МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический университет
Петра Великого»

Институт компьютерных наук и кибербезопасности

Направление: 02.03.01 Математика и компьютерные науки

ОТЧЁТ О ВЫПОЛНЕНИИ НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЫ

Технологии параллельного программирования в операционных системах Linux

Студент,
группы 5130201/30002 _____ Михайлова А. А.

Преподаватель _____ Чуватов М. В.

«____»_____ 2025 г.

Санкт-Петербург, 2025

Реферат

Общий объём отчёта: 41 страница
Иллюстраций: 11
Использованных источников: 7
Количество приложений: 1

Цель работы – проектирование сети компьютеров, а также разработка терминального приложения, реализовываюшего некоторые алгоритмы на графах с применением методов параллельного программирования. В рамках достижения этой цели были реализованы алгоритмы для подсчёта количества остовных деревьев с помощью матричной теоремы Кирхгофа, поиска минимального остова алгоритмом Прима, кодирования деревьев посредством кода Прюфера, а также нахождения минимальной раскраски графа.

В процессе установлены виртуальная среда VirtualBox и операционная система Slackware.

В результате проведённой работы создано приложение, реализующие необходимые алгоритмы и корректно запускающееся на сети компьютеров. Основное достижение заключается в успешной адаптации классических алгоритмов теории графов к параллельным архитектурам, что позволило сократить время вычислений.

Содержание

Термины и определения

Виртуальная машина – изолированный программный контейнер, эмулирующий работу отдельного компьютера с собственной операционной системой, размещённый на физическом хост-устройстве.

Сеть NAT (Network Address Translation) – тип виртуальной сети, при котором гостевая система использует IP-адрес хоста для выхода во внешнюю сеть, обеспечивая изоляцию и экономию адресов.

Параллельное программирование – метод разработки программ, позволяющий выполнять несколько вычислений одновременно, с целью повышения производительности и ускорения решения сложных задач.

OpenMPI – реализация интерфейса передачи сообщений MPI (Message Passing Interface), предназначенная для написания параллельных приложений, работающих на распределённых системах.

OpenMP – API (интерфейс программирования приложений), поддерживающий многопоточное программирование в многопроцессорных системах с общей памятью, обычно используется в языках C/C++ и Fortran.

Операционная система – программное обеспечение, управляющее аппаратными ресурсами компьютера и предоставляющее интерфейс для взаимодействия с пользователем и другими программами. Отвечает за управление процессами, памятью, устройствами ввода-вывода и обеспечивает безопасность, стабильность и эффективность работы компьютера.

Slackware Linux – один из первых дистрибутивов Linux.

SSH (Secure Shell) – это протокол для безопасного удаленного доступа к компьютерам и серверам через интернет. Он позволяет выполнять команды на удаленной машине, передавать файлы и управлять другими сетевыми сервисами.

Введение

В рамках научно-исследовательской работы была поставлена задача изучить технологии параллельного программирования в операционной системе на базе ядра Linux. Необходимо было освоить основные принципы и методы параллельного программирования, а также понять особенности и преимущества использования этой ОС для выполнения параллельных задач.

Целью работы являлось:

- Создание и настройка сети виртуальных компьютеров;
- Разработка параллельных версий алгоритмов теории графов;
- Создание приложения для изучения параллельных вычислений.

В работе необходимо реализовать следующие методы:

- Нахождение количества остовных деревьев;
- Алгоритм Прима для поиска минимального остова;
- Кодирование Прюфера для минимального остова;
- Нахождение минимальной раскраски графа.

Компьютеры в сети управлялись ОС Slackware. IP заданы в адресном пространстве 10.0.50.0/24. Использовалась среда виртуализации Oracle VirtualBox. Приложение разрабатывалось в среде Visual Studio Code, подключенной к одной из виртуальных машин с помощью ssh.

1 Настройка окружения

1.1 Создание виртуальных машин и настройка ОС

Был скачан образ Slackware с официального сайта. При создании виртуальной машины указаны её название, папка для хранения данных и путь до образа ISO. Был указан тип системы Linux и подтип Other Linux. Я создала 3 виртуальных машины, для каждой вылелила 2 виртуальных ядра и 3 гб оперативной памяти. Под жёсткий диск выделено 20 гб, однако система динамически выделяет место.

После создания ВМ была запущена для установки ОС. Сеть была настроена вручную. IP-адрес машины – адрес из доступного диапазона (10.0.50.0/24). Первый адрес узла всегда занят виртуальным маршрутизатором VirtualBox, он является шлюзом по умолчанию и адресом сервера DNS. ВМ были назначены адреса – 10.0.50.101, 10.0.50.102, 10.0.50.103. Имена машин – vm1, clon, vmclon. Домен hpc.

