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 дисциплине "Анализ алгоритмов"

Тема Многопоточное программирование

Студент Хамзина Р. Р.

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2021 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.

Целью данной лабораторной работы является изучение многопоточности на основе алгоритма Флойда поиска кратчайших расстояний между всеми парами вершин графа. Для достижения поставленной цели требуется выполнить следующие задачи:

- изучить основы теории графов;
- выбрать способ представления графа;

- изучить последовательный и параллельный варианты алгоритма поиска кратчайших расстояний между всеми парами вершин графа;
- привести схемы изучаемого алгоритма;
- описать используемые типы и структуры данных;
- описать структуру разрабатываемого программного обеспечения;
- определить средства программной реализации выбранного алгоритма;
- реализовать разработанный алгоритм;
- провести функциональное тестирование программного обеспечения;
- провести сравнительный анализ по времени реализованного алгоритма;
- подготовить отчет о выполненной 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:

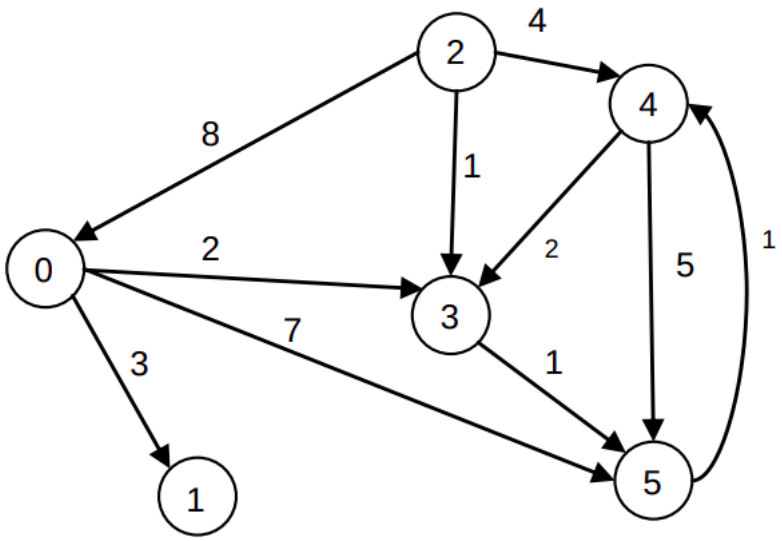

Рисунок 1.1 – Ориентированный взвешенный граф

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:

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

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 -1):

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

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 вы-

числительных 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.

1.3 Вывод

Были изучены основы теории графов и способы представления графа. Граф будет представляться при помощи матрицы смежности. Также были рассмотрены последовательный и параллельный варианты алгоритма Флойда поиска кратчайших путей между всеми парами вершин графа.

Программе, реализующей данный алгоритм, на вход будет подаваться матрица смежности, которая задает граф. Выходными данными такой программы должна быть матрица смежности кратчайших путей между всеми парами вершин графа. Программа должна работать в рамках следующих ограничений:

- веса ребер графа - целые неотрицательные числа;
- осутствие пути между вершинами обозначается -1;
- порядок графа должен быть больше, либо равен 2;
- должно быть выдано сообщение об ошибке при вводе пустой матрицы смежности или при вводе недопустимого веса ребра графа.

Пользователь должен иметь возможность выбора построения алгоритма с распараллеливанием и без него. В случае параллельного алгоритма пользователь должен иметь возможность ввода числа потоков. Также должны быть реализованы сравнение алгоритмов по времени работы в зависимости от числа потоков и в зависимости от порядка графа с выводом результатов на экран и получение графического представления результатов сравнения. Результат данных действий пользователь должен получать при помощи меню.

2 Конструкторская часть

В данном разделе будут представлены схемы последовательного и параллельного вариантов алгоритма Флойда. Будут описаны типы и структуры данных, используемые для реализации, а также структура разрабатываемого программного обеспечения. Кроме того, будут выделены классы эквивалентности для тестирования.

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.

