть с использованием TBB	13
VI.	РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ	15
	Проверка корректности	15
	Анализ сложности алгоритма	18
	Результат работы параллельных частей	19
VII.	ВЫВОДЫ	23
VIII.	ИСТОЧНИКИ	24
IX.	ПРИЛОЖЕНИЯ	25
	Приложение №1	25
	Приложение №2	29
	Приложение №3	36

I. ВВЕДЕНИЕ

Почему теория графов?

Теория графов — раздел дискретной математики, изучающий свойства графов. В общем смысле граф представляется как множество вершин (узлов), соединённых рёбрами.

Области применения теории графов:

- Химия (для описания структур, путей сложных реакций, правило фаз также может быть интерпретировано как задача теории графов); компьютерная химия — сравнительно молодая область химии, основанная на применении теории графов. Теория графов представляет собой математическую основу хемоинформатики. Теория графов позволяет точно определить число теоретически возможных изомеров углеводородов и других органических соединений;
- Информатика и программирование (граф-схема алгоритма, автоматы);
- Коммуникационные и транспортные системы. В частности, маршрутизация данных в Интернете;
- Экономика;
- Логистика;
- Схемотехника (топология межсоединений элементов на печатной плате или микросхеме представляет собой граф или гиперграф);

Почему алгоритм поиска кратчайшего пути?

Задача о кратчайшем пути — задача поиска самого короткого пути между двумя вершинами на графе, в которой минимизируется сумма весов рёбер, составляющих путь. Задача о кратчайшем пути является одной из важнейших классических задач теории графов.

Значимость данной задачи определяется её различными практическими применениями. Например, в GPS-навигаторах осуществляется поиск кратчайшего пути между двумя перекрёстками. В качестве вершин выступают перекрёстки, а дороги являются рёбрами, которые лежат между ними. Если сумма длин дорог между перекрёстками минимальна, тогда найденный путь самый короткий.

В различных постановках задачи роль длины ребра могут играть не только сами длины, но и время, стоимость, расходы, объём затрачиваемых ресурсов (материальных, финансовых, топливно-энергетических и т. п.) или другие характеристики, связанные с прохождением каждого ребра. Таким образом, задача находит практическое применение в большом количестве областей (информатика, экономика, география и др.).

Почему алгоритм Мура?

Существует множество различных постановок данной задачи. Алгоритм Мура имеет одну из самых простых реализаций и позволяет находить кратчайшие расстояния в взвешенных ориентированных графах. Кроме того, этот алгоритм обладает асимптотической сложностью $O(V \times E)$, где V — число вершин в графе, E — число ребер. Данная сложность является одной из самых оптимальных среди всех алгоритмов поиска кратчайшего пути.

История алгоритма связана сразу с тремя независимыми математиками: Лестером Фордом, Ричардом Беллманом и Эдвардом Муром. Форд и Беллман опубликовали алгоритм в 1956 и 1958 годах соответственно, а Мур сделал это в 1957 году. И иногда его называют алгоритмом Беллмана — Форда — Мура, алгоритм Мура или алгоритмом Беллмана-Форда. Все 3 математика разработали схожий по смыслу алгоритм, но с небольшими отличиями в реализации.

Для чего нужно использовать параллельное программирование?

Область применения теории графов и, в частности, алгоритма поиска кратчайшего пути достаточно обширна, предполагая в том числе работу с графами, в которых сотни тысяч или даже миллионы вершин. Если предположить, что максимальное число ребер в графе (без петель и кратных ребер) с числом вершин V исчисляется как $V \times (V - 1)$, то верхняя асимптотическая сложность алгоритма Мура оценивается как $O(V^3)$, а это получается порядка 10^{15} операций, что достаточно много.

Конечно, на практике существует множество методов оптимизации данных алгоритмов, но тем не менее, использование параллельного программирования позволяет значительно сократить их время работы.

II. ПОСТАНОВКА ЗАДАЧИ

Итак, целью данной работы является реализация алгоритма Мура для поиска кратчайшего пути в графе с помощью основных технологий параллельного программирования для систем с общей памятью (OpenMP и TBB).

Работа будет проводиться в несколько этапов:

1. Изучение основ графов и алгоритма Мура;
2. Разработка функции для генерации случайного графа с фиксированным порядком и размером;
3. Реализация функции для генерации графа решетки в пространстве R^2 фиксированного порядка;
4. Реализация последовательной версии алгоритма Мура;
5. Оценка времени работы последовательной версии в зависимости от порядка и размера графа;
6. Разработка схемы распараллеливания алгоритма;
7. Реализация разработанной схемы с помощью технологии OpenMP, подбор оптимальной схемы распараллеливания;
8. Реализация разработанной схемы с помощью технологии TBB, подбор оптимальной схемы распараллеливания;
9. Сравнение времени работы и ускорения полученных реализаций на сгенерированном случайном графе;
10. Сравнение времени работы и ускорения полученных реализаций на сгенерированном графе решетки;
11. Анализ полученных результатов и подведение итогов;

Все реализации будут протестированы на персональном компьютере с оперативной памятью 8Гб и процессором Intel® Core™ i5-8280U, имеющим 8 логических и 4 физических ядра с частотой 1,60 ГГц.

III. СУТЬ АЛГОРИТМА МУРА

Необходимые понятия теории графов

Для того, чтобы понимать смысл алгоритма Мура, необходимо иметь представление о теории графов. Поэтому здесь будут указаны необходимые понятия:

- **Неориентированный граф** G — это упорядоченная пара $G = (V, E)$, где V — непустое множество **вершин** или **узлов**, а E — множество пар вершин, называемых **рёбрами**;
- **Ориентированный граф** (сокращённо **орграф**) G — это упорядоченная пара $G = (V, E)$, где V — непустое множество **вершин** или **узлов**, и E — множество (упорядоченных) пар различных вершин, называемых **дугами**, или **ориентированными рёбрами**.
- Пусть (v, w) — это дуга. Тогда вершину v называют её **началом**, а w — **концом**.
- **Порядком** графа называется $|V|$ - число вершин этого графа;
- **Размером** графа называется $|E|$ - число ребер (или дуг) графа;
- Два ребра называются **кратными**, если множества их концевых вершин совпадают.
- Дуги называются **кратными**, если у них совпадают начала и концы;
- Ребро называется **петлёй**, если его концы совпадают, то есть $e = (v, v)$;
- Граф без петель и кратных рёбер называется **простым**;
- Граф называется **взвешенным**, если каждому ребру графа поставлено в соответствие некоторое число, называемое **весом** ребра;
- **Матрица смежности** - таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот);

Это наиболее удобный способ представления плотных графов.

- **Граф решётки** — это граф, рисунок которого, вложенный в некоторое евклидово пространство R^n , образует регулярную мозаику.

Общий вид **графа решётки** — это граф, вершины которого соответствуют точкам на плоскости с различными координатами, x -координатами из диапазона $1, 2, \dots, n$, y -координатами из диапазона $1, 2, \dots, m$, и вершины которого соединены ребром, если соответствующие точки находятся на расстоянии 1.

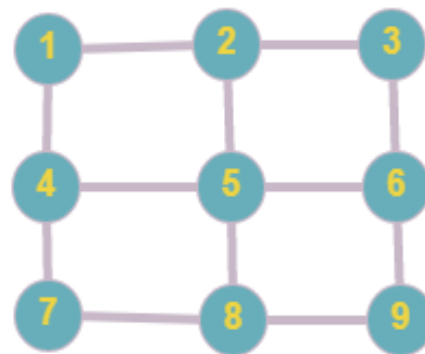


Рисунок 1. Квадратный граф-решетка 3x3

Задача алгоритма

Дан ориентированный или неориентированный граф G со взвешенными рёбрами. Требуется найти кратчайшие пути от выделенной вершины s до всех вершин графа.

Алгоритм

Перед началом работы алгоритма упростим граф G , удалив петли и оставив среди кратных ребер только ребра с наименьшей длиной, поскольку эти «лишние» ребра не будут учитываться в оптимальном пути.

Теперь обратим внимание на 2 свойства:

1. В оптимальном маршруте нет повторяющихся вершин (ну или как минимум их можно избежать - это зависит от того, допускаются или нет нулевые и отрицательные веса рёбер; для краткости будем считать, что не допускаются и тем самым повторяющиеся вершины просто невозможны).
2. Для любого оптимального маршрута между некоторыми двумя вершинами любой его участок также будет оптимальным для своих крайних вершин. В частности, будет оптимальным участок между первой и предпоследней вершинами исходного маршрута.

Алгоритм основан на построении матрицы кратчайших расстояний d между исходной вершиной s и всеми остальными. Каждому столбцу матрицы отвечает одна из вершин. В строке с номером i хранятся минимальные расстояния до этих вершин, за которые можно добраться до них из первой за не более чем i шагов (т.е. рёбер). Так для 0-ой строки будет расстояние только от вершины s до самой себя, т.е.:

$$d[0][j] = \begin{cases} 0, & j = s \\ \text{infinity}, & j \neq s \end{cases}$$

Будем построчно заполнять эту матрицу. Первоначально скопируем значение i -той строки матрицы из $i-1$ -ой:

$$d[i][j] = d[i-1][j]$$

Теперь для каждой вершины $from$, до которой мы можем дойти за i шагов ($d[i][from] \neq \text{infinity}$) пройдем по всем ребрам, которые исходят из этой вершины.

Пусть окончанием текущего ребра будет вершина to . Тогда, проверим, будет ли улучшаться расстояние от вершины s до вершины to , если мы перед вершиной to будем проходить через вершину $from$. Иначе говоря:

$$d[i][to] == \text{infinity} \text{ или } d[i][to] > d[i-1][from] + g[from][to], \text{ где } g[from][to] - \text{вес ребра из } from \text{ в } to.$$

В случае выполнения этого условия обновим расстояние до вершины to :

$$d[i][to] = d[i-1][from] + g[from][to]$$

После того, как мы прошли по всем вершинам $from$, необходимо проверить алгоритм на завершение. Для этого сравним строки i и $i-1$. В случае, если эти строки совпадают, значит за эту итерацию не было никаких изменений, следовательно сколько бы еще итераций не проводилось, никакие расстояния обновляться не будут. Т.е. расстояния, описанные в строке i и будут оптимальными для вершины s , и алгоритм можно завершить.

Стоит отметить, что максимальное число итераций будет равно числу вершин в графе исходя из свойства №1, описанного выше.

Таким образом мы можем найти длины оптимальных путей. В случае, если нам надо найти сам путь (т.е. последовательность вершин этого пути), заведем дополнительный массив $prev$, в котором будем хранить для каждой вершины такую вершину, из которой мы туда пришли. И каждый раз, когда мы обновляем для вершины to расстояние, исходя из вершины $from$, также будем обновлять и значение в массиве $prev$:

$$prev[to] = from.$$

Теперь после завершения алгоритма восстановим путь, если он существует. Возьмем следующую последовательность для вершины v :

$$v, prev[v], prev[prev[v]], \dots, s.$$

Найдя эту последовательность и перевернув ее, мы получим путь от вершины s до v .

Пример работы алгоритма

Возьмем граф на рис.2 и найдем оптимальные пути от вершины 1 до всех остальных.