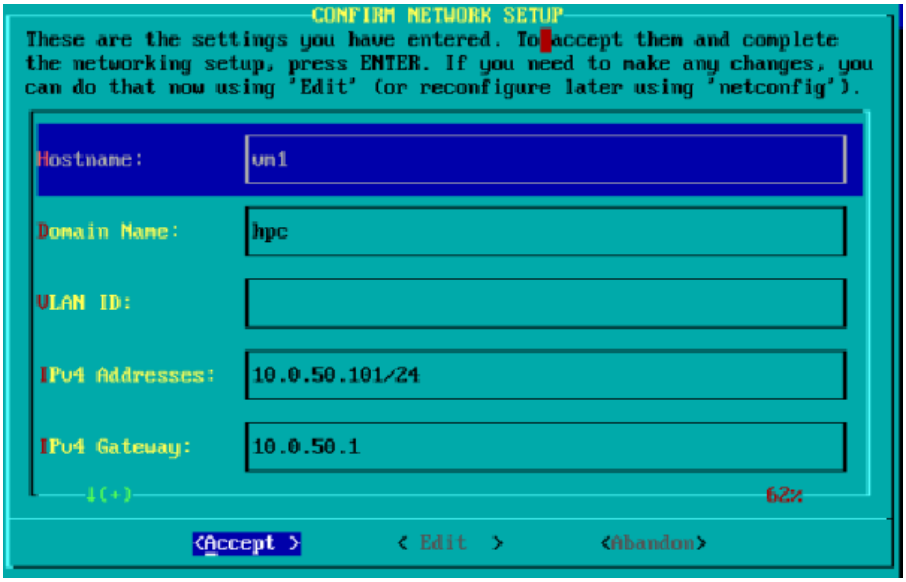

Рис. 1: netconfig

Рис. 2: netconfig

1.2 hosts

Для обращения к другим машинам по коротким именам (vm1, clon, vmclon) был настроен файл /etc/hosts. Для каждой машины были перечислены адреса остальных машин – рис. 3.

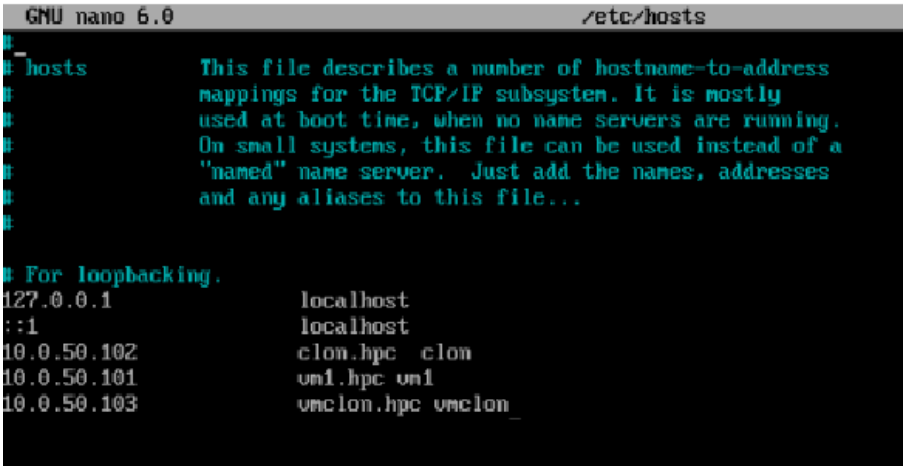

Рис. 3: /etc/hosts

1.3 Создание пользователя

Следующим шагом были настроены учетные записи суперпользователя (root) и пользователя (user1). Учётная запись суперпользователя создана автоматически и для неё задан пароль. Вход в систему от имени администратора или переключение выполняются с помощью команды su -. Запуск программ на Linux с использованием root может повлечь за собой неисправимые последствия для ОС. Поэтому с помощью adduser был создан пользователь.

1.4 SSH, NFS

Затем на все машины были установлены необходимые программные пакеты, обеспечивающие работу протокола SSH, настройку общей файловой системы через NFS.

Для обеспечения взаимодействия между машинами без запроса пароля был настроен механизм SSH-ключей(ssh-keygen). Публичный ключ был скопирован на все машины, выполняя команду ssh-copy-id со всех машин. После их выполнение стало возможным подключение по SSH с одной виртуальной машины на другую.