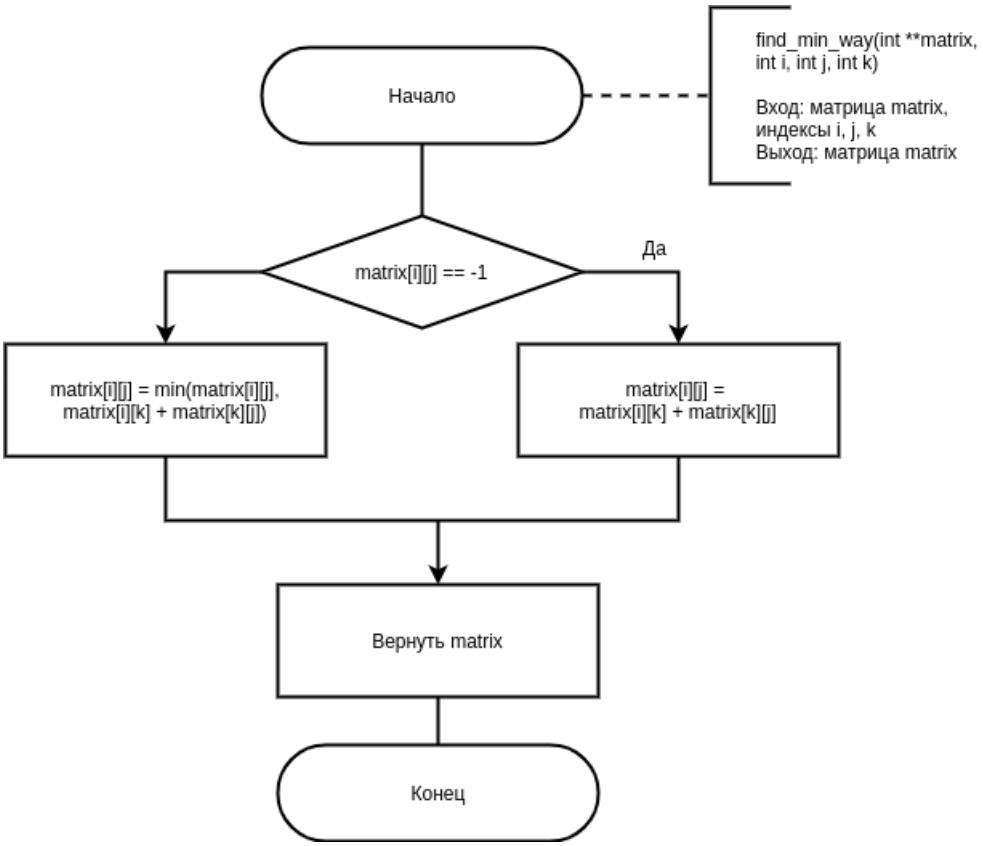

Рисунок 2.1 – Алгоритм нахождения кратчайшего пути между двумя вершинами

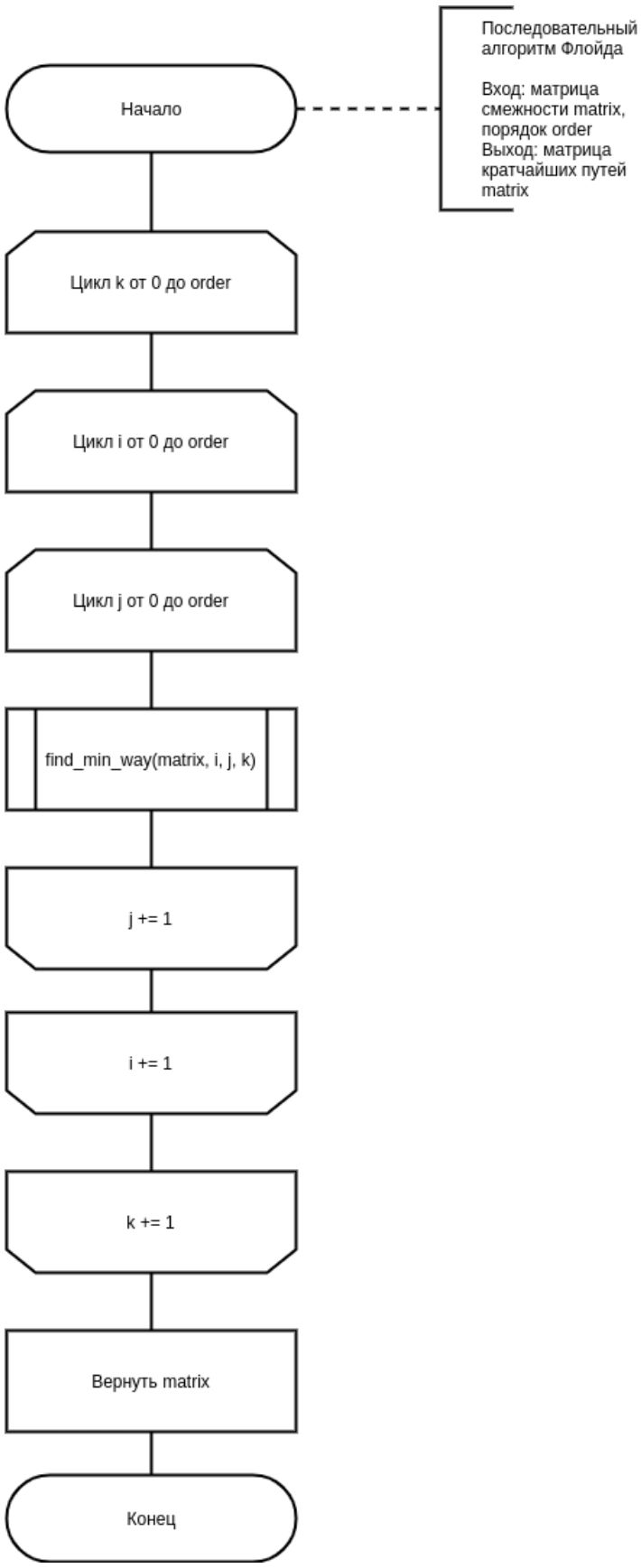

Рисунок 2.2 – Последовательный алгоритм Флойда

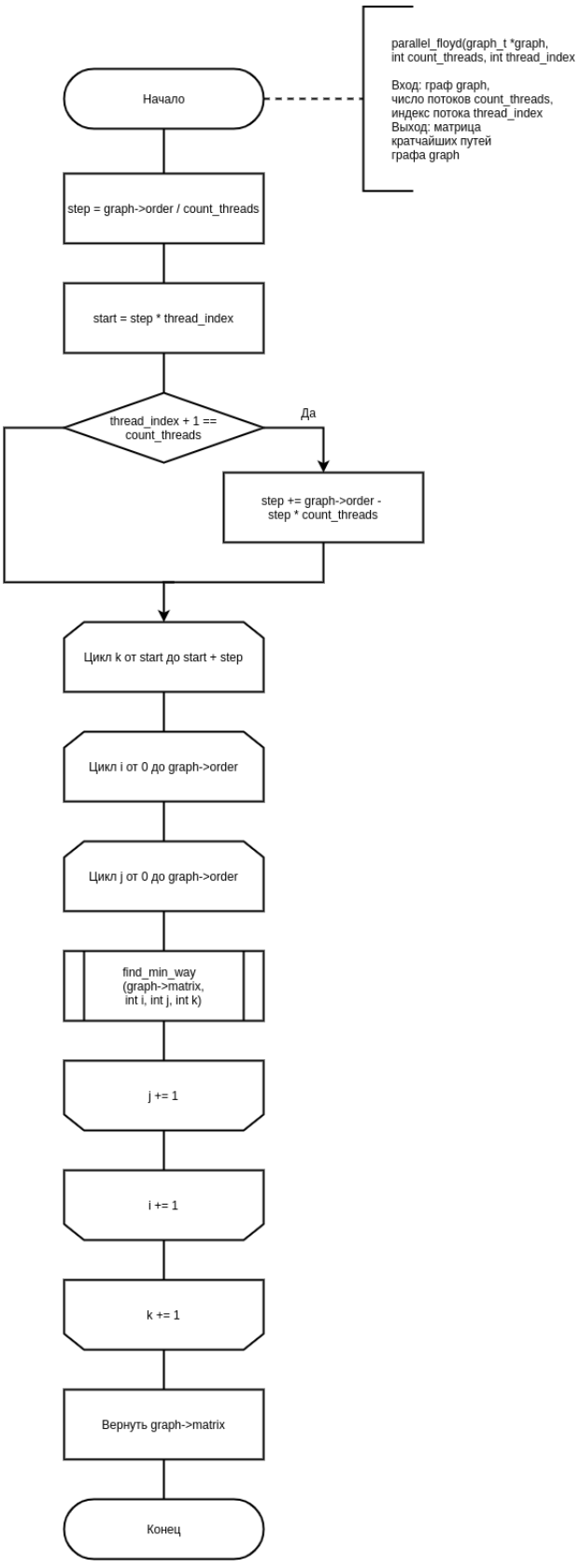

Рисунок 2.3 – Параллельный алгоритм Флойда

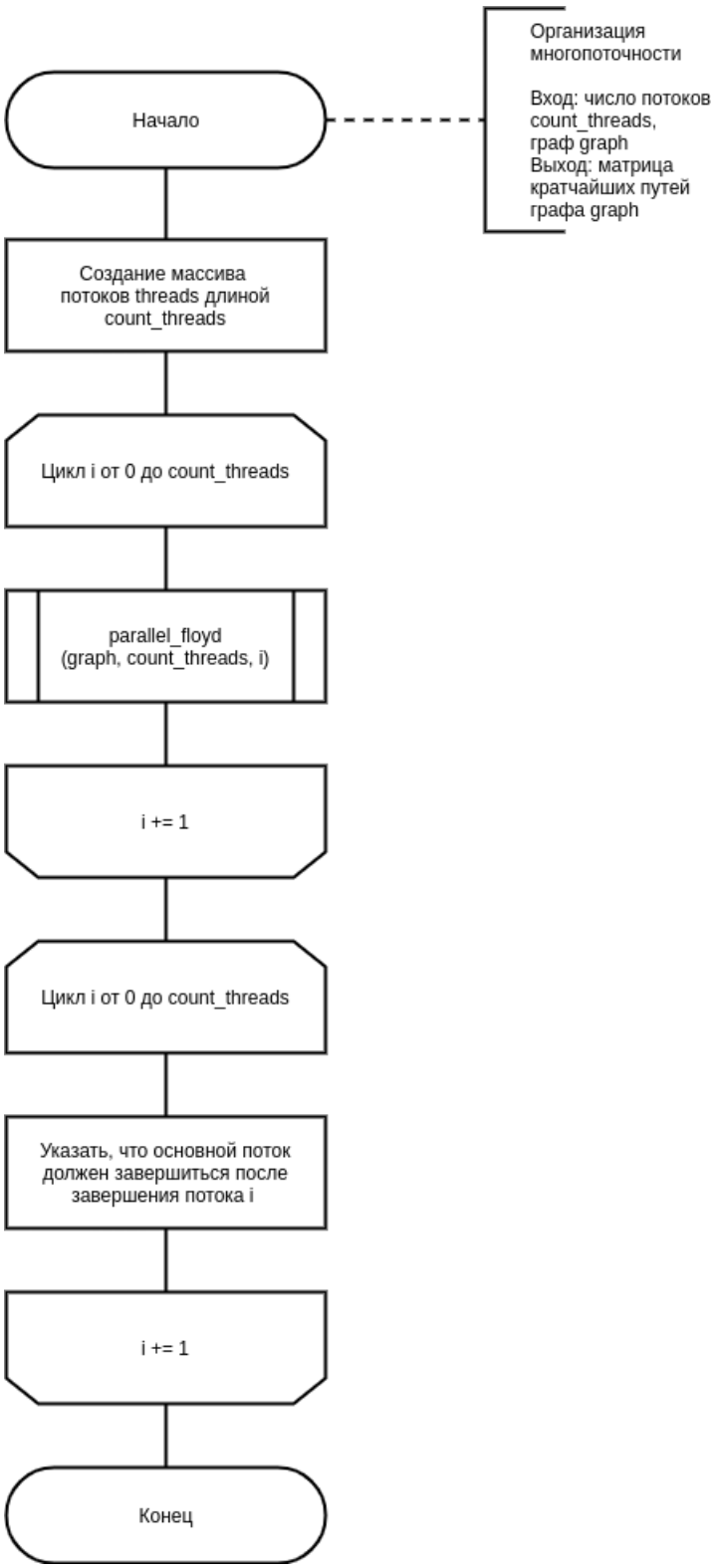

Рисунок 2.4 – Организация многопоточности

2.2 Описание используемых типов и структур данных

Для реализации многопоточности будет использован тип данных int - для числа потоков.

Для представления графа будет использована структура данных $graph_t$, которая имеет следующие поля:

- $order$ - число типа int, которое задает порядок графа;
- $matrix$ типа $int**$ - массив указателей на указатели на целые числа, который задает матрицу смежности.

2.3 Структура разрабатываемого ПО

При реализации разрабатываемого программного обеспечения будет использоваться метод структурного программирования. Для взаимодействия с пользователем в функции $intmain(void)$ будет реализовано меню, при помощи которого будут вызываться последовательный и параллельный методы нахождения матрицы кратчайших путей в графе и функции сравнительного анализа. Для работы с графом будут разработаны следущие функции, входным параметром которых является указатель на структуру графа:

- процедура инициализации графа;
- создание графа со случайными весами ребер, выходным параметром функции является сгенерированная матрица смежности;
- ввод графа, выходным параметром является введенная матрица смежности;
- процедура вывода графа;
- процедура освобождения памяти из-под матрицы смежности;

- функции поиска кратчайших путей между двумя любыми вершинами в графе для каждого алгоритма(последовательного и параллельного), у которых на выходе - матрица кратчайших расстояний между любыми двумя вершинами в графе.

Для сравнительного анализа будут реализованы:

- функции замеров времени, входными параметрами которых являются граф и число потоков, выходным парметром - массив временных значений;
- функция графического представления замеров времени, у которой на входе - массив временных значений, на выходе - его графическое представление.

2.4 Классы эквивалентности при тестировании

Для тестирования разрабатываемой программы будут выделены следующие классы эквивалентности:

- некорректный порядок графа (меньше 2);
- некорректный вес графа (не число или число, меньшее -1);
- некорректный ввод числа потоков (меньше 1);
- корректный ввод всех параметров.

2.5 Вывод

Были представлены схемы поиска кратчайших расстояний между двумя любыми вершинами в графе. Были указаны типы и структуры данных, используемые для реализации, и описана структура разрабатываемого программного обеспечения. Также были выделены классы эквивалентности для тестирования ПО.

3 Технологическая часть

В данном разделе будут указаны средства реализации, сведения о модулях программы, будут представлены листинги кода, а также функциональные тесты.

3.1 Средства реализации

Реализация данной лабораторной работы выполнялась при помощи языка программирования $C++$ [4]. Выбор ЯП обусловлен наличием инструментов для реализации принципов многопоточного программирования.

Замеры времени проводились при помощи функции $std :: chrono :: system\_clock :: now(...)$ из библиотеки *chrono* [5].

Реализация графического представления замеров времени производилась при помощи языка программирования *Python* [6], так как данный ЯП представляет простую в использовании графическую библиотеку.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- main.cpp - главный файл программы, предоставляющий пользователю меню для выполнения основных функций;
- graph.cpp и graph_t.h - файлы, содержащие функции работы с графом;
- time_test.cpp и time_test.h - файлы, содержащие функции замеров времени работы указанных алгоритмов;
- constants.h - файл, содержащий все необходимые константы;
- graph_result.py - файл, содержащий функции визуализации временных характеристик описанных алгоритмов.