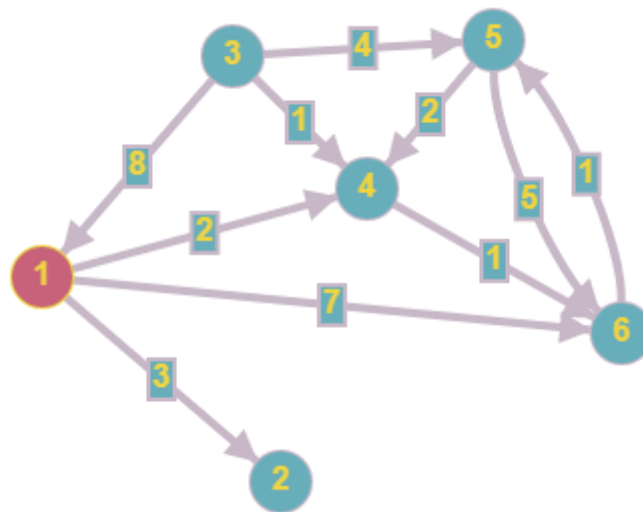


Рисунок 2. Пример графа для алгоритма Мура

Итерации алгоритма для него будет выглядеть следующим образом:

$d[i][j]$	1	2	3	4	5	6
0	0	inf	inf	inf	inf	inf
1	0	3	inf	2	inf	7
2	0	3	inf	2	inf	3
3	0	3	inf	2	4	3
4	0	3	inf	2	4	3

$prev[j]$	1	2	3	4	5	6
$i = 0$	-	-	-	-	-	-
$i = 1$	-	1	-	1	-	1
$i = 2$	-	1	-	1	-	4
$i = 3$	-	1	-	1	6	4
$i = 4$	-	1	-	1	6	4

В результате получаются следующие пути:

Вершина	Расстояние	Путь
1	0	1
2	3	1 → 2
3	Infinity	—
4	2	1 → 4
5	4	2 → 4 → 6 → 5
6	3	1 → 4 → 6

Оптимизация алгоритма

Для работы данного алгоритма используется $O(N^2)$ дополнительной памяти за счет матрицы d , где N – число вершин в графе. Однако, данное значение можно сократить до $O(N)$. Для этого вместо хранения всей матрицы, будем хранить только одну ее строчку, обновляя результаты внутри нее самой. При этом понятие d меняется на массив, содержащий расстояния от вершины s до любой другой.

При этом может измениться результат итераций, потому что для вершины обработанные в конце, с высокой вероятностью будут использовать новое расстояние, полученное на этой итерации.

Так для примера изменение реализации будет выглядеть следующим образом:

$d[i]$	1	2	3	4	5	6
$i = 0$	1	inf	inf	inf	inf	inf
$i = 1$	1	4	inf	3	inf	4
$i = 2$	1	4	inf	3	5	4
$i = 3$	1	4	inf	3	5	4

$prev[j]$	0	1	2	3	4	5
$i = 0$	-	-	-	-	-	-
$i = 1$	-	1	-	1	-	4
$i = 2$	-	1	-	1	6	4
$i = 3$	-	1	-	1	6	4

Исходя из таблиц видно, что конечный результат не меняется несмотря на то, что меняется результат промежуточных итераций.

Число решений алгоритма

Нетрудно заметить, что решений задачи поиска кратчайшего пути может быть несколько. Главным критерием оптимальности является значение длины полученного пути. Так, две разные реализации алгоритма могут предоставить разные результаты, однако различаться могут только последовательности вершин в пути, а значение длины должно быть одинаковым минимальным.

Это стоит учесть при сравнении результатов различных реализаций алгоритма Мура.

IV. СХЕМА РАСПАРАЛЛЕЛИВАНИЯ

Первое, что стоит заметить, это невозможность распараллеливания самих итераций. Дело в том, что результат каждой итерации зависит от предыдущей, и если разные итерации раздать на выполнение разным потокам, то получится, что потоки несколько раз выполнят одну и ту же итерацию, т.к. в качестве источника результатов предыдущего шага они будут использовать одну и ту же строку матрицы d (ну или состояние массива, если используется оптимизированная реализация).

Однако внутри каждой итерации находится цикл, который позволяет проходить по всем вершинам графа. И работа, проводимая с каждой из этих вершин (в алгоритме эта вершина *from*) происходит независимо от остальных вершин, и порядок их не важен. Именно этот цикл и будет распараллелен с помощью технологий параллельного программирования.

Каждый поток будет получать вершину *from*, затем смотреть на все ее ребра, получая для каждого ребра вершину *to*, для которой будет проверяться возможность обновления расстояния $d[i][to]$ (ну или $d[to]$ в оптимизированной версии).

При этом стоит отметить, что может появиться гонка данных. Так, два разных потока, работая с разными вершинами *from1* и *from2*, могут одновременно попытаться обновить расстояние до вершины *to*. Не уменьшая общности будем полагать, что через вершину *from1* расстояние будет лучше. Тогда, если оба потока одновременно проверили условие на изменение расстояния до *to*, получается 2 случая:

1. Первым поменял поток, который использует *from1*.

Затем второй поток (поскольку он проверил условие со старым расстоянием) поменял расстояние на свое (несмотря на то, что оно хуже). Тогда, поскольку в этой итерации были изменения, будет проведена следующая, в которой какой-то поток будет снова работать с вершиной *from1*. И обратившись к вершине *to*, этот поток также проверит условие и заново обновит значение расстояния на свое.

2. Первым поменял поток, который использует *from2*.

Затем первый поток поменяет значение на свое (независимо от того, лучше расстояние через *from1* или хуже). А значит, что в эту итерацию у вершины *to* расстояние будет наилучшим.

Таким образом, гонки данных могут повлиять на количество итераций, но изменить результат работы алгоритма они не могут. Кроме того, общее число операций может увеличиться только если гонка данных произошла на последних итерациях, и вероятность такого события достаточно мала (с учетом того, что вершин в графе достаточно много). Таким образом введение критической секции, которая приведет к замедлению алгоритма на всех итерациях, ради того, чтобы общее число итераций могло стать на пару единиц меньше, просто нерационально.

Помимо распараллеливания самого алгоритма Мура, можно также распараллеливать сбор путей для вершин. Поскольку для каждой вершины определение пути происходит независимо от других, то разные потоки могут одновременно собирать пути для разных вершин, одновременно обращаясь к общему массиву *prev*, но не меняя ничего в нем. Таким образом в этом месте критические секции также не имеют смысла.

V. МЕТОД РЕШЕНИЯ

В данной работе будет использовано 3 реализации алгоритма, код которых можно увидеть в приложениях данного отчета:

1. Последовательная версия (Приложение №1);
2. Реализация с использованием OpenMP (Приложение №2);
3. Реализация с использованием TBB (Приложение №3);

Этап последовательной реализации присутствует во всех 3х программах с целью отслеживания работы во всех программах.

Входные параметры

Для удобства использования при запуске программы пользователя попросят ввести основные параметры:

1. В программах, которые используют технологии параллельного программирования, запрашивается число потоков, выполняющих алгоритм;
2. Тип используемого графа;
 - 2.1. Граф, представленный в примере работы алгоритма Мура;
 - 2.2. Сгенерированный случайным образом;
 - 2.3. Квадратный граф-решетка;
3. Если используется сгенерированный случайным образом граф, то будут запрошены порядок графа и его размер;
4. Если используется граф-решетка, то будут запрошены размеры графа в каждой из 2х плоскостей (x и y);
5. Номер стартовой вершины;
6. Необходимость вывода дополнительной информации. Она нужна в случае, если нужно выводить на консоль промежуточную информацию и результат работы программы. Это позволяет отслеживать корректность алгоритма. Без дополнительной информации будут выводиться только сообщения о завершении промежуточных этапов работы алгоритма (генерация графа, последовательная реализация, параллельная реализация).

Формат хранения графа

Для работы алгоритма Мура будем использовать матрицу смежности, а вершины графа обозначать цифрами от 1 до N . Матрица смежности в программах будет храниться в формате двумерного массива типа *char*. Предполагается, что вес ребра может быть числом в диапазоне от 1 до 100. Отсутствие ребра между двумя вершинами обозначается 0 в соответствующей ячейке матрицы.

$$graph_matrix[i][j] = \begin{cases} k, & k \in [1, 100] \text{ – существует ребро } (i, j) \text{ веса } k \\ 0, & \text{ребро } (i, j) \text{ отсутствует} \end{cases}$$

Так, для графа из примера (рис. 2) матрица будет выглядеть следующим образом:

<i>Matrix[i][j]</i>	1	2	3	4	5	6
1	0	3	0	2	0	7
2	0	0	0	0	0	0
3	8	0	0	1	4	0
4	0	0	0	0	0	1
5	0	0	0	2	0	5
6	0	0	0	0	1	0

Генерация графа

За генерацию графа отвечают 3 функции, используемые во всех 3х программах:

1. *GetCaseMatrix* – возвращает сгенерированную матрицу из примера (рис. 2);
2. *GenerateMatrix* – создает случайный граф заданного порядка и размера. В случае, если число ребер в графе достаточно маленькое, случайным образом генерирует существующие ребра, а если велико, то сначала алгоритм генерирует случайный полный граф, а затем случайно удаляет ребра.
3. *GetLatticeMatrix* – создает квадратный граф решетку заданных размеров. Предполагается, что граф-решетка имеет определенную структуру, представленную на рисунке 3. Веса ребер подбираются случайным образом.

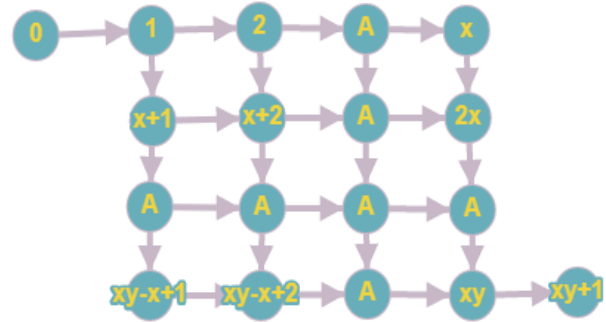


Рисунок 3. Структура графа-решетки для *GetLatticeMatrix*

Тип вызываемой функции определяется параметром, который запрашивается у пользователя во время начала работы с программой.

Последовательная часть

Блок последовательной части расположен в функции *SequentialResult*, работу которой можно разделить на несколько этапов:

1. Выделение памяти и установка начальных значений следующих массивов;
2. Нахождение массива расстояний до вершин и массива предшествующих вершин;
3. Восстановление путей;
4. Очистка памяти и возврат результата;

Во время первого этапа выделяется память под следующие массивы:

- *distance* – массив расстояний;
- *prev_vertex* – массив предшествующих вершин;
- *path* – массив векторов путей;

Также выполняется нулевая итерация и устанавливается наличие изменений:

$$distance[i] = \begin{cases} infinity, & i \neq start_vertex \\ 0, & i = start_vertex \end{cases}$$
$$check_compare = 0$$

Второй этап действует по следующему алгоритму:

Пока *check_compare* \neq 1 выполнять:

1. Установить *check_compare* = 1;
2. Выполнить для всех вершин *from*:
 - 2.1. Если *distance[from]* == *infinity*, перейти к следующей вершине *from*;
 - 2.2. Выполнить для всех вершин *to*:
 - 2.2.1. Если *graph_matrix[from][to]* == 0, перейти к следующей вершине *to*;
 - 2.2.2. Если *distance[to]* > *distance[from]* + *graph_matrix[from][to]*, выполнить:
 - 2.2.2.1. Установить *check_compare* = 0;
 - 2.2.2.2. Установить *distance[to]* = *distance[from]* + *graph_matrix[from][to]*;
 - 2.2.2.3. Установить *prev_vertex[to]* = *from*;

Третий этап работает по следующему алгоритму:

Выполнить для всех вершин *vertex*:

1. Если $distance[vertex] == infinity$, перейти к следующей вершине *vertex*;
2. Установить $cur_vertex = vertex$;
3. Пока последний элемент $path[vertex] \neq start_vertex$, выполнить:
 - 3.1. Добавить в $path[vertex]$ элемент cur_vertex ;
 - 3.2. Установить $cur_vertex = prev_vertex[cur_vertex]$;
4. Перевернуть $path[vertex]$;

Четвертый этап предполагает очистку памяти под *prev_vertex* и возврат массивов *distance* и *path*.