Для синхронизации исполняемых файлов и данных между машинами была настроена общая сетевая файловая система с использованием протокола NFS. Порядок действий:

1. Выбор сервера NFS - машина с IP-адресом 10.0.50.101. На ней размещена общая директория;
2. На сервере создана директория, доступная всем клиентам /mnt/share/;
3. В файл /etc/exports была добавлена строка: /mnt/share/ *(rw,sync, no_subtree_check) для разрешения чтения и записи из любой машины в подсети;

4. Служба NFS была перезапущена и на двух других машинах была смонтирована общая директория с сервера: mount 10.0.50.101:/mnt/share /mnt/share.

   Для автоматического монтирования при загрузке запись была добавлена в файл /ets/fstab: 10.0.50.101:/mnt/share /mnt/share nfs rw, defaults 0 0.

   Директория /mnt/share стала доступной на всех виртуальных машинах.

1.5 Настройка сети

Для обеспечения взаимодействия между виртуальными машинами и подключения к внешней сети была создана виртуальная сеть типа NAT. Адресное пространство было задано как 10.0.50.0/24. Для ручного управления IP-адресами была отключена DHCP. После настройки сети были выполнены пробросы портов для всех машин для удалённого подключения по протоколу SSH с хостовой машины. Каждая виртуальная машина получила статический IP-адрес в заданном раннее диапазоне – рис. 4, 5.

Рис. 4: Сеть NAT

Рис. 5: Проброс портов

1.6 OpenMP и MPI

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, C++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Message Passing Interface (MPI, интерфейс передачи сообщений) – программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации

для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

MPI была установлена с официального сайта с помощью общей папки и команд:

```
mkdir -p /mnt/openmpi
mount -t vboxsf op /mnt/openmpi
ls /mnt/openmpi
installpkg /mnt/openmpi/pkgname
```

1.7 Директивы OpenMP

Использованные директивы библиотеки OpenMP:

- #pragma omp parallel for – распределяет итерации цикла между потоками;
- #pragma omp atomic – атомарный доступ к ячейке памяти;
- #pragma omp parallel for reduction(...) – параллельный цикл с операцией приведения, объединяя значения переменных из разных потоков в единый результат;
- #pragma omp parallel – параллельная область, в которой код выполняется несколькими потоками одновременно;
- #pragma omp for – приводит к тому, что работа в цикле for внутри параллельного региона будет разделена между потоками;
- #pragma omp critical – блок кода, который может выполняться только одним потоком за раз;
- #pragma omp for nowait – переопределяет барьер, неявный в директиве.

1.8 Функции MPI

- MPI_Comm_rank(MPI_COMM_WORLD, &rank) — возвращает номер текущего процесса (идентификатор процесса, rank);
- MPI_Comm_size(MPI_COMM_WORLD, &size) — возвращает общее количество запущенных процессов (size);
- MPI_Bcast(buffer, count, datatype, root, MPI_COMM_WORLD) — рассылает данные от одного процесса (с номером root) всем остальным;

- MPI_Allreduce(...) — аналог MPI_Reduce, но результат рассылается всем процессам.
- MPI_Init() — инициализирует среду MPI и готовит её к использованию.
- MPI_Finalize() — завершает работу с MPI и освобождает все выделенные ресурсы.
- MPI_Barrier(MPI_COMM_WORLD) — блокирует выполнение до тех пор, пока все процессы в коммуникаторе не достигнут этой точки (синхронизация).

2 Особенности реализации

Исходный код следующих методов представлен в приложении.

2.1 Класс OrientedGraph

Класс OrientedGraph реализует ориентированный граф, его матрицу весов и матрицу смежности, а также методы. Далее будут описаны методы данного класса.

2.1.1 generateRandomGraph()

Вход: запрос на генерацию нового графа
Выход: граф и матрица весов для него
Метод генерирует граф в виде матрицы смежности и матрицу весов для него. Несколько потоков внутри одного процесса одновременно генерируют ребра для разных вершин графа.

2.1.2 generateWeightMatrix()

Вход: запрос на генерацию нового графа
Выход: матрица весов
Метод реализует матрицу весов для графа.

2.1.3 kirghoff()

Вход: матрица смежности
Выход: матрица Кирхгоффа и число остовных деревьев
Метод реализует формирование матрицы кирхгоффа и подсчет числа остовных деревьев. OpenMP используется для ускорения подготовки матриц на каждом узле.

2.1.4 det()

Вход: матрица Кирхгоффа
Выход: результат подсчета определителя
Метод реализует подсчет определителя матрицы, на 1 порядок меньшей, чем матрица Кирхгоффа.

2.1.5 broadcast_matrices()

Вход: состояние объектов внутри класса
Выход: измененное состояние
Метод рассылает копии матриц смежности и весов от главного процесса всем остальным процессам в группе.