3.3 Листинги кода

В листинге 3.1 представлен алгоритм нахождения кратчайшего пути между двумя вершинами. В листингах 3.2-3.3 представлены последовательный и параллельный алгоритмы Флойда.

Листинг 3.1 – Алгоритм нахождения кратчайшего пути между двумя вершинами

```
1  void find_min_way(int **matrix, int i, int j, int k)
2  {
3      if (i != j && matrix[i][k] != NO_WAY && matrix[k][j] !=
           NO_WAY)
4      {
5          if (matrix[i][j] == NO_WAY)
6              matrix[i][j] = matrix[i][k] + matrix[k][j];
7          else
8              matrix[i][j] = min(matrix[i][j], matrix[i][k] +
                   matrix[k][j]);
9      }
10 }
```

Листинг 3.2 – Последовательный алгоритм Флойда

```
1  void floyd(graph_t* graph)
2  {
3      for (int k = 0; k < graph->order; k++)
4      {
5          for (int i = 0; i < graph->order; i++)
6          {
7              for (int j = 0; j < graph->order; j++)
8              {
9                  find_min_way(graph->matrix, i, j, k);
10             }
11         }
12     }
13 }
```

Листинг 3.3 – Параллельный алгоритм Флойда

```
1  void parallel_floyd(graph_t* graph, int count_threads, int
       thread_index)
2  {
3      int step = graph->order / count_threads;
4      int start = thread_index * step;
5
6      if (thread_index + 1 == count_threads)
7      {
8          step += (graph->order - step * count_threads);
9      }
10
11     for (int k = start; k < start + step; k++)
12     {
13         for (int i = 0; i < graph->order; i++)
14         {
15             for (int j = 0; j < graph->order; j++)
16             {
17                 find_min_way(graph->matrix, i, j, k);
18             }
19         }
20     }
21 }
22
23 void multithreading(int count_threads, graph_t* graph)
24 {
25     std::vector<std::thread> threads(count_threads);
26
27     for (int i = 0; i < count_threads; i++)
28     {
29         threads[i] = std::thread(parallel_floyd, std::ref(graph),
               count_threads, i);
30     }
31
32     for (int i = 0; i < count_threads; i++)
33     {
34         threads[i].join();
35     }
36 }
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для функций, реализующих алгоритм Флойда. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Матрица A	Кол-во потоков	Ожидаемый результат
$\begin{pmatrix} \end{pmatrix}$	4	Сообщение об ошибке
$\begin{pmatrix} 0 & 1 & -1 \\ -2 & 0 & 3 \\ 4 & 5 & 0 \end{pmatrix}$	4	Сообщение об ошибке
$\begin{pmatrix} 0 & 1 & -1 \\ -2 & 0 & 3 \\ 4 & 5 & 0 \end{pmatrix}$	0	Сообщение об ошибке
$\begin{pmatrix} 0 & 5 & 9 & -1 \\ -1 & 0 & 2 & 8 \\ -1 & -1 & 0 & 7 \\ 4 & -1 & -1 & 0 \end{pmatrix}$	4	$\begin{pmatrix} 0 & 5 & 7 & 13 \\ 12 & 0 & 2 & 8 \\ 11 & 16 & 0 & 7 \\ 4 & 9 & 11 & 0 \end{pmatrix}$

3.5 Вывод

Были реализованы функции последовательного и параллельного алгоритма Флойда. Было проведено функциональное тестирование указанных функций.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, и будет проведен сравнительный анализ реализованных алгоритмов по количеству потоков и по порядку графа.

4.1 Технические характеристики

Тестирование проводилось на устройстве со следующими техническими характеристиками:

- операционная система: Ubuntu 20.04.1 Linux x86_64 [7];
- память : 8 GiB;
- процессор: AMD® Ryzen™ 3 3200u @ 2.6 GHz;
- 4 физических ядра, 4 логических ядра [8].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 приведен пример работы программы.