Параллельная часть с использованием OpenMP

Данный блок находится во второй программе в функции *ParallelResult*. Согласно схеме распараллеливания, алгоритм, использующийся в данной функции, совпадает с алгоритмом последовательной части за следующими исключениями:

1. Во втором этапе с помощью директивы *#pragma omp parallel for* будет распараллелен цикл по вершинам *from*.
2. В третьем этапе с помощью директивы *#pragma omp parallel for* будет распараллелен цикл по вершинам *vertex*.

Для того, чтобы добиться максимальной эффективности опытным путем была вычислена оптимальная стратегия распараллеливания директивы *parallel for*. В обеих частях по очереди были использованы стратегии *dynamic*, *static* и *guided* и замерены время работы этих блоков на 8 потоках для случайно сгенерированного графа порядка 10000 вершин и размера в 49995000 ребер (средняя заполненность графа). С целью исключения погрешностей каждое измерение проводилось по 5 раз и найдено было среднее значение времени:

Время (сек.)	2 этап	3 этап
guided	0,422232	0,0582432
dynamic	0,288347	0,0492641
static	0,356627	0,049742

Исходя из данной таблицы видно, что лучшее время достигается при использовании стратегии *dynamic* в обоих случаях. Поэтому в дальнейшем была использована эта стратегия.

Параллельная часть с использованием TBV

Данный блок находится во третьей программе в функции *ParallelResult*. Согласно схеме распараллеливания, алгоритм, использующийся в данной функции, совпадает с алгоритмом последовательной части за следующими исключениями:

1. Во втором этапе с помощью функции *tbb::parallel for* на одномерном итерационном пространстве от 0 до числа вершин будет распараллелен цикл по вершинам *from*.
2. В третьем этапе с помощью функции *tbb::parallel for* на одномерном итерационном пространстве от 0 до числа вершин будет распараллелен цикл по вершинам *vertex*.

Также, как и для OpenMP для TBV опытным путем был найдена оптимальная схема распараллеливания. Все замеры времени проводились с использованием 8 потоков на случайном графе порядка 10000 вершин и размером в 49995000 ребер. Все замеры проводились по 5 раз с целью исключения погрешностей.

В результате было получено следующее:

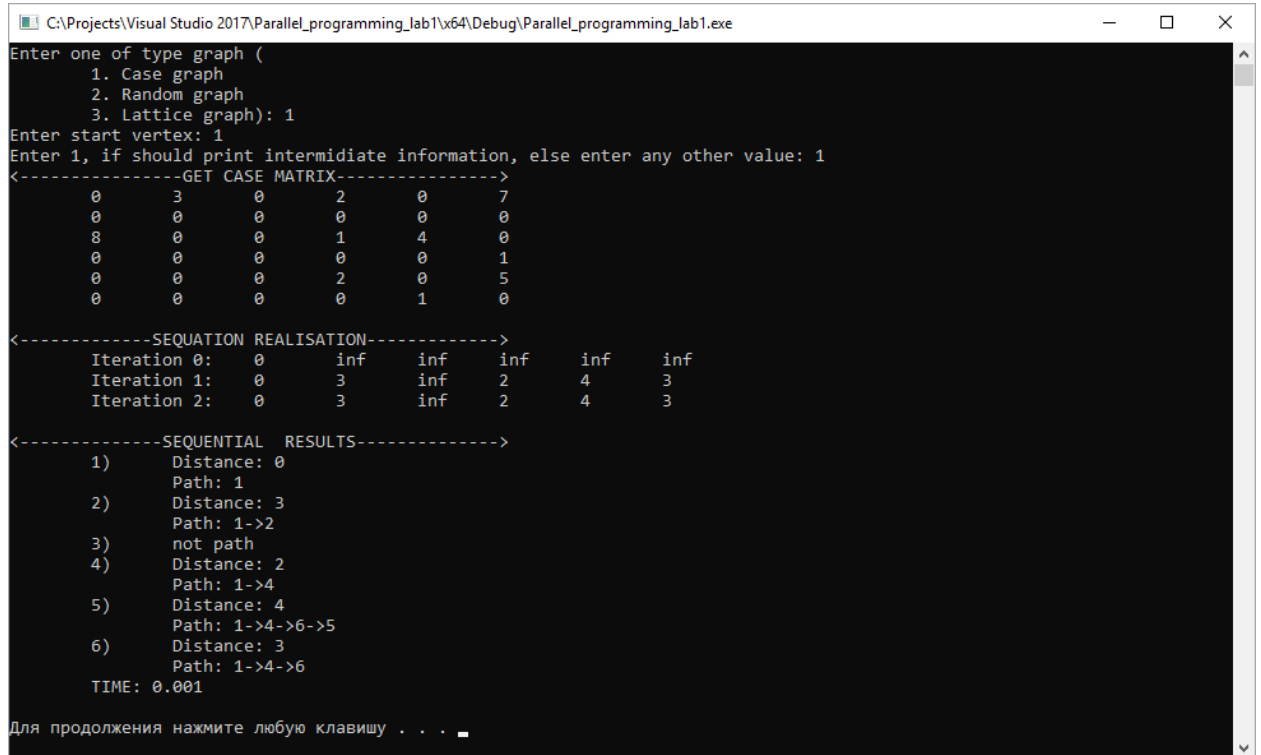
Время (сек.)	2 этап	3 этап
simple_partitioner	0,372006	0,0453194
auto_partitioner	0,3340446	0,043426

Исходя из результатов видно, что более оптимальной стратегией планирования является *auto_partitioner*.

VI. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Проверка корректности

Для проверки корректности алгоритма будет использоваться матрица из примера (рис. 2). Результат работы программ должен совпадать с результатом разбора алгоритма на данном графе.



```
C:\Projects\Visual Studio 2017\Parallel_programming_lab1\Debug\Parallel_programming_lab1.exe
Enter one of type graph (
  1. Case graph
  2. Random graph
  3. Lattice graph): 1
Enter start vertex: 1
Enter 1, if should print intermidiate information, else enter any other value: 1
<-----GET CASE MATRIX----->
      0      3      0      2      0      7
      0      0      0      0      0      0
      8      0      0      1      4      0
      0      0      0      0      0      1
      0      0      0      2      0      5
      0      0      0      0      1      0

<-----SEQUATION REALISATION----->
Iteration 0:  0      inf      inf      inf      inf      inf
Iteration 1:  0      3      inf      2      4      3
Iteration 2:  0      3      inf      2      4      3

<-----SEQUENTIAL RESULTS----->
1) Distance: 0
   Path: 1
2) Distance: 3
   Path: 1->2
3) not path
4) Distance: 2
   Path: 1->4
5) Distance: 4
   Path: 1->4->6->5
6) Distance: 3
   Path: 1->4->6
TIME: 0.001

Для продолжения нажмите любую клавишу . . .
```

Рисунок 4. Результат работы алгоритма на примере в первой программе

Исходя из результатов работы алгоритма на примере в последовательной части (рис. 4) видно, что результат программы полностью совпадает с результатом, полученным во время разбора алгоритма.

В остальных программах помимо последовательной части реализована и параллельная, и по рис. 5 и 6 видно, что алгоритм также корректен на них.

```

C:\Projects\Visual Studio 2017\Parallel_programming_lab2\Debug\Parallel_programming_lab2.exe
Enter one of type graph (
  1. Case graph
  2. Random graph
  3. Lattice graph): 1
Enter count of threads: 1
Enter start vertex: 1
Enter 1, if should print intermediate information, else enter any other value: 1
<-----GET CASE MATRIX----->
  0      3      0      2      0      7
  0      0      0      0      0      0
  8      0      0      1      4      0
  0      0      0      0      0      1
  0      0      0      2      0      5
  0      0      0      0      1      0

<-----PARALLEL REALISATION----->
  Iteration 0:  0      inf      inf      inf      inf      inf
  Iteration 1:  0      3      inf      2      4      3
  Iteration 2:  0      3      inf      2      4      3

<-----PARALLEL RESULTS----->
  1) Distance: 0
    Path: 1
  2) Distance: 3
    Path: 1->2
  3) not path
  4) Distance: 2
    Path: 1->4
  5) Distance: 4
    Path: 1->4->6->5
  6) Distance: 3
    Path: 1->4->6
  TIME: 0.00122823

<-----SEQUATION REALISATION----->
  Iteration 0:  0      inf      inf      inf      inf      inf
  Iteration 1:  0      3      inf      2      4      3
  Iteration 2:  0      3      inf      2      4      3

<-----SEQUENTIAL RESULTS----->
  1) Distance: 0
    Path: 1
  2) Distance: 3
    Path: 1->2
  3) not path
  4) Distance: 2
    Path: 1->4
  5) Distance: 4
    Path: 1->4->6->5
  6) Distance: 3
    Path: 1->4->6
  TIME: 0.00243029

Acceleration: 1.97869
Для продолжения нажмите любую клавишу . . .

