Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №4 по курсу "Анализ алгоритмов"

Тема Многопоточное программирование

Студент Чепиго Д.С.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Содержание

Введение

Одной из задач программирования является ускорение решения вычислительных задач. Один из способов ее решения – использование параллельных вычислений.

В последовательном алгоритме решения какой-либо задачи есть операции, которые может выполнять только один процесс, например, операции ввода и вывода. Кроме того, в алгоритме могут быть операции, которые могут выполняться параллельно разными процессами. Способность центрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой, называют многопоточностью [1].

Процессом является программа в ходе своего выполнения. Каждый процесс состоит из одного или нескольких потоков - исполняемых сущностей, которые выполняют задачи, стоящие перед исполняемым приложением. После окончания выполнения всех потоков завершается процесс.

Современные процессоры могут выполнять две задачи на одном ядре при помощи дополнительного виртуального ядра. Такие процессоры называются многоядерными. Каждое ядро может выполнять только один поток за единицу времени. Если потоки выполняются последовательно, то их выполняет только одно ядро процессора, другие ядра не задействуются. Если независимые вычислительные задачи будут выполняться несколькими потоками параллельно, то будет задействовано несколько ядер процессора и решение задач ускорится.

Для распараллеливания может быть рассмотрена задача поиска кратчайших путей между всеми парами вершин графа. Данная задача решается при помощи алгоритма Флойда.

Цель лабораторной работы – получить навык организации параллельных вычислений на базе нативных потоков.

Задачи лабораторной работы:

- изучить основы теории графов;
- изучить последовательный и параллельный варианты алгоритма поиска кратчайших расстояний между всеми парами вершин графа;

- описать структуру разрабатываемого программного обеспечения;
- определить средства программной реализации выбранного алгоритма;
- реализовать разработанный алгоритм;
- провести сравнительный анализ по времени реализованного алгоритма;
- подготовить отчет о лабораторной работе.

1 Аналитическая часть

В данном разделе будут изучены основы теории графов, а также будет выбран способ представления графа. Кроме того, будет описан алгоритм Флойда поиска кратчайших расстояний между всеми парами вершин графа.

1.1 Основы теории графов

Многие объекты, возникающие в жизни человека, могут быть смоделированы (представлены в памяти компьютера) при помощи графов. Например, транспортные схемы (схема метрополитена и т. д.) изображают в виде станций, соединенных линиями. В терминах графов станции называются вершинами графа, а линии – ребра.

Графом [2] называется конечное множество вершин и множество ребер. Каждому ребру сопоставлены две вершины – концы ребра. Число вершин графа называют порядком. Путем на графе называется последовательность ребер, в которой конец одного ребра является началом следующего ребра. Начало первого ребра называется началом пути, конец последнего ребра - концом пути.

Бывают различные варианты определения графа. В данном определении концы у каждого ребра – равноправны. В этом случае нет разницы где начало, а где – конец у ребра. Но, например, в транспортных сетях бывают случаи одностороннего движения по ребру, тогда говорят об ориентированном графе – графе, у ребер которого одна вершина считается начальной, а другая – конечной.

Очень часто рассматриваются графы, в которых каждому ребру приписана некоторая числовая характеристика – вес. Вес может означать длину дороги или стоимость проезда по данному маршруту. Соответствующие графы называются взвешенными.

1.1.1 Способы представления графов

Представление графов в памяти – это способ хранения информации о ребрах графа, позволяющий решать следующие задачи:

- для двух данных вершин проверить, соединены ли вершины ребром;
- перебрать все ребра, исходящие из данной вершины.

Существует два способа представления графа. Рассмотрим их далее для следующего графа:

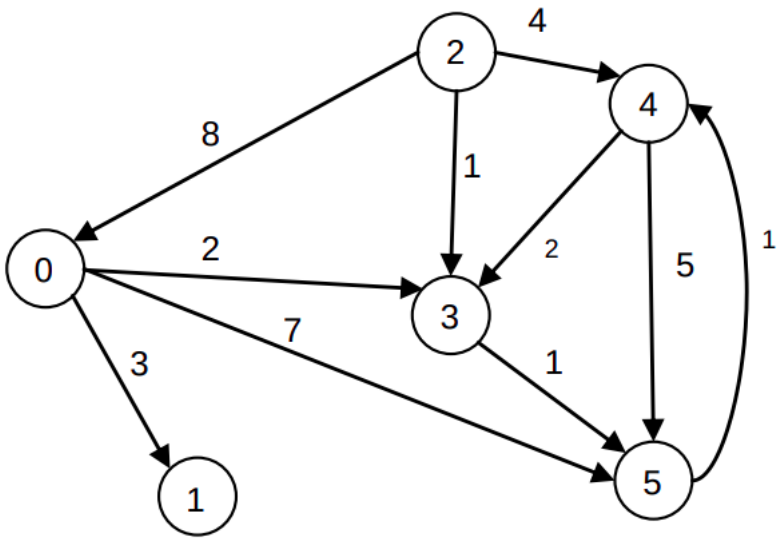

Рисунок 1.1 – Ориентированный взвешенный граф

Списки смежности

При представлении графа списками смежности для каждой вершины i хранится список W[i] смежных с ней вершин. Для рассмотренного примера списки будут такими:

$$W[1] = [2, 4, 6]$$
$$W[2] = []$$
$$W[3] = [1, 4, 5]$$
$$W[4] = [6]$$
$$W[5] = [4, 6]$$
$$W[6] = [5]$$

Рисунок 1.2 – Матрица смежности

Таким образом, весь граф можно представить одним списком, состоящим из вложенных списков смежности вершин.

При представлении взвешенного графа списками смежности можно поступить двумя способами. Можно в списках смежности хранить пару (кортеж) из двух элементов – номер конечной вершины и вес ребра. Но в этом случае неудобно проверять наличие ребра между двумя вершинами.

Другой способ – хранить списки смежности как ранее, а веса ребер хранить в отдельном ассоциативном массиве, в котором ключом будет пара из двух номеров вершин (номер начальной и конечной вершины), а значением будет вес ребра между этими вершинами.

Матрица смежности

При представлении взвешенного графа матрицей смежности информация о ребрах графа хранится в квадратной матрице (двумерном списке), где элемент A[i][j] равен весу ребра, если ребра i и j соединены ребром. Иначе равен определенному символу (для рассматриваемого графа 0):

$$A = \begin{pmatrix} 0 & 3 & 0 & 2 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 5 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.1}$$

Если граф не является взвешенным, то элемент матрицы смежности равен 1, если ребра соединены.

Матрица смежности требует $O(n^2)$ памяти и может оказаться неэффективным способом хранения дерева или разреженных графов. Но использование матрицы смежности позволяет применять при реализации вычислительных процедур анализа графов матричные алгоритмы обработки данных, которые можно распараллелить. Поэтому при реализации данной лабораторной работы граф будет представляться при помощи матрицы смежности.

1.2 Алгоритм Флойда

Алгоритм Флойда [3] позволяет найти кратчайшее расстояние между любыми двумя вершинами в графе.

1.2.1 Последовательный алгоритм

Будем считать, что в графе n вершин, пронумерованных числами от 0 до $n-1$. Граф задан матрицей смежности, вес ребра $i-j$ равен w_{ij}. При отсутствии ребра $i-j$ значение $w_{ij}=-1$, также будем считать, что $w_{ii}=0$.

Пусть значение a_{ij}^k равно длине кратчайшего пути из вершины i в вершину j, при этом путь может заходить в промежуточные вершины только с номерами меньшими k (не считая начала и конца пути). То есть a_{ij}^0 - это длина кратчайшего пути из i в j, который вообще не содержит промежуточных вершин, то есть состоит только из одного ребра $i-j$, поэтому $a_{ij}^0=w_{ij}$. Значение $a_{ij}^1=w_{ij}$ равно длине кратчайшего пути, который может проходить через промежуточную вершину с номером 0, путь с весом a_{ij}^2 может проходить через промежуточные вершины с номерами 0 и 1 и т. д. Путь с весом a_{ij}^n может проходить через любые промежуточные вершины, поэтому значение a_{ij}^n равно длине кратчайшего пути из i в j.

Алгоритм Флойда последовательно вычисляет a_{ij}^0, a_{ij}^1, a_{ij}^2, $\ldots$, a_{ij}^n, увеличивая значение параметра k. Начальное значение - $a_{ij}^0=w_{ij}$.

Теперь предполагая, что известны значения a_{ij}^{k-1} вычислим a_{ij}^k. Кратчайший путь из вершины i в вершину j, проходящий через вершины с номерами,

меньшими, чем k может либо содержать, либо не содержать вершину с номером $k-1$. Если он не содержит вершину с номером $k-1$, то вес этого пути совпадает с a_{ij}^{k-1}. Если же он содержит вершину $k-1$, то этот путь разбивается на две части: $i-(k-1)$ и $(k-1)-j$. Каждая из этих частей содержит промежуточные вершины только с номерами, меньшими $k-1$, поэтому вес такого пути равен $a_{i,k-1}^{k-1}+a_{k-1,i}^{k-1}$. Из двух рассматриваемых вариантов необходимо выбрать вариант наименьшей стоимости, поэтому:

$$a_{ij}^{k}=min(a_{ij}^{k-1},a_{i,k-1}^{k-1}+a_{k-1,i}^{k-1}) \tag{1.2}$$

1.2.2 Параллельный алгоритм

Более эффективный способ алгоритма может состоять в одновременном выполнении нескольких операций обновления значений матрицы смежности A. Параллельный алгоритм Флойда заключается в том, что на k-й итерации мы отсылаем k-ю строку всем процессам, а затем каждый процесс выполняет последовательный алгоритм Флойда для полосы матрицы смежности, одновременно обновляя значение матрицы.