```
МЕНЮ:
1. Поиск без распараллеливания
2. Многопоточный поиск
3. Замер времени
0. Выход
Выбор: 2

Введите порядок графа: 4

Введите матрицу смежности графа:
0
5
9
-1
-1
0
2
8
-1
-1
0
7
4
-1
-1
0

Матрица смежности графа:
0 5 9 -1
-1 0 2 8
-1 -1 0 7
4 -1 -1 0

Введите число потоков:
4

Матрица кратчайших путей графа:
0 5 7 13
12 0 2 8
11 16 0 7
4 9 11 0

Граф удален.
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Функция $std :: chrono :: system\_clock :: now(...)$ из библиотеки $chrono$ ЯП $C++$ возвращает процессорное время в секундах - значение типа float.

Для замера времени:

- получить значение времени до начала выполнения алгоритма, затем после её окончания. Чтобы получить результат, необходимо вычесть из второго значения первое;
- первый шаг необходимо повторить iters раз (в программе iters равно 100), суммируя полученные значения, а затем усреднить результат.

Сравнительный анализ по времени в зависимости от количества потоков проводился для матриц смежности, заполненных случайным образом, порядком 150. Результаты измерения времени приведены в таблице 4.1 (в с).

Таблица 4.1 – Результаты замеров времени в зависимости от числа потоков

Кол-во потоков	С распараллеливанием	Без распараллеливания
1	0.052929	0.055785
2	0.038668	
4	0.033114	
8	0.037336	
16	0.035919	
24	0.037970	
32	0.036334	

На рисунке 4.2 приведены графические результаты сравнения временных характеристик для различного числа потоков.

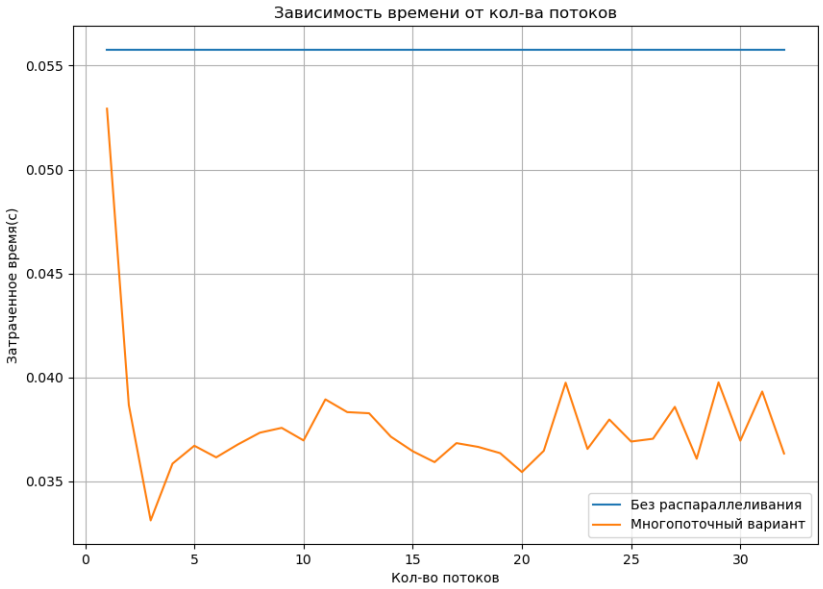

Рисунок 4.2 – Сравнение по времени в зависимости от количества потоков

Кроме того, замеры времени проводились для матриц смежности, заполненных случайным образом, порядком от 100 до 200. Результаты измерения времени работы алгоритмов в зависимости от порядка графа приведены в таблице 4.2 (в с).

Таблица 4.2 – Результаты замеров времени в зависимости от порядка графа

Порядок	4 потока	Без распараллеливания
100	0.010940	0.015457
110	0.015257	0.019669
120	0.017407	0.025236
130	0.022600	0.033559
140	0.026211	0.042147
150	0.032678	0.058643
160	0.040683	0.068319
170	0.049890	0.086102
180	0.060872	0.096152
190	0.070856	0.110270
200	0.083142	0.130470

На рисунке 4.3 приведены графические результаты сравнения временных характеристик для различного порядка графа.