```

Рисунок 5. Результат работы алгоритма на примере во второй программе


```

C:\Projects\Visual Studio 2017\Parallel_programming_lab3\x64\Debug\Parallel_programming_lab3.exe
Enter one of type graph (
  1. Case graph
  2. Random graph
  3. Lattice graph): 1
Enter count of threads: 1
Enter start vertex: 1
Enter 1, if should print intermediate information, else enter any other value: 1
<-----GET CASE MATRIX----->
  0      3      0      2      0      7
  0      0      0      0      0      0
  8      0      0      1      4      0
  0      0      0      0      0      1
  0      0      0      2      0      5
  0      0      0      0      1      0

<-----PARALLEL REALISATION----->
  Iteration 0:  0      inf      inf      inf      inf      inf
  Iteration 1:  0      3      inf      2      4      3
  Iteration 2:  0      3      inf      2      4      3

<-----PARALLEL RESULTS----->
  1) Distance: 0
    Path: 1
  2) Distance: 3
    Path: 1->2
  3) not path
  4) Distance: 2
    Path: 1->4
  5) Distance: 4
    Path: 1->4->6->5
  6) Distance: 3
    Path: 1->4->6
  TIME: 0.00132608

<-----SEQUATION REALISATION----->
  Iteration 0:  0      inf      inf      inf      inf      inf
  Iteration 1:  0      3      inf      2      4      3
  Iteration 2:  0      3      inf      2      4      3

<-----SEQUENTIAL RESULTS----->
  1) Distance: 0
    Path: 1
  2) Distance: 3
    Path: 1->2
  3) not path
  4) Distance: 2
    Path: 1->4
  5) Distance: 4
    Path: 1->4->6->5
  6) Distance: 3
    Path: 1->4->6
  TIME: 0.00271758

Acceleration: 2.04934
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6. Результат работы алгоритма на примере в третьей программе

Анализ сложности алгоритма

Для проверки теоретической сложности алгоритма была протестирована зависимость времени работы первой программы от порядка и размера графа. Для этого эта программа запускалась с различными параметрами случайного графа.

Если граф имеет N вершин, то максимально возможное число вершин в этом графе $N(N-1)$, поэтому число вершин для графов было вычислено в зависимости от числа ребер.

Время (сек.)	N=10	N=50	N=100	N=500	N=1000	N=5000	N=10000
0	0	0	0,001	0,001	0,001	0,004	0,007
$\frac{1}{4} N(N-1)$	0	0,001	0,002	0,015	0,037	0,537	1,538
$\frac{1}{2} N(N-1)$	0	0,001	0,001	0,016	0,041	0,599	2,31
$\frac{3}{4} N(N-1)$	0	0,002	0,001	0,011	0,033	0,47	1,203
$N(N-1)$	0	0,001	0,001	0,009	0,021	0,296	0,751

В результате получаются следующие графики:

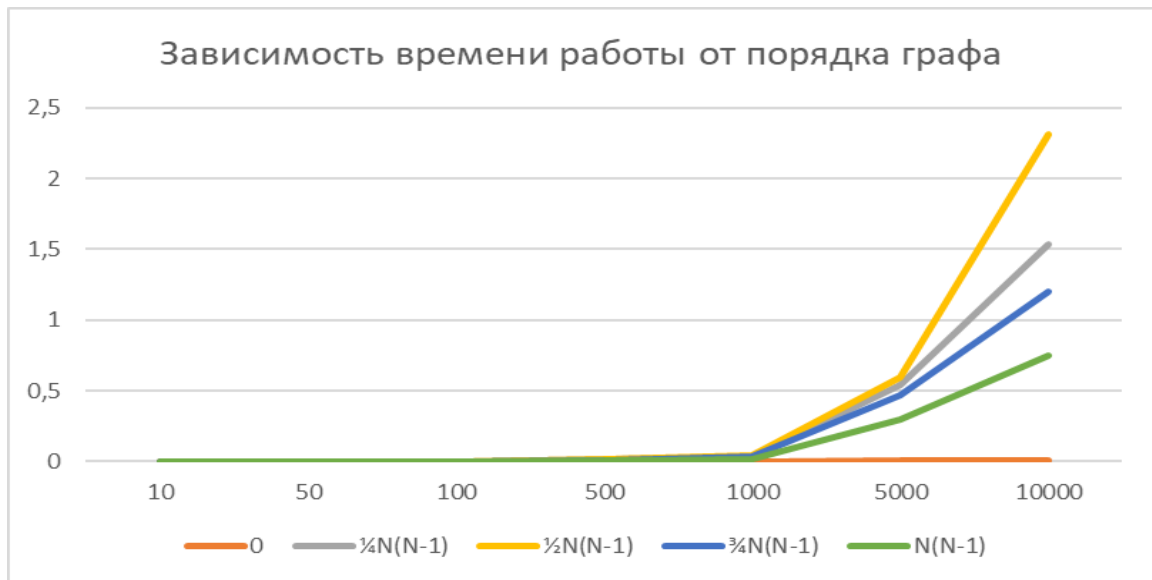


Рисунок 7. Зависимость времени работы от порядка графа

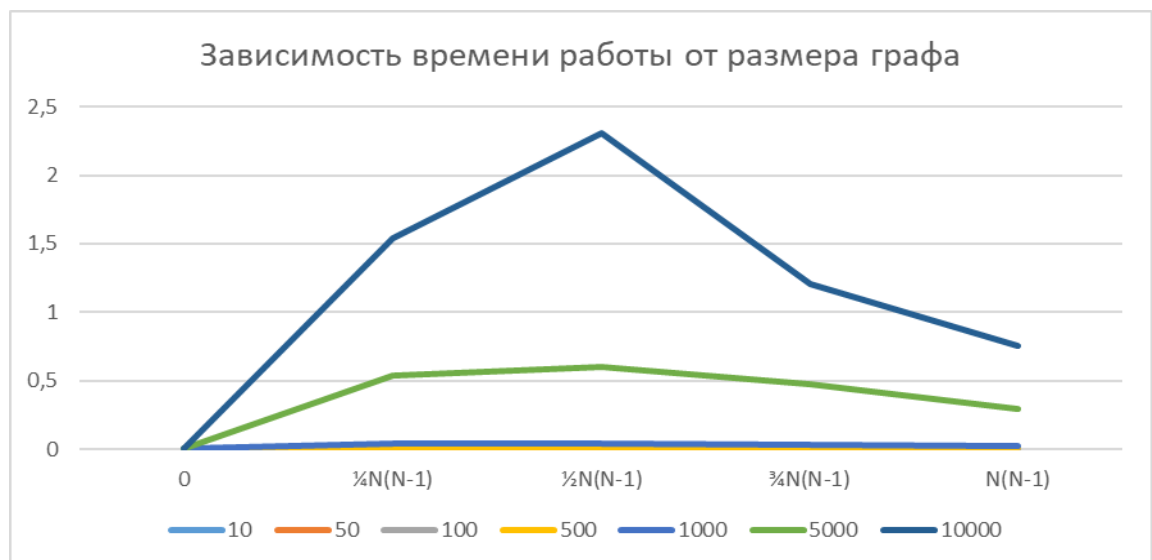


Рисунок 8. Зависимость времени работы от размера графа

Исходя из графиков видно, что наибольшее время работы достигается при средней заполненности графа. В тоже время с увеличением порядка графа, время работы растет. Это происходит потому, что при достаточно маленьком числе ребер вероятность нахождения вершин, до которых нельзя добраться, больше, а значит меньше вершин в конце концов будет обработано алгоритмом, из-за чего его время работы уменьшается. В тоже время при большом количестве ребер все больше вершин обрабатывается за одну итерацию алгоритма и с высокой вероятностью оптимальный путь до некоторой вершины будет найден быстрее, чем алгоритм пройдет по всем возможным ребрам. Получается, в этом случае итераций в алгоритме будет меньше, а значит время работы также будет не таким большим.

Что касается оптимального порядка графа, то при большом числе вершин возникает ситуация, когда компьютеру сложно найти непрерывные блоки памяти для размещения массивов. Поэтому при очень больших размерах графа программа может не заработать из-за отсутствия памяти. Так, на тестируемом компьютере программа не может обработать граф порядка 20000 вершин. Кроме того, с ростом числа вершин, увеличивается время генерации графа (например, на 10000 вершинах с средней заполненностью генерация графа занимает порядка 30 сек., что много больше, чем работа самого алгоритма). Поэтому при сравнении реализаций между собой будет использоваться случайно сгенерированный граф порядка 10000 вершин и размером в 49995000 ребер.

Результат работы параллельных частей

При сравнении работы параллельных реализаций алгоритма помимо времени работы также замерялось ускорение по отношению к последовательной части. Тестирование проводилось в зависимости от числа потоков, выполняющих распараллеливание. В результате получаются следующие цифры:

Время (сек.)	1	2	4	8	16	32	64
OpenMP	2,24436	1,18508	0,661937	0,431893	0,424682	0,446107	0,448772
TBB	2,50761	1,2811	0,728852	0,473628	0,473654	0,499502	0,514565

Ускорение	1	2	4	8	16	32	64
OpenMP	0,98418	1,94013	3,49682	5,35454	5,41474	5,17082	5,13465
TBB	0,902365	1,74564	3,26145	4,72497	4,66853	4,48486	4,35728

По данным таблицам получаются следующие графики:

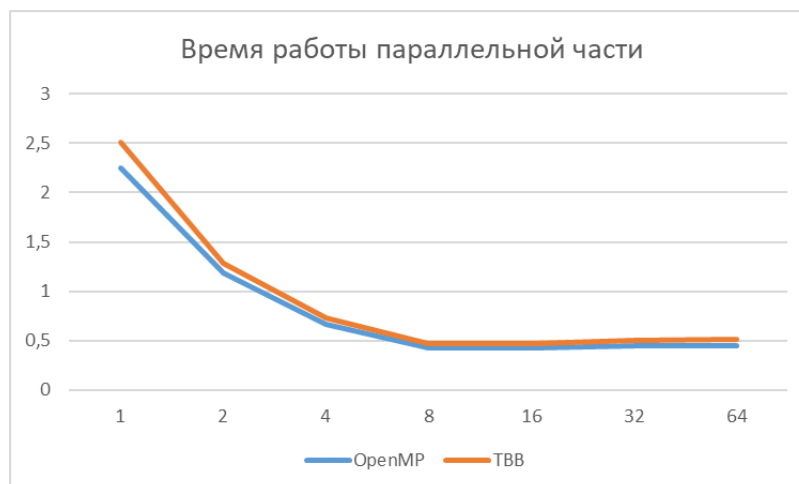


Рисунок 9. Время работы параллельной части на случайном графе

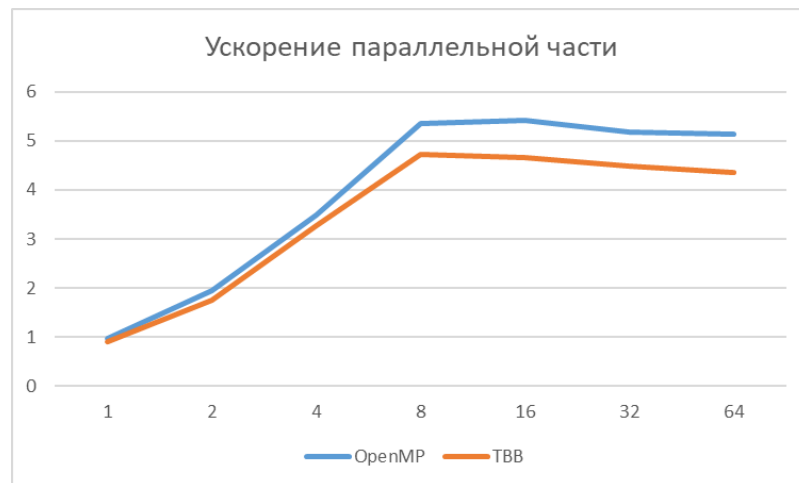


Рисунок 10. Ускорение параллельной части на случайном графе

По графикам явно видна эффективность обеих технологий распараллеливания при числе потоков меньших 16. Однако с их дальнейшим ростом, ускорение и время работы остаются примерно на том же уровне. Это происходит потому, что на тестируемом компьютере 8 логических ядер, и при большом числе потоков, им приходится ждать освобождения процессора, что приводит к неэффективности их использования.

Для того, чтобы лучше оценить ускорение и время работы различных технологий, алгоритм был протестирован на квадратном графе решетке. Дело в том, что в сгенерированном графе присутствует элемент случайности. Так, мы не можем точно утверждать число итераций в алгоритме, а значит в результатах работы может быть большой разброс. Так, при тестировании на OpenMP на 8 потоках графа с одинаковыми параметрами ускорение принимало значение от 4 до 6.

Квадратный граф решетка отличается тем, что в нем, независимо от весов ребер максимальное число вершин всегда будет одинаковым. Так, в графе размером $x \times y$ длина пути между первой и последней вершиной будет составлять $x + y$.

В качестве входных параметров использовались размеры графа 100 на 100 для того, чтобы общее число вершин было порядка 10000 (достаточно большое для оценки).

Получившиеся таблицы выглядят следующим образом:

Время (сек.)	1	2	4	8	16	32	64
OpenMP	1,69057	0,987399	0,863662	0,745563	0,738392	0,764114	0,768334
TBB	1,44019	0,863615	0,808979	1,04616	1,16257	1,29158	1,24666

Ускорение	1	2	4	8	16	32	64
OpenMP	0,969141	1,63854	1,93758	2,19575	2,19767	2,21808	2,24997
TBB	0,973467	1,62877	1,77964	1,35148	1,21084	1,13621	1,13881

На основе таблиц были сформированы следующие графики:

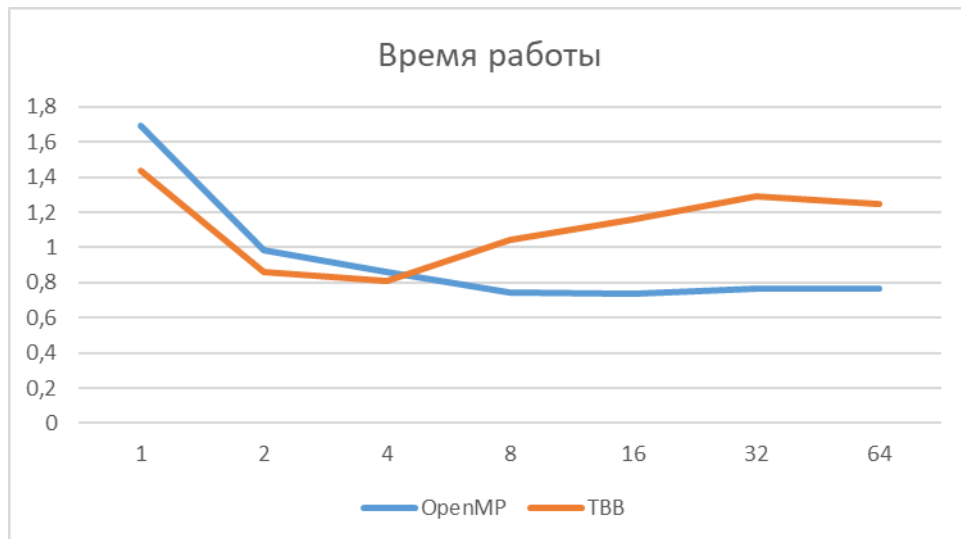


Рисунок 11. Время работы параллельной части на графе-решетке

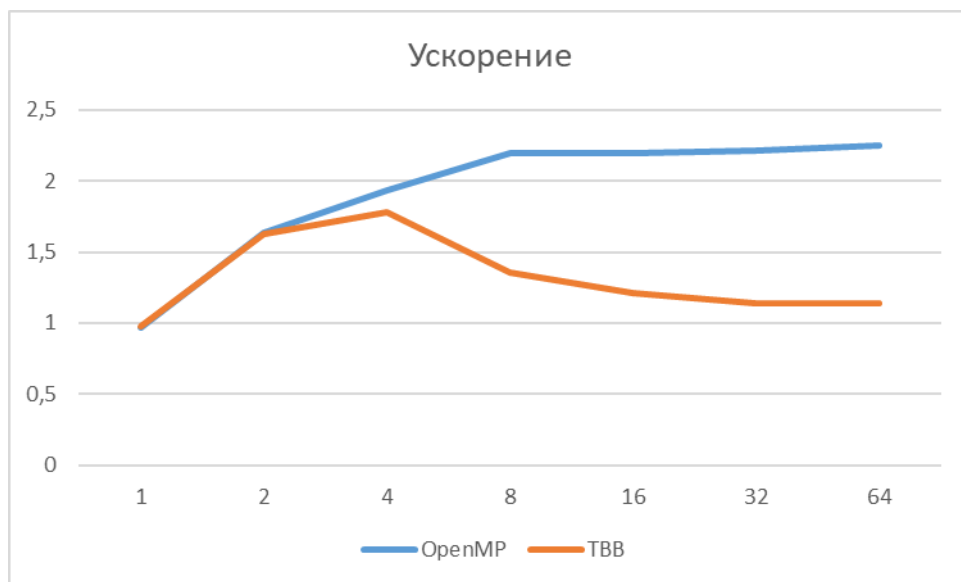


Рисунок 12. Ускорение параллельной части на графе-решетке

На основе полученных данных можно сделать следующие выводы:

1. Использование технологий параллельного программирования с большим числом потоков не так эффективно на графе-решетке. Это происходит потому, что несмотря на фиксированное число итераций $x + y \approx 2\sqrt{N}$, это число достаточно маленькое (по сравнению с максимальным числом в N итераций, где N – число вершин). И ускорение уменьшается из-за накладных расходов, на которые тратится тоже самое количество времени. Также стоит отметить то, сколько вершин было обработано на каждой итерации. Поскольку от каждой вершины исходит по 1-2 ребра, то большинство итераций распараллеливаемого цикла проходит впустую, в большинстве случаев обращается внимание либо на уже обработанные вершины, либо на те, до которых пути еще найдено не было. А проверка данного условия занимает не так много времени по сравнению с накладными расходами на выделение итерации потоку.
2. Имеется существенная разница между работами технологий TBB и OpenMP. Если обратить внимание на графики, становится понятно, что время работы TBB

увеличивается с ростом числа потоков по сравнению с OpenMP. Здесь все кроется в разных стратегиях распараллеливания. Технология OpenMP с стратегией *dynamic* отдает итерацию потоку по мере его обращения. И ему нет особой разницы между числом потоков, ведь в итоге будет запрошено одинаковое число итераций. Что касается TBB, то перед работой потоков эта технология разделяет итерационное пространство, а стратегия *auto_partitioner* старается снизить накладные расходы на организацию вычислений. И чем больше потоков используется, тем сложнее это сделать.

VII. ВЫВОДЫ

На основе полученных результатов можно отметить следующее:

1. Технологии OpenMP и TBB позволяют в несколько раз ускорить время работы программы при правильно подобранном числе потоков;
2. Данные технологии имеют накладные расходы на организацию параллелизма, которые увеличиваются с ростом числа потоков и размеров обрабатываемой информации;
3. Ускорение программы с помощью параллельных вычислений ограничено размером последовательной части. Поэтому ускорение не будет увеличиваться пропорционально росту числа потоков. Кроме того, здесь накладываются ограничения вычислительной машины, у которой ограниченное число ядер для работы с потоками;
4. Ускорение программы также изменяется с ростом размеров вычислений. Опять же, из-за присутствия последовательной части, этот рост не прямо пропорционален. Кроме того, из-за ограничений вычислительной системы, время последовательной части также увеличивается (например, за счет выделения памяти больших размеров). Это приводит к тому, что при больших размерах вычислений ускорение программы также может падать;
5. Имеются существенные различия между организацией распараллеливания технология OpenMP и TBB. Несмотря на то, что в среднем их время работы примерно одинаково, TBB более чувствителен к числу потоков. Это стоит учитывать при выборе технологии распараллеливания;
6. Для вычислений не всегда оптимально использовать число потоков, равное числу ядер процессоров. Это также зависит от обрабатываемых данных. Поэтому прежде, чем выбирать число потоков, необходимо проанализировать ход распараллеливания на различных вариантах входных параметров. Возможно, имеются ограничения на них, которые сделают неэффективным использование определенного числа потоков;

VIII. ИСТОЧНИКИ

1. Википедия – свободная энциклопедия
 - 1.1. Теория графов [Электронный ресурс]
https://ru.wikipedia.org/wiki/Теория_графов
 - 1.2. Задача о кратчайшем пути [Электронный ресурс]
https://ru.wikipedia.org/wiki/Задача_о_кратчайшем_пути
 - 1.3. Алгоритм Беллмана-Форда-Мура [Электронный ресурс]
https://ru.wikipedia.org/wiki/Алгоритм_Беллмана—Форда
2. Kvodo.ru – алгоритм Беллмана-Форда-Мура [Электронный ресурс]
<http://kvodo.ru/bellman-ford-algorithm.html>
3. docs.microsoft.com – Директивы OpenMP [Электронный ресурс]
<https://docs.microsoft.com/ru-ru/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019>
4. software.intel.com – Справочник разработчика TBB [Электронный ресурс]
https://software.intel.com/en-us/node/506130?_ga=2.201389848.827660588.1559131244-125401665.1557049039

IX. ПРИЛОЖЕНИЯ

Приложение №1

// Copyright 2019 Mezina Margarita

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>

#define PATH_INFINITY ~(unsigned int)0

typedef unsigned char u_char;
typedef unsigned int u_int;

struct res_format {
    u_int* distance;
    std::vector<u_int>* path;
};

/* GENERATE GRAPH MATRIX */
u_char** GetCaseMatrix(bool writing_work) {
    std::cout << "<-----GET CASE MATRIX----->\n";
    u_char case_matrix[6][6] = { { 0, 3, 0, 2, 0, 7 },
                                   { 0, 0, 0, 0, 0, 0 },
                                   { 8, 0, 0, 1, 4, 0 },
                                   { 0, 0, 0, 0, 0, 1 },
                                   { 0, 0, 0, 2, 0, 5 },
                                   { 0, 0, 0, 0, 1, 0 } };

    u_char** graph_matrix = new u_char*[6];
    for (u_int i = 0; i < 6; ++i) {
        graph_matrix[i] = new u_char[6];
        for (u_int j = 0; j < 6; ++j)
            graph_matrix[i][j] = case_matrix[i][j];
    }

    // Write
    if (writing_work) {
        for (u_int x = 0; x < 6; ++x) {
            for (u_int y = 0; y < 6; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    return graph_matrix;
}

u_char** GenerateMatrix(u_int vertex_count, u_int edges_count, bool writing_work) {
    std::cout << "<-----GENERATE MATRIX----->\n";

    // Memory allocation matrix
    u_char** graph_matrix = new u_char*[vertex_count];
    for (u_int i = 0; i < vertex_count; ++i) {
        graph_matrix[i] = new u_char[vertex_count];
    }

    // Generate matrix
    if (edges_count <= vertex_count * (vertex_count - 1) / 2) {
        for (u_int i = 0; i < vertex_count; ++i) {
            for (u_int j = 0; j < vertex_count; ++j)
                graph_matrix[i][j] = 0;
        }
    }
}
```

```

    }
    for (u_int i = 0; i < edges_count; ++i) {
        u_int x, y;
        do {
            x = std::rand() % vertex_count;
            y = std::rand() % vertex_count;
        } while (x == y || graph_matrix[x][y] != 0);
        u_char value = std::rand() % 100 + 1;
        graph_matrix[x][y] = value;
    }
}
else {
    for (u_int x = 0; x < vertex_count; ++x) {
        for (u_int y = 0; y < vertex_count; ++y)
            graph_matrix[x][y] = (x != y) ? std::rand() % 100 + 1 : 0;
    }
    for (u_int i = 0; i < vertex_count * (vertex_count - 1) - edges_count; ++i)
    {
        u_int x, y;
        do {
            x = std::rand() % vertex_count;
            y = std::rand() % vertex_count;
        } while (graph_matrix[x][y] == 0);
        graph_matrix[x][y] = 0;
    }
}

// Write
if (writing_work) {
    for (u_int x = 0; x < vertex_count; ++x) {
        for (u_int y = 0; y < vertex_count; ++y)
            std::cout << "\t" << (u_int)(graph_matrix[x][y]);
        std::cout << "\n";
    }
    std::cout << "\n";
}
return graph_matrix;
}
u_char** GetLatticeMatrix(u_int x_size, u_int y_size, bool writing_work) {
    std::cout << "<-----GET LATTICE MATRIX----->\n";
    u_char** graph_matrix = new u_char*[x_size * y_size + 2];
    for (u_int i = 0; i < x_size * y_size + 2; ++i)
        graph_matrix[i] = new u_char[x_size * y_size + 2];
    for (int i = 0; i < x_size * y_size + 2; ++i) {
        for (int j = 0; j < x_size * y_size + 2; ++j) {
            graph_matrix[i][j] = 0;
        }
    }
    for (int i = 1; i <= y_size; ++i) {
        for (int j = 1; j <= x_size; ++j) {
            int vertex = (i - 1) * x_size + j;
            if (j != x_size) graph_matrix[vertex][vertex + 1] = std::rand() % 100
+ 1;
            if (i != y_size) graph_matrix[vertex][vertex + x_size] = std::rand()
% 100 + 1;
        }
    }
    graph_matrix[0][1] = std::rand() % 100 + 1;
    graph_matrix[x_size * y_size][x_size * y_size + 1] = std::rand() % 100 + 1;
    if (writing_work) {
        for (u_int x = 0; x < x_size * y_size + 2; ++x) {
            for (u_int y = 0; y < x_size * y_size + 2; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
    }
}