2 Конструкторская часть

2.1 Разработка алгоритмов

На рисунке 2.2 представлена схема последовательного алгоритма Флойда, на рисунке 2.3 - параллельного. Схема алгоритма нахождения крайтчайшего расстояния между двумя вершинами показана на рисунке 2.1. Схема организации многопоточности представлена на рисунке 2.4.

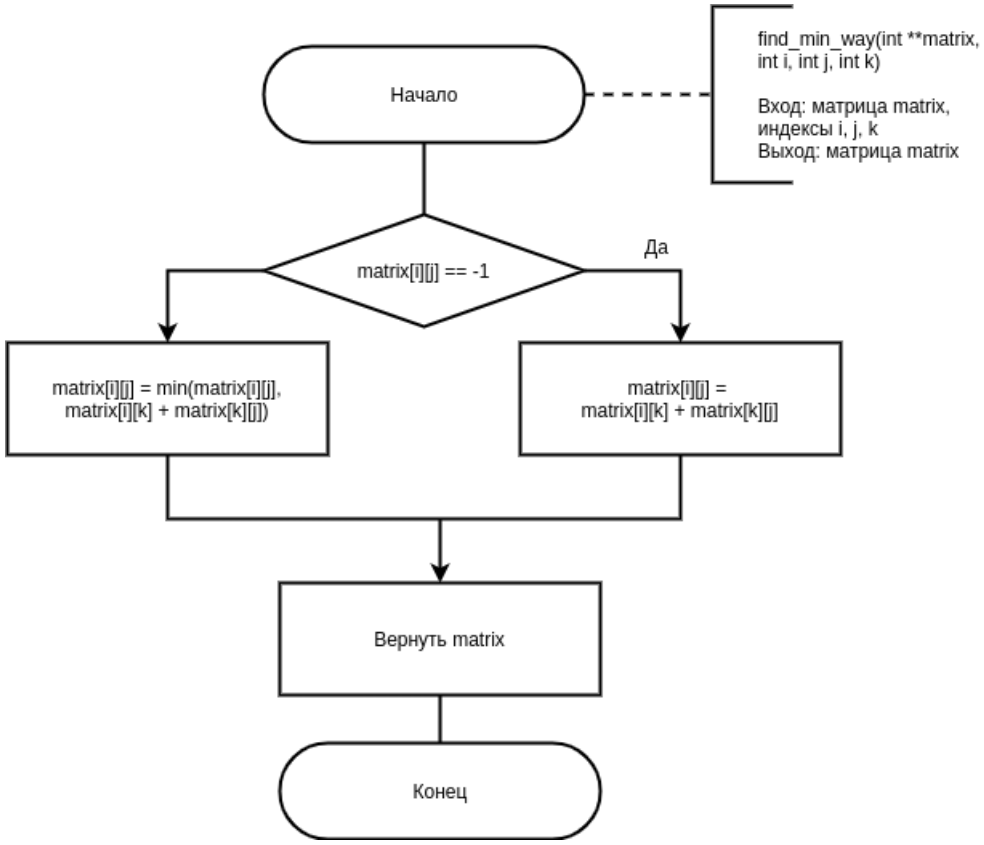