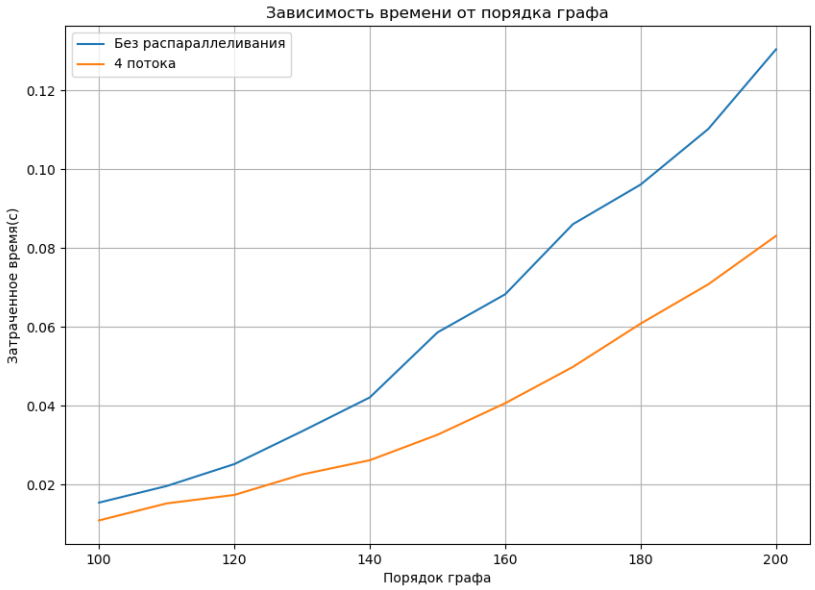

Рисунок 4.3 – Сравнение по времени в зависимости от порядка графа

4.4 Вывод

В результате эксперимента было получено, что выполнение алгоритма Флойда при помощи 4 потоков быстрее его выполнения при отсутсвии многопоточности в 1.7 раза для графа порядка 150. Лучшие результаты применения многопоточности были получены при 4 потоках, так как это максимальное число потоков, допустимое на устройстве, которое было использовано для тестирования. Тогда, для указанных данных небходимо использовать параллельную реализацию алгоритма поиска кратчайших расстояний между любыми двумя вершинами.

Также в результате сравнительного анализа было установлено, что при увеличении порядка графа параллельная реализация алгоритма быстрее последовательной: при порядке, равном 120 - в 1.4 раза, при порядке, равном 200 - в 1.6 . Можно сделать вывод, что алгоритм Флойда следует параллелить при больших значениях порядка графа.

Заключение

В результате исследования было получено, что на устройстве, использованном для тестирования, распараллеливание алгоритма Флойда показывает лучшие результаты при работе 4 потоков - быстрее последовательного алгоритма в 1.7 раза для порядка графа, равного 150. При этом многопоточность следует использовать на больших значениях порядка графа, так как параллельный алгоритм на порядке, равном 100, работает быстрее последовательного в 1.4 раза, а на порядке, равном 200 - в 1.6 раза.

Цель, поставленная перед началом работы, была достигнута. В ходе лабораторной работы были решены следующие задачи:

- были изучены последовательный и параллельный варианты алгоритма Флойда поиска кратчайших путей между двумя любыми вершинами в графе;
- были разработаны изученные алгоритмы;
- был проведен сравнительный анализ реализованных алгоритмов;
- был подготовлен отчет о выполненной лабораторной работе.

Список литературы

[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] Документация по языку C++ [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения: 8.11.2021).

[5] Date and time utilities [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono (дата обращения: 8.11.2021).

[6] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 8.11.2021).

[7] Ubuntu 20.04 LTS (Focal Fossa) Beta [Электронный ресурс]. Режим доступа: http://old-releases.ubuntu.com/releases/20.04.1/ (дата обращения: 8.11.2021).

[8] Мобильный процессор AMD Ryzen™ 3 3200U с графикой Radeon™ Vega 3 [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-3-3200u (дата обращения: 8.11.2021).