```

```

    }
    std::cout << "\n";
}
return graph_matrix;
}

/* SEQUENTIAL REALISATION */
res_format SequentialResult(u_char** graph_matrix, u_int vertex_count, u_int
start_vertex, bool writing_work) {
    std::cout << "<-----SEQUATION REALISATION----->\n";

    // Memory allocation
    u_int* distance = new u_int[vertex_count];
    u_int* prev_vertex = new u_int[vertex_count];
    for (u_int i = 0; i < vertex_count; ++i)
        distance[i] = prev_vertex[i] = PATH_INFINITY;
    distance[start_vertex] = 0;
    prev_vertex[start_vertex] = start_vertex;
    bool check_compare = false;
    if (writing_work) {
        std::cout << "\tIteration 0:\t";
        for (u_int i = 0; i < vertex_count; ++i) {
            if (distance[i] == PATH_INFINITY)
                std::cout << "inf\t";
            else
                std::cout << distance[i] << "\t";
        }
        std::cout << "\n";
    }

    int iter = 0;
    // Find distance matrix
    while (!check_compare) {
        check_compare = 1;
        // Update
        for (u_int from = 0; from < vertex_count; ++from) {
            if (distance[from] == PATH_INFINITY) continue;
            for (u_int to = 0; to < vertex_count; ++to) {
                if (graph_matrix[from][to] == 0) continue;
                if (distance[to] > distance[from] + graph_matrix[from][to]) {
                    check_compare = 0;
                    distance[to] = distance[from] + graph_matrix[from][to];
                    prev_vertex[to] = from;
                }
            }
        }
        // Write
        if (writing_work) {
            std::cout << "\tIteration " << (++iter) << ":\t";
            for (unsigned int i = 0; i < vertex_count; ++i) {
                if (distance[i] == PATH_INFINITY)
                    std::cout << "inf\t";
                else
                    std::cout << distance[i] << "\t";
            }
            std::cout << "\n";
        }
    }

    // Find vectors_paths
    std::vector<u_int>* path = new std::vector<u_int>[vertex_count];
    for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
        if (distance[vertex] == PATH_INFINITY) continue;
        u_int cur_vertex = vertex;