Рисунок 2.1 – Алгоритм нахождения кратчайшего пути между двумя вершинами

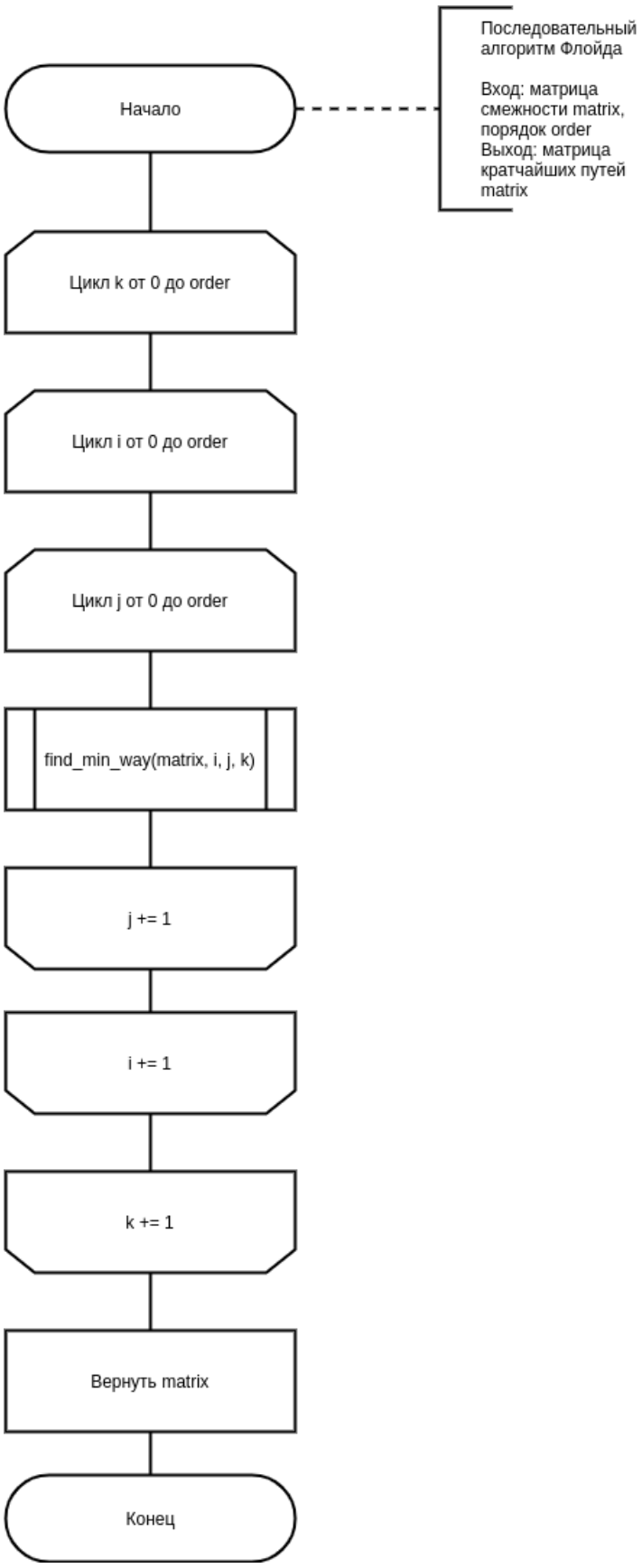

Рисунок 2.2 – Последовательный алгоритм Флойда

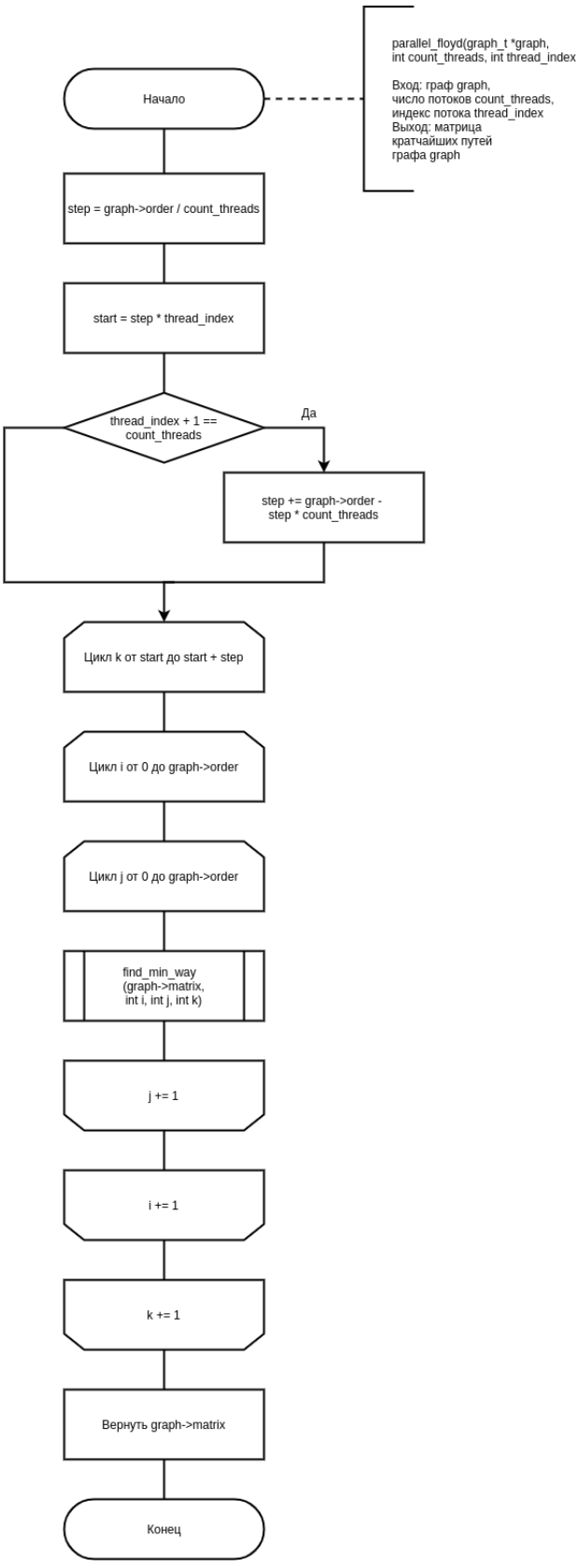

Рисунок 2.3 – Параллельный алгоритм Флойда

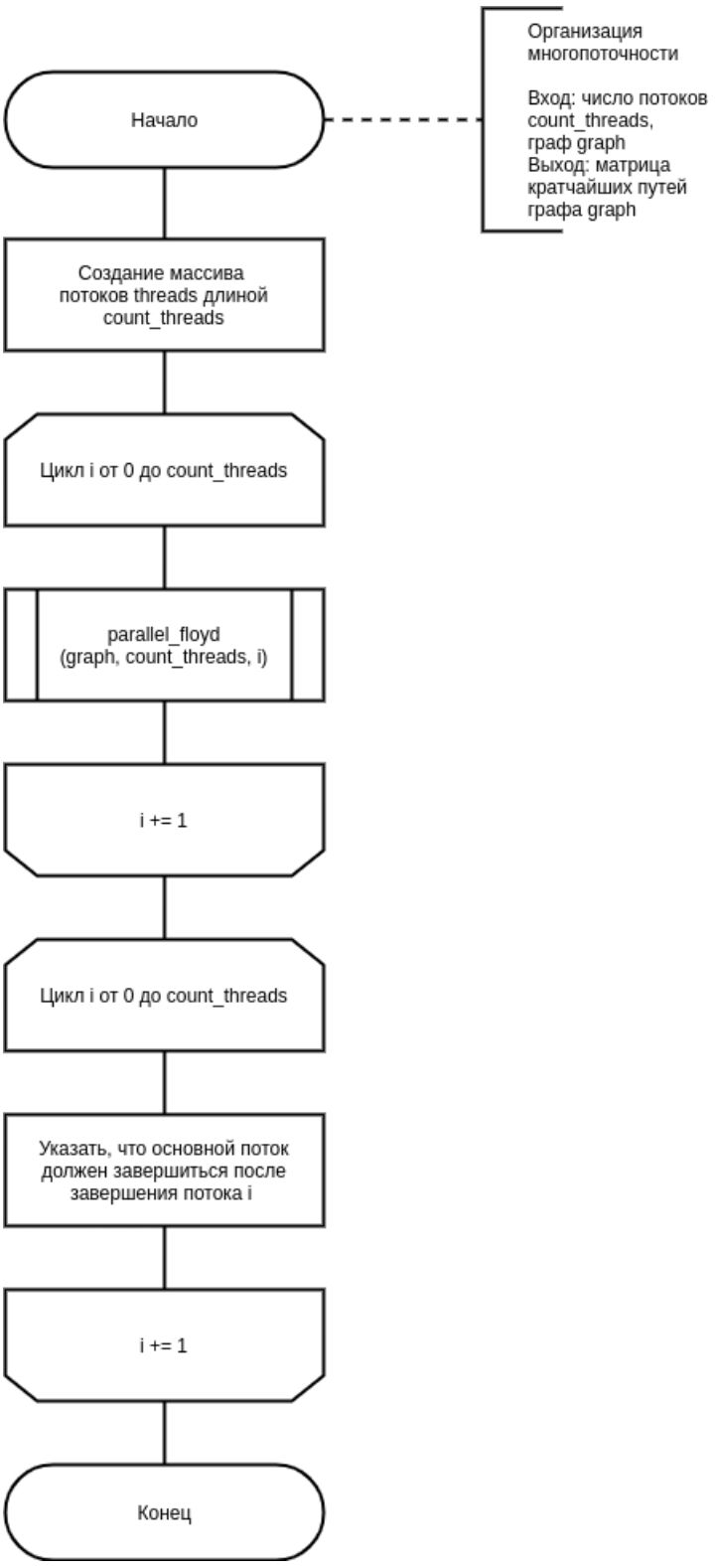

Рисунок 2.4 – Организация многопоточности

3 Технологическая часть

3.1 Требования к ПО

К программе предъявляется ряд требований:

- входными данными являются количество вершин графа и матрица смежности;
- элементы матрицы смежности – натуральные числа и число 0;
- на выходе – результирующая матрица с кратчайшими расстояниями между всеми вершинами.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран C++ – компилируемый, статически типизированный язык программирования общего назначения [4].