```

```

        do {
            path[vertex].push_back(cur_vertex);
            cur_vertex = prev_vertex[cur_vertex];
        } while (path[vertex].back() != start_vertex);
        reverse(path[vertex].begin(), path[vertex].end());
    }

    // Return
    if (writing_work) std::cout << "\n";
    res_format result = { distance, path };
    delete[] prev_vertex;
    return result;
}

/* PRINT RESULT */
void PrintResults(res_format result, u_int vertex_count, bool writing_work) {
    std::cout << "<-----SEQUENTIAL RESULTS----->\n";

    // Write
    u_int* distance = result.distance;
    std::vector<u_int>* path = result.path;
    if (writing_work) {
        for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
            std::cout << "\t" << vertex + 1 << ")\t";
            if (distance[vertex] == PATH_INFINITY) {
                std::cout << "not path" << std::endl;
                continue;
            }
            std::cout << "Distance: " << distance[vertex] << std::endl;
            std::cout << "\t\tPath: ";
            std::cout << path[vertex][0] + 1;
            for (unsigned int i = 1; i < path[vertex].size(); ++i)
                std::cout << "->" << path[vertex][i] + 1;
            std::cout << "\n";
        }
    }
}

int main(int argc, char** argv) {
    // Graph params
    u_int graph_type = 0;
    u_int vertex_count = 0, edges_count = 0;
    u_int x_size = 0, y_size = 0;
    u_char** graph_matrix = nullptr;
    bool writing_work = 0;
    u_int start_vertex = 0;

    // Enter graph params
    std::cout << "Enter one of type graph (\n";
    std::cout << "\t1. Case graph\n";
    std::cout << "\t2. Random graph\n";
    std::cout << "\t3. Lattice graph): ";
    std::cin >> graph_type;
    switch (graph_type) {
        case 1: break;
        case 2:
            std::cout << "Enter count of vertexes: ";
            std::cin >> vertex_count;
            std::cout << "Enter count of edges: ";
            std::cin >> edges_count;
            break;
        case 3:
            std::cout << "Enter x-size: ";
            std::cin >> x_size;

```

```

        std::cout << "Enter y_size: ";
        std::cin >> y_size;
        break;
    }
    std::cout << "Enter start vertex: ";
    std::cin >> start_vertex;
    start_vertex--;
    std::cout << "Enter 1, if should print intermediate information, else enter any
other value: ";
    std::string s;
    std::cin >> s;
    if (s == "1") writing_work = true;

    // Generate matrix
    srand(static_cast<int>(time(0)));
    switch (graph_type) {
    case 1:
        graph_matrix = GetCaseMatrix(writing_work);
        vertex_count = 6;
        break;
    case 2:
        graph_matrix = GenerateMatrix(vertex_count, edges_count, writing_work);
        break;
    case 3:
        graph_matrix = GetLatticeMatrix(x_size, y_size, writing_work);
        vertex_count = x_size * y_size + 2;
        break;
    }

    // Sequential realisation
    u_int seq_start_time, seq_finish_time;
    seq_start_time = clock();
    res_format seq_result = SequentialResult(graph_matrix, vertex_count, start_vertex,
writing_work);
    seq_finish_time = clock();

    // Write sequential results
    PrintResults(seq_result, vertex_count, writing_work);
    std::cout << "\tTIME: " << (seq_finish_time - seq_start_time)/1000.0 << "\n\n";

    // Free memory
    delete[] seq_result.distance;
    for (u_int i = 0; i < seq_result.path->size(); ++i)
        seq_result.path[i].clear();
    delete[] seq_result.path;

    system("pause");
    return 0;
}

```

Приложение №2

// Copyright 2019 Mezina Margarita

```

#include <omp.h>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>

#define PATH_INFINITY ~(unsigned int)0

typedef unsigned char u_char;

```

```

typedef unsigned int u_int;

struct res_format {
    u_int* distance;
    std::vector<u_int>* path;
};

/* GENERATE GRAPH MATRIX */
u_char** GetCaseMatrix(bool writing_work) {
    std::cout << "<-----GET CASE MATRIX----->\n";
    u_char case_matrix[6][6] = { { 0, 3, 0, 2, 0, 7 },
                                   { 0, 0, 0, 0, 0, 0 },
                                   { 8, 0, 0, 1, 4, 0 },
                                   { 0, 0, 0, 0, 0, 1 },
                                   { 0, 0, 0, 2, 0, 5 },
                                   { 0, 0, 0, 0, 1, 0 } };

    u_char** graph_matrix = new u_char*[6];
    for (u_int i = 0; i < 6; ++i) {
        graph_matrix[i] = new u_char[6];
        for (u_int j = 0; j < 6; ++j)
            graph_matrix[i][j] = case_matrix[i][j];
    }

    // Write
    if (writing_work) {
        for (u_int x = 0; x < 6; ++x) {
            for (u_int y = 0; y < 6; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    return graph_matrix;
}

u_char** GenerateMatrix(u_int vertex_count, u_int edges_count, bool writing_work) {
    std::cout << "<-----GENERATE MATRIX----->\n";

    // Memory allocation matrix
    u_char** graph_matrix = new u_char*[vertex_count];
    for (u_int i = 0; i < vertex_count; ++i) {
        graph_matrix[i] = new u_char[vertex_count];
    }

    // Generate matrix
    if (edges_count <= vertex_count * (vertex_count - 1) / 2) {
        for (u_int i = 0; i < vertex_count; ++i) {
            for (u_int j = 0; j < vertex_count; ++j)
                graph_matrix[i][j] = 0;
        }
        for (u_int i = 0; i < edges_count; ++i) {
            u_int x, y;
            do {
                x = std::rand() % vertex_count;
                y = std::rand() % vertex_count;
            } while (x == y || graph_matrix[x][y] != 0);
            u_char value = std::rand() % 100 + 1;
            graph_matrix[x][y] = value;
        }
    }
    else {
        for (u_int x = 0; x < vertex_count; ++x) {
            for (u_int y = 0; y < vertex_count; ++y)
                graph_matrix[x][y] = (x != y) ? std::rand() % 100 + 1 : 0;
        }
    }
}

```

```

    }
    for (u_int i = 0; i < vertex_count * (vertex_count - 1) - edges_count; ++i)
{
    u_int x, y;
    do {
        x = std::rand() % vertex_count;
        y = std::rand() % vertex_count;
    } while (graph_matrix[x][y] == 0);
    graph_matrix[x][y] = 0;
    }
}

// Write
if (writing_work) {
    for (u_int x = 0; x < vertex_count; ++x) {
        for (u_int y = 0; y < vertex_count; ++y)
            std::cout << "\t" << (u_int)(graph_matrix[x][y]);
        std::cout << "\n";
    }
    std::cout << "\n";
}
return graph_matrix;
}

u_char** GetLatticeMatrix(u_int x_size, u_int y_size, bool writing_work) {
    std::cout << "<-----GET LATTICE MATRIX----->\n";
    u_char** graph_matrix = new u_char*[x_size * y_size + 2];
    for (u_int i = 0; i < x_size * y_size + 2; ++i)
        graph_matrix[i] = new u_char[x_size * y_size + 2];
    for (int i = 0; i < x_size * y_size + 2; ++i) {
        for (int j = 0; j < x_size * y_size + 2; ++j) {
            graph_matrix[i][j] = 0;
        }
    }
    for (int i = 1; i <= y_size; ++i) {
        for (int j = 1; j <= x_size; ++j) {
            int vertex = (i - 1) * x_size + j;
            if (j != x_size) graph_matrix[vertex][vertex + 1] = std::rand() % 100
+ 1;
            if (i != y_size) graph_matrix[vertex][vertex + x_size] = std::rand()
% 100 + 1;
        }
    }
    graph_matrix[0][1] = std::rand() % 100 + 1;
    graph_matrix[x_size * y_size][x_size * y_size + 1] = std::rand() % 100 + 1;
    if (writing_work) {
        for (u_int x = 0; x < x_size * y_size + 2; ++x) {
            for (u_int y = 0; y < x_size * y_size + 2; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    return graph_matrix;
}

/* SEQUENTIAL REALISATION */
res_format SequentialResult(u_char** graph_matrix, u_int vertex_count, u_int
start_vertex, bool writing_work) {
    std::cout << "<-----SEQUATION REALISATION----->\n";

    // Memory allocation
    u_int* distance = new u_int[vertex_count];
    u_int* prev_vertex = new u_int[vertex_count];
    /*for (u_int i = 0; i < vertex_count; ++i)

```

```

        distance_matrix[0][i] = prev_vertex[i] = PATH_INFINITY;*/
for (u_int i = 0; i < vertex_count; ++i)
    distance[i] = prev_vertex[i] = PATH_INFINITY;
/*distance_matrix[0][start_vertex] = 0;*/
distance[start_vertex] = 0;
prev_vertex[start_vertex] = start_vertex;
bool check_compare = false;
if (writing_work) {
    std::cout << "\tIteration 0:\t";
    for (u_int i = 0; i < vertex_count; ++i) {
        if (distance[i] == PATH_INFINITY)
            std::cout << "inf\t";
        else
            std::cout << distance[i] << "\t";
    }
    std::cout << "\n";
}

int iter = 0;
// Find distance matrix
while (!check_compare) {
    check_compare = 1;
    // Update
    for (u_int from = 0; from < vertex_count; ++from) {
        if (distance[from] == PATH_INFINITY) continue;
        for (u_int to = 0; to < vertex_count; ++to) {
            if (graph_matrix[from][to] == 0) continue;
            if (distance[to] > distance[from] + graph_matrix[from][to]) {
                check_compare = 0;
                distance[to] = distance[from] + graph_matrix[from][to];
                prev_vertex[to] = from;
            }
        }
    }
    // Write
    if (writing_work) {
        std::cout << "\tIteration " << (++iter) << ":\t";
        for (unsigned int i = 0; i < vertex_count; ++i) {
            if (distance[i] == PATH_INFINITY)
                std::cout << "inf\t";
            else
                std::cout << distance[i] << "\t";
        }
        std::cout << "\n";
    }
}

// Find vectors_paths
std::vector<u_int>* path = new std::vector<u_int>(vertex_count);
for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
    if (distance[vertex] == PATH_INFINITY) continue;
    u_int cur_vertex = vertex;
    do {
        path[vertex].push_back(cur_vertex);
        cur_vertex = prev_vertex[cur_vertex];
    } while (path[vertex].back() != start_vertex);
    reverse(path[vertex].begin(), path[vertex].end());
}

// Return
if (writing_work) std::cout << "\n";
res_format result = { distance, path };
delete[] prev_vertex;
return result;

```



```

}

/* PARALLEL REALISATION */
res_format ParallelResult(u_char** graph_matrix, u_int vertex_count, u_int start_vertex,
bool writing_work) {
    std::cout << "<-----PARALLEL REALISATION----->\n";

    // Memory allocation
    u_int* distance = new u_int[vertex_count];
    u_int* prev_vertex = new u_int[vertex_count];
    for (int i = 0; i < static_cast<int>(vertex_count); ++i)
        distance[i] = prev_vertex[i] = PATH_INFINITY;
    distance[start_vertex] = 0;
    prev_vertex[start_vertex] = start_vertex;
    bool check_compare = false;
    if (writing_work) {
        std::cout << "\tIteration 0:\t";
        for (u_int i = 0; i < vertex_count; ++i) {
            if (distance[i] == PATH_INFINITY)
                std::cout << "inf\t";
            else
                std::cout << distance[i] << "\t";
        }
        std::cout << "\n";
    }

    // Find distance matrix
    int iter;
    /* double start_test_distance = omp_get_wtime(); */
    for (iter = 0; !check_compare; iter++) {
        check_compare = 1;
        // Update
#pragma omp parallel for schedule(dynamic)
        for (int from = 0; from < static_cast<int>(vertex_count); ++from) {
            if (distance[from] == PATH_INFINITY) continue;
            for (u_int to = 0; to < vertex_count; ++to) {
                if (graph_matrix[from][to] == 0) continue;
                if (distance[to] > distance[from] + graph_matrix[from][to]) {
                    check_compare = 0;
                    distance[to] = distance[from] + graph_matrix[from][to];
                    prev_vertex[to] = from;
                }
            }
        }
        // Write
        if (writing_work) {
            std::cout << "\tIteration " << (iter + 1) << ":\t";
            for (unsigned int i = 0; i < vertex_count; ++i) {
                if (distance[i] == PATH_INFINITY)
                    std::cout << "inf\t";
                else
                    std::cout << distance[i] << "\t";
            }
            std::cout << "\n";
        }
    }
    /* double finish_test_distance = omp_get_wtime();
    std::cout << "Test distance: " << finish_test_distance - start_test_distance <<
    std::endl; */

    // Find vectors_paths
    std::vector<u_int>* path = new std::vector<u_int>(vertex_count);
    /* double start_test_path = omp_get_wtime(); */
#pragma omp parallel for schedule(dynamic)

```

```

for (int vertex = 0; vertex < static_cast<int>(vertex_count); ++vertex) {
    if (distance[vertex] == PATH_INFINITY) continue;
    u_int cur_vertex = vertex;
    do {
        path[vertex].push_back(cur_vertex);
        cur_vertex = prev_vertex[cur_vertex];
    } while (path[vertex].back() != start_vertex);
    reverse(path[vertex].begin(), path[vertex].end());
}
/* double finish_test_path = omp_get_wtime();
std::cout << "Test path: " << finish_test_path - start_test_path << std::endl; */

// Return
if (writing_work) std::cout << "\n";
res_format result = { distance, path };
delete[] prev_vertex;
return result;
}

/* PRINT RESULT */
void PrintResults(res_format result, u_int vertex_count, bool writing_work, bool
is_sequential) {
    if (is_sequential)
        std::cout << "<-----SEQUENTIAL RESULTS----->\n";
    else
        std::cout << "<-----PARALLEL RESULTS----->\n";

    // Write
    u_int* distance = result.distance;
    std::vector<u_int>* path = result.path;
    if (writing_work) {
        for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
            std::cout << "\t" << vertex + 1 << "\t";
            if (distance[vertex] == PATH_INFINITY) {
                std::cout << "not path" << std::endl;
                continue;
            }
            std::cout << "Distance: " << distance[vertex] << std::endl;
            std::cout << "\t\tPath: ";
            std::cout << path[vertex][0] + 1;
            for (unsigned int i = 1; i < path[vertex].size(); ++i)
                std::cout << "->" << path[vertex][i] + 1;
            std::cout << "\n";
        }
    }
}

int main(int argc, char** argv) {
    // Graph params
    u_int graph_type = 0;
    u_int vertex_count = 0, edges_count = 0;
    u_int x_size = 0, y_size = 0;
    u_char** graph_matrix = nullptr;
    bool writing_work = 0;
    u_int start_vertex = 0;
    u_int threads_count = 0;

    // Enter graph params
    std::cout << "Enter one of type graph (\n";
    std::cout << "\t1. Case graph\n";
    std::cout << "\t2. Random graph\n";
    std::cout << "\t3. Lattice graph): ";
    std::cin >> graph_type;
    std::cout << "Enter count of threads: ";

```

```

std::cin >> threads_count;
switch (graph_type) {
case 1: break;
case 2:
    std::cout << "Enter count of vertexes: ";
    std::cin >> vertex_count;
    std::cout << "Enter count of edges: ";
    std::cin >> edges_count;
    break;
case 3:
    std::cout << "Enter x-size: ";
    std::cin >> x_size;
    std::cout << "Enter y_size: ";
    std::cin >> y_size;
    break;
}
std::cout << "Enter start vertex: ";
std::cin >> start_vertex;
start_vertex--;
std::cout << "Enter 1, if should print intermediate information, else enter any
other value: ";
std::string s;
std::cin >> s;
if (s == "1") writing_work = true;

// Generate matrix
srand(static_cast<int>(time(0)));
switch (graph_type) {
case 1:
    graph_matrix = GetCaseMatrix(writing_work);
    vertex_count = 6;
    break;
case 2:
    graph_matrix = GenerateMatrix(vertex_count, edges_count, writing_work);
    break;
case 3:
    graph_matrix = GetLatticeMatrix(x_size, y_size, writing_work);
    vertex_count = x_size * y_size + 2;
    break;
}
omp_set_num_threads(threads_count);

// Parallel realisation
double par_start_time, par_finish_time;
par_start_time = omp_get_wtime();
res_format par_result = ParallelResult(graph_matrix, vertex_count, start_vertex,
writing_work);
par_finish_time = omp_get_wtime();

// Write parallel results
PrintResults(par_result, vertex_count, writing_work, false);
std::cout << "\tTIME: " << (par_finish_time - par_start_time) << "\n\n";

// Free memory
delete[] par_result.distance;
for (u_int i = 0; i < par_result.path->size(); ++i)
    par_result.path[i].clear();
delete[] par_result.path;

// Sequential realisation
double seq_start_time, seq_finish_time;
seq_start_time = omp_get_wtime();
res_format seq_result = SequentialResult(graph_matrix, vertex_count, start_vertex,
writing_work);

```

```

seq_finish_time = omp_get_wtime();

// Write sequential results
PrintResults(seq_result, vertex_count, writing_work, true);
std::cout << "\tTIME: " << (seq_finish_time - seq_start_time) << "\n\n";

// Free memory
delete[] seq_result.distance;
for (u_int i = 0; i < seq_result.path->size(); ++i)
    seq_result.path[i].clear();
delete[] seq_result.path;

std::cout << "Acceleration: " << (seq_finish_time - seq_start_time) /
(par_finish_time - par_start_time) << "\n";

system("pause");
return 0;
}

```

Приложение №3

// Copyright 2019 Mezina Margarita

```

#include <omp.h>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>

#define PATH_INFINITY ~(unsigned int)0

typedef unsigned char u_char;
typedef unsigned int u_int;

struct res_format {
    u_int* distance;
    std::vector<u_int>* path;
};

/* GENERATE GRAPH MATRIX */
u_char** GetCaseMatrix(bool writing_work) {
    std::cout << "<-----GET CASE MATRIX----->\n";
    u_char case_matrix[6][6] = { { 0, 3, 0, 2, 0, 7 },
                                  { 0, 0, 0, 0, 0, 0 },
                                  { 8, 0, 0, 1, 4, 0 },
                                  { 0, 0, 0, 0, 0, 1 },
                                  { 0, 0, 0, 2, 0, 5 },
                                  { 0, 0, 0, 0, 1, 0 } };

    u_char** graph_matrix = new u_char*[6];
    for (u_int i = 0; i < 6; ++i) {
        graph_matrix[i] = new u_char[6];
        for (u_int j = 0; j < 6; ++j)
            graph_matrix[i][j] = case_matrix[i][j];
    }

    // Write
    if (writing_work) {
        for (u_int x = 0; x < 6; ++x) {
            for (u_int y = 0; y < 6; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
}

```

```

    }
    return graph_matrix;
}

u_char** GenerateMatrix(u_int vertex_count, u_int edges_count, bool writing_work) {
    std::cout << "<-----GENERATE MATRIX----->\n";

    // Memory allocation matrix
    u_char** graph_matrix = new u_char*[vertex_count];
    for (u_int i = 0; i < vertex_count; ++i) {
        graph_matrix[i] = new u_char[vertex_count];
    }

    // Generate matrix
    if (edges_count <= vertex_count * (vertex_count - 1) / 2) {
        for (u_int i = 0; i < vertex_count; ++i) {
            for (u_int j = 0; j < vertex_count; ++j)
                graph_matrix[i][j] = 0;
        }
        for (u_int i = 0; i < edges_count; ++i) {
            u_int x, y;
            do {
                x = std::rand() % vertex_count;
                y = std::rand() % vertex_count;
            } while (x == y || graph_matrix[x][y] != 0);
            u_char value = std::rand() % 100 + 1;
            graph_matrix[x][y] = value;
        }
    }
    else {
        for (u_int x = 0; x < vertex_count; ++x) {
            for (u_int y = 0; y < vertex_count; ++y)
                graph_matrix[x][y] = (x != y) ? std::rand() % 100 + 1 : 0;
        }
        for (u_int i = 0; i < vertex_count * (vertex_count - 1) - edges_count; ++i)
        {
            u_int x, y;
            do {
                x = std::rand() % vertex_count;
                y = std::rand() % vertex_count;
            } while (graph_matrix[x][y] == 0);
            graph_matrix[x][y] = 0;
        }
    }

    // Write
    if (writing_work) {
        for (u_int x = 0; x < vertex_count; ++x) {
            for (u_int y = 0; y < vertex_count; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    return graph_matrix;
}

u_char** GetLatticeMatrix(u_int x_size, u_int y_size, bool writing_work) {
    std::cout << "<-----GET LATTICE MATRIX----->\n";
    u_char** graph_matrix = new u_char*[x_size * y_size + 2];
    for (u_int i = 0; i < x_size * y_size + 2; ++i)
        graph_matrix[i] = new u_char[x_size * y_size + 2];
    for (int i = 0; i < x_size * y_size + 2; ++i) {
        for (int j = 0; j < x_size * y_size + 2; ++j) {
            graph_matrix[i][j] = 0;
        }
    }
}