Данный выбор обусловлен поддержкой языком парадигмы объектно – ориентированного программирования, возможностью создавать нативные потоки и наличием методов для замера процессорного времени.

Время работы реализованных алгоритмов было замерено с помощью библиотеки chrono [5].

3.3 Реализация алгоритмов

В листинге 3.1 приведена реализация последовательного алгоритма Флойда.

Листинг 3.1 – Реализация последовательного алгоритма Флойда

```
1 void Graph::simpleFWA()
2 {
3     for (int i = 0; i < size; i++)
4     {
5         for (int j = 0; j < size; j++)
```

```
        {
            for (int k = 0; k < size; k++)
            {
                if (i != j && matrix[i][k] != 0 && matrix[k][j] != 0)
                {
                    if (matrix[i][j] == 0)
                        matrix[i][j] = matrix[i][k] + matrix[k][j];
                    else
                        matrix[i][j] = min(matrix[i][j], matrix[i][k] +
                            matrix[k][j]);
                }
            }
        }
    }
}
```

В листинге 3.2 приведена реализация параллельного алгоритма Флойда.

Листинг 3.2 – Реализация параллельного алгоритма Флойда

```

void Graph::parallelFWA(int index)
{
    int step = size / countThreads;
    int start = index * step;

    if (index + 1 == countThreads)
    {
        step += + (size - step * countThreads);
    }

    for (int k = start; k < start + step; k++)
    {
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (i != j && matrix[i][k] != 0 && matrix[k][j] != 0)
                {
                    if (matrix[i][j] == 0)
                        matrix[i][j] = matrix[i][k] + matrix[k][j];
                    else
                        matrix[i][j] = min(matrix[i][j], matrix[i][k] +
                            matrix[k][j]);
                }
            }
        }
    }
}
```

```
29
30 void Graph::doParallel()
31 {
32     std::vector<std::thread> threads(countThreads);
33
34     for (int i = 0; i < countThreads; i++)
35     {
36         threads[i] = std::thread(&Graph::parallelFWA, this, i);
37     }
38
39     for (int i = 0; i < countThreads; i++)
40     {
41         threads[i].join();
42     }
43 }
```

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся замерный эксперимент:

- операционная система Ubuntu 22.04.1 LTS Linux x86_64 [6];
- память 8 ГБ;
- процессор Intel® Core™ i3-7130U.

Замеры проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно замерным экспериментом.

4.2 Пример работы программы

На рисунке 4.2 представлен пример работы программы. Вводится количество вершин в графе и матрица смежности. Далее выводится матрица кратчайших путей и время выполнения последовательного и параллельного алгоритма в микросекундах.

```
dashori@fossa ~/P/A/b/l/code (main)> ./a.out
Введите количество вершин графа: 4
Введите 1 строчку матрицы смежности:
0 8 0 1
Введите 2 строчку матрицы смежности:
0 0 1 0
Введите 3 строчку матрицы смежности:
4 0 0 0
Введите 4 строчку матрицы смежности:
0 2 9 0
Матрица смежности:
0 8 0 1
0 0 1 0
4 0 0 0
0 2 9 0

Последовательный алгоритм:
Матрица кратчайших путей:
0 3 4 1
5 0 1 6
4 7 0 5
7 2 3 0
Время: 11 (микросекунды)

Параллельный алгоритм:
Введите количество потоков: 4
Матрица кратчайших путей:
0 3 4 1
5 0 1 6
4 7 0 5
7 2 3 0
Время: 876 (микросекунды)
```

Рисунок 4.1 – Пример работы программы

Также при компиляции программы можно указать вывод графа на экран, но для графов с количеством вершин более 10 такой граф не является наглядным.

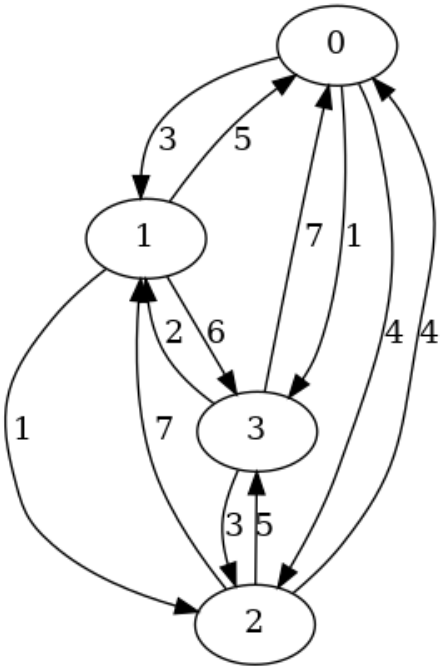

Рисунок 4.2 – Пример графа из тестовой программы

4.3 Время выполнения реализованных алгоритмов

Было проведено два эксперимента по замеру времени. Первый – зависимость времени работы реализованных алгоритмов от количества потоков, при это количество вершин графа фиксированно и равно 150. Второй – зависимость времени от количества вершин графа, при этом для параллельной реализации алгоритма количество потоков фиксировано и равно 4.

Замеры времени работы реализованных алгоритмов для каждого эксперимента проводились 200 раз. В таблице 4.1 представлен результат зависимости времени работы реализованных алгоритмов от количества вершин в графе, количество потоков для параллельной реализации равно 4.

Таблица 4.1 – Результаты замеров времени реализованных алгоритмов в микросекундах

Количество вершин в графе	Последовательный алгоритм	Параллельный алгоритм
1	0	428
11	99	162
21	548	617
31	1467	963
41	3401	1805
50	5831	3273
100	42865	20937
150	138661	67138
200	331876	156506

В таблице 4.2 представлен результат зависимости времени работы реализованных алгоритмов от количества потоков, количество вершин графа равно 150.

Таблица 4.2 – Результаты замеров времени реализованных алгоритмов в микросекундах

Количество потоков	Последовательный алгоритм	Параллельный алгоритм
1	137489	132363
2	274394	203034
3	411187	284118
4	546128	349896
5	681251	420196
6	815748	488302
7	950798	555731
8	1094473	625521
9	1230592	693366
10	1367051	760079
11	1502139	826971
12	1637389	893409
13	1772377	960084
14	1907026	1028013
15	2042003	1093619
16	2176615	1160360

На рисунке 4.3 представлена зависимость времени работы реализованных алгоритмов Флойда от количества вершин в графе. На рисунке 4.4 представлена зависимость времени работы реализованных алгоритмов Флойда от количества потоков.

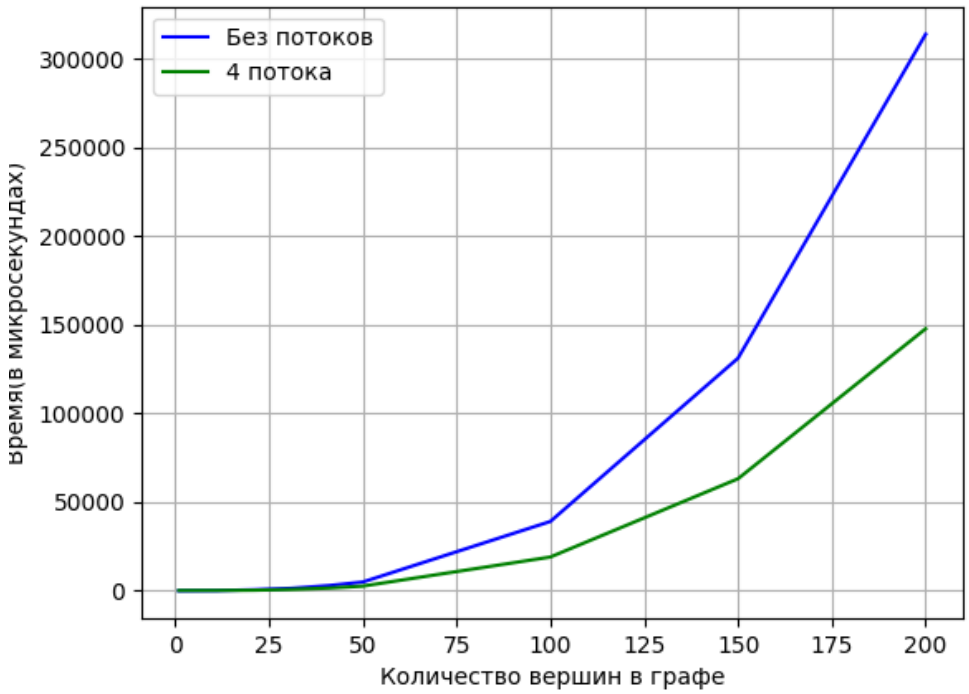

Рисунок 4.3 – Зависимость времени работы реализованных алгоритмов Флойда от количества вершин в графе