```

```

    }
    for (int i = 1; i <= y_size; ++i) {
        for (int j = 1; j <= x_size; ++j) {
            int vertex = (i - 1) * x_size + j;
            if (j != x_size) graph_matrix[vertex][vertex + 1] = std::rand() % 100
+ 1;
            if (i != y_size) graph_matrix[vertex][vertex + x_size] = std::rand()
% 100 + 1;
        }
    }
    graph_matrix[0][1] = std::rand() % 100 + 1;
    graph_matrix[x_size * y_size][x_size * y_size + 1] = std::rand() % 100 + 1;
    if (writing_work) {
        for (u_int x = 0; x < x_size * y_size + 2; ++x) {
            for (u_int y = 0; y < x_size * y_size + 2; ++y)
                std::cout << "\t" << (u_int)(graph_matrix[x][y]);
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    return graph_matrix;
}

/* SEQUENTIAL REALISATION */
res_format SequentialResult(u_char** graph_matrix, u_int vertex_count, u_int
start_vertex, bool writing_work) {
    std::cout << "<-----SEQUATION REALISATION----->\n";

    // Memory allocation
    u_int* distance = new u_int[vertex_count];
    u_int* prev_vertex = new u_int[vertex_count];
    /*for (u_int i = 0; i < vertex_count; ++i)
        distance_matrix[0][i] = prev_vertex[i] = PATH_INFINITY;*/
    for (u_int i = 0; i < vertex_count; ++i)
        distance[i] = prev_vertex[i] = PATH_INFINITY;
    /*distance_matrix[0][start_vertex] = 0;*/
    distance[start_vertex] = 0;
    prev_vertex[start_vertex] = start_vertex;
    bool check_compare = false;
    if (writing_work) {
        std::cout << "\tIteration 0:\t";
        for (u_int i = 0; i < vertex_count; ++i) {
            if (distance[i] == PATH_INFINITY)
                std::cout << "inf\t";
            else
                std::cout << distance[i] << "\t";
        }
        std::cout << "\n";
    }

    int iter = 0;
    // Find distance matrix
    while (!check_compare) {
        check_compare = 1;
        // Update
        for (u_int from = 0; from < vertex_count; ++from) {
            if (distance[from] == PATH_INFINITY) continue;
            for (u_int to = 0; to < vertex_count; ++to) {
                if (graph_matrix[from][to] == 0) continue;
                if (distance[to] > distance[from] + graph_matrix[from][to]) {
                    check_compare = 0;
                    distance[to] = distance[from] + graph_matrix[from][to];
                    prev_vertex[to] = from;
                }
            }
        }
        ++iter;
    }
}

```

```

    }
}

// Write
if (writing_work) {
    std::cout << "\tIteration " << (++iter) << ":\t";
    for (unsigned int i = 0; i < vertex_count; ++i) {
        if (distance[i] == PATH_INFINITY)
            std::cout << "inf\t";
        else
            std::cout << distance[i] << "\t";
    }
    std::cout << "\n";
}

// Find vectors_paths
std::vector<u_int>* path = new std::vector<u_int>[vertex_count];
for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
    if (distance[vertex] == PATH_INFINITY) continue;
    u_int cur_vertex = vertex;
    do {
        path[vertex].push_back(cur_vertex);
        cur_vertex = prev_vertex[cur_vertex];
    } while (path[vertex].back() != start_vertex);
    reverse(path[vertex].begin(), path[vertex].end());
}

// Return
if (writing_work) std::cout << "\n";
res_format result = { distance, path };
delete[] prev_vertex;
return result;
}

/* PARALLEL REALISATION */
res_format ParallelResult(u_char** graph_matrix, u_int vertex_count, u_int start_vertex,
bool writing_work) {
    std::cout << "<-----PARALLEL REALISATION----->\n";

    // Memory allocation
    u_int* distance = new u_int[vertex_count];
    u_int* prev_vertex = new u_int[vertex_count];
    for (int i = 0; i < static_cast<int>(vertex_count); ++i)
        distance[i] = prev_vertex[i] = PATH_INFINITY;
    distance[start_vertex] = 0;
    prev_vertex[start_vertex] = start_vertex;
    bool check_compare = false;
    if (writing_work) {
        std::cout << "\tIteration 0:\t";
        for (u_int i = 0; i < vertex_count; ++i) {
            if (distance[i] == PATH_INFINITY)
                std::cout << "inf\t";
            else
                std::cout << distance[i] << "\t";
        }
        std::cout << "\n";
    }

    // Find distance matrix
    int iter;
    /* double start_test_distance = omp_get_wtime(); */
    for (iter = 0; !check_compare; iter++) {
        check_compare = 1;

```

```

        // Update
#pragma omp parallel for schedule(dynamic)
        for (int from = 0; from < static_cast<int>(vertex_count); ++from) {
            if (distance[from] == PATH_INFINITY) continue;
            for (u_int to = 0; to < vertex_count; ++to) {
                if (graph_matrix[from][to] == 0) continue;
                if (distance[to] > distance[from] + graph_matrix[from][to]) {
                    check_compare = 0;
                    distance[to] = distance[from] + graph_matrix[from][to];
                    prev_vertex[to] = from;
                }
            }
        }
        // Write
        if (writing_work) {
            std::cout << "\tIteration " << (iter + 1) << ":\t";
            for (unsigned int i = 0; i < vertex_count; ++i) {
                if (distance[i] == PATH_INFINITY)
                    std::cout << "inf\t";
                else
                    std::cout << distance[i] << "\t";
            }
            std::cout << "\n";
        }
    }
    /* double finish_test_distance = omp_get_wtime();
    std::cout << "Test distance: " << finish_test_distance - start_test_distance <<
    std::endl; */

    // Find vectors_paths
    std::vector<u_int>* path = new std::vector<u_int>(vertex_count);
    /* double start_test_path = omp_get_wtime(); */
#pragma omp parallel for schedule(dynamic)
    for (int vertex = 0; vertex < static_cast<int>(vertex_count); ++vertex) {
        if (distance[vertex] == PATH_INFINITY) continue;
        u_int cur_vertex = vertex;
        do {
            path[vertex].push_back(cur_vertex);
            cur_vertex = prev_vertex[cur_vertex];
        } while (path[vertex].back() != start_vertex);
        reverse(path[vertex].begin(), path[vertex].end());
    }
    /* double finish_test_path = omp_get_wtime();
    std::cout << "Test path: " << finish_test_path - start_test_path << std::endl; */

    // Return
    if (writing_work) std::cout << "\n";
    res_format result = { distance, path };
    delete[] prev_vertex;
    return result;
}

/* PRINT RESULT */
void PrintResults(res_format result, u_int vertex_count, bool writing_work, bool
is_sequential) {
    if (is_sequential)
        std::cout << "<-----SEQUENTIAL RESULTS----->\n";
    else
        std::cout << "<-----PARALLEL RESULTS----->\n";

    // Write
    u_int* distance = result.distance;
    std::vector<u_int>* path = result.path;
    if (writing_work) {

```



```

        for (u_int vertex = 0; vertex < vertex_count; ++vertex) {
            std::cout << "\t" << vertex + 1 << "\t";
            if (distance[vertex] == PATH_INFINITY) {
                std::cout << "not path" << std::endl;
                continue;
            }
            std::cout << "Distance: " << distance[vertex] << std::endl;
            std::cout << "\t\tPath: ";
            std::cout << path[vertex][0] + 1;
            for (unsigned int i = 1; i < path[vertex].size(); ++i)
                std::cout << "->" << path[vertex][i] + 1;
            std::cout << "\n";
        }
    }
}

int main(int argc, char** argv) {
    // Graph params
    u_int graph_type = 0;
    u_int vertex_count = 0, edges_count = 0;
    u_int x_size = 0, y_size = 0;
    u_char** graph_matrix = nullptr;
    bool writing_work = 0;
    u_int start_vertex = 0;
    u_int threads_count = 0;

    // Enter graph params
    std::cout << "Enter one of type graph (\n";
    std::cout << "\t1. Case graph\n";
    std::cout << "\t2. Random graph\n";
    std::cout << "\t3. Lattice graph): ";
    std::cin >> graph_type;
    std::cout << "Enter count of threads: ";
    std::cin >> threads_count;
    switch (graph_type) {
        case 1: break;
        case 2:
            std::cout << "Enter count of vertexes: ";
            std::cin >> vertex_count;
            std::cout << "Enter count of edges: ";
            std::cin >> edges_count;
            break;
        case 3:
            std::cout << "Enter x-size: ";
            std::cin >> x_size;
            std::cout << "Enter y_size: ";
            std::cin >> y_size;
            break;
    }
    std::cout << "Enter start vertex: ";
    std::cin >> start_vertex;
    start_vertex--;
    std::cout << "Enter 1, if should print intermediate information, else enter any
other value: ";
    std::string s;
    std::cin >> s;
    if (s == "1") writing_work = true;

    // Generate matrix
    srand(static_cast<int>(time(0)));
    switch (graph_type) {
        case 1:
            graph_matrix = GetCaseMatrix(writing_work);
            vertex_count = 6;

```

```

        break;
    case 2:
        graph_matrix = GenerateMatrix(vertex_count, edges_count, writing_work);
        break;
    case 3:
        graph_matrix = GetLatticeMatrix(x_size, y_size, writing_work);
        vertex_count = x_size * y_size + 2;
        break;
    }
    omp_set_num_threads(threads_count);

    // Parallel realisation
    double par_start_time, par_finish_time;
    par_start_time = omp_get_wtime();
    res_format par_result = ParallelResult(graph_matrix, vertex_count, start_vertex,
writing_work);
    par_finish_time = omp_get_wtime();

    // Write parallel results
    PrintResults(par_result, vertex_count, writing_work, false);
    std::cout << "\tTIME: " << (par_finish_time - par_start_time) << "\n\n";

    // Free memory
    delete[] par_result.distance;
    for (u_int i = 0; i < par_result.path->size(); ++i)
        par_result.path[i].clear();
    delete[] par_result.path;

    // Sequential realisation
    double seq_start_time, seq_finish_time;
    seq_start_time = omp_get_wtime();
    res_format seq_result = SequentialResult(graph_matrix, vertex_count, start_vertex,
writing_work);
    seq_finish_time = omp_get_wtime();

    // Write sequential results
    PrintResults(seq_result, vertex_count, writing_work, true);
    std::cout << "\tTIME: " << (seq_finish_time - seq_start_time) << "\n\n";

    // Free memory
    delete[] seq_result.distance;
    for (u_int i = 0; i < seq_result.path->size(); ++i)
        seq_result.path[i].clear();
    delete[] seq_result.path;

    std::cout << "Acceleration: " << (seq_finish_time - seq_start_time) /
(par_finish_time - par_start_time) << "\n";

    system("pause");
    return 0;
}

```