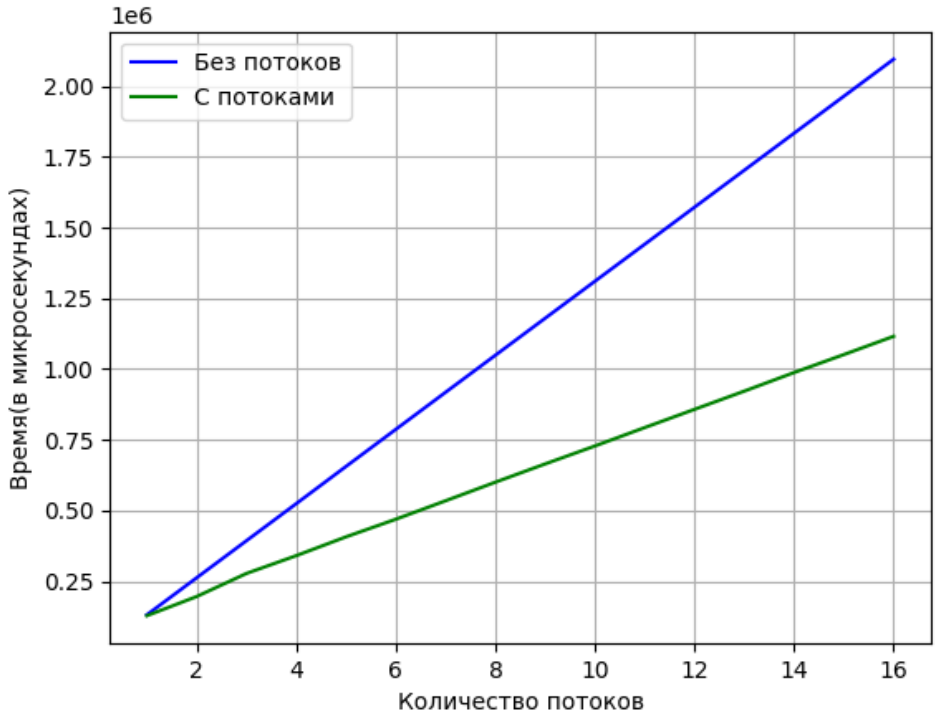

Рисунок 4.4 – Зависимость времени работы реализованных алгоритмов сортировки от длины массива, где длина – степень числа два

Из рисунка 4.3 видно особенность битонной сортировки, которая сортирует только массивы, длина которых является степенью числа два. Все массивы, имеющую иную длину, дополняются фиктивными элементами до ближайшей степени числа два, что видно на рисунке 4.3.

Самая быстрая сортировка из исследуемых – сортировка подсчетом. На размерах массива до 64 она превосходит в 8 раз битонную сортировку и в 3 раза сортировку слиянием. На размерах от 100 до 600 она работает быстрее в 30-50 чем битонная и в 6 раз быстрее чем сортировка слиянием. На максимальном тестируемом размере – 65536 сортировка подсчетом быстрее битонной сортировки в 60 раз, а сортировки слиянием в 7 раз.

Быстрая скорость сортировки подсчетом объясняется тем фактом, что она проигрывает по памяти двум другим алгоритмам сортировки, так как хранит дополнительный массив длина которого – диапазон значений. Так как значения элементов массива генерировались произвольно в диапазоне (-INT_MAX, INT_MAX) [**?**], исключается вариант, что сортировка подсчетом работает быстрее из-за тестовых данных. Также алгоритм битонной сортировки был реализован не параллельно, что ухудшает его трудоемкость.

Список использованных источников

[1] Многопоточность [Электронный ресурс]. Режим доступа: https://ru.coursera.org/lecture/ios-multithreading/chto-takoie-mnoghopotochnost-4MMgN (дата обращения: 08.11.2021).

[2] Теория графов [Электронный ресурс]. Режим доступа: https://foxford.ru/wiki/informatika/teoriya-grafov (дата обращения: 08.11.2021).

[3] Алгоритм Флойда [Электронный ресурс]. Режим доступа: https://foxford.ru/wiki/informatika/algoritm-floyda (дата обращения: 08.11.2021).

[4] Язык программирования C++ [Электронный ресурс]. Режим доступа: https://isocpp.org/ (дата обращения: 11.11.2022).

[5] Стандарт языка C++ [Электронный ресурс]. Режим доступа: https://isocpp.org/files/papers/N4860.pdf (дата обращения: 11.11.2022).

[6] Операционная система Ubuntu 22.04 LTS [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/jammy/ (дата обращения: 11.11.2022).