2.1.6 find_mst_parallel()

Вход: граф

Выход: минимальное остовное дерево и его вес

Метод реализует алгоритм Прима для нахождения минимального остовного дерева. Вершины графа делятся на группы и каждый MPI-процесс отвечает за свою группу. Ускоряется поиск лучщей вершины для обновления расстояний до других вершин.

2.1.7 prufer_code_parallel()

Вход: минимальный остов

Выход: код Прюфера

Метод реализует кодирование остова с помощью кода Прюфера. MPI начально синхронизирует данные, а далее процессы выполняют одни и те же шаги, синхронно изменяя свои копии данных, распараллелены начальный подсчет степеней вершин, поиск минимального листа.

2.1.8 graphColoring_parallel()

Вход: граф

Выход: минимальная раскраска графа

Метод реализует алгоритм минимальной раскраски графа. Вершины делятся на группы, и MPI отвечает за свою группу, ускоряется создание неориентированной копии графа, проверка вершин, запись новых цветов в массив.

2.2 main()

Вход: инициализация параметров программы

Выход: результаты вызванных алгоритмов

Функция реализует основное меню программы. MPI синхронизирует все процессы.

3 Результаты работы программы

После запуска программы пользователя просят ввести количество вершин. После чего выводится меню, матрица смежности и матрица весов – рис. 6.

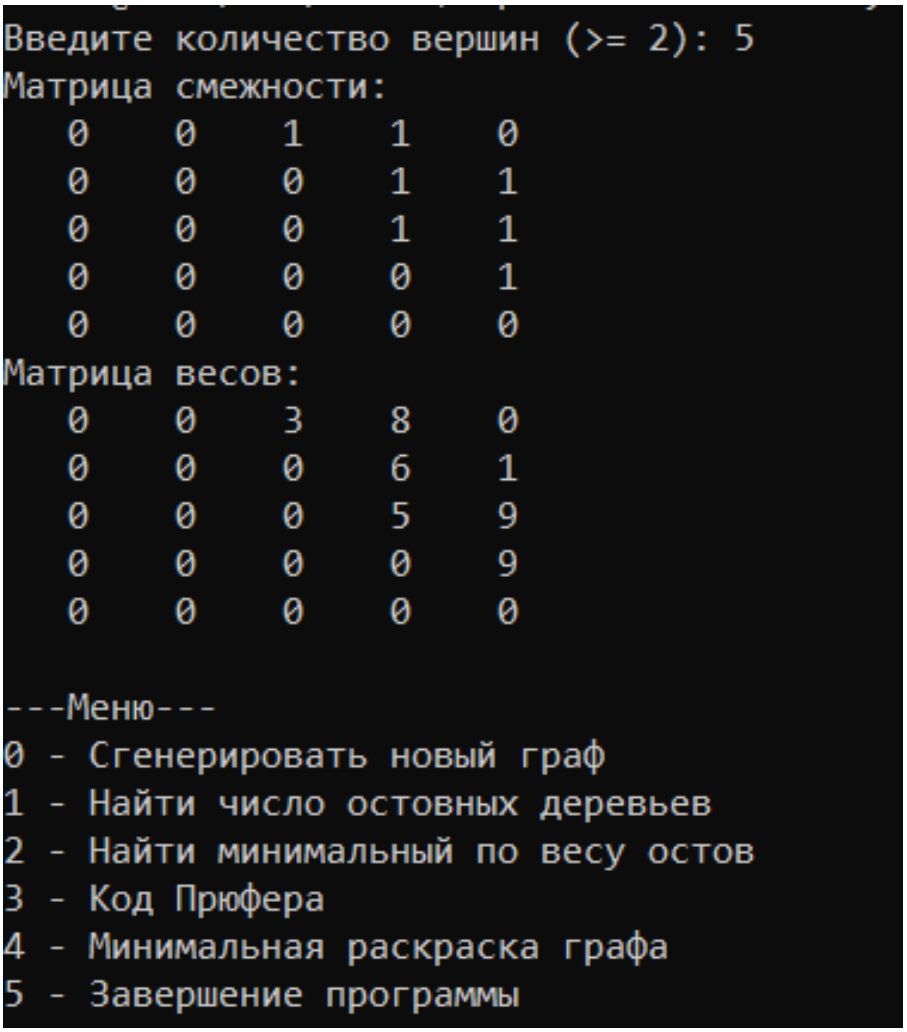

Рис. 6: Запуск программы и начальное меню

При выборе 0 повторяется ввод количества вершин и выводится информация о новом графе – рис. 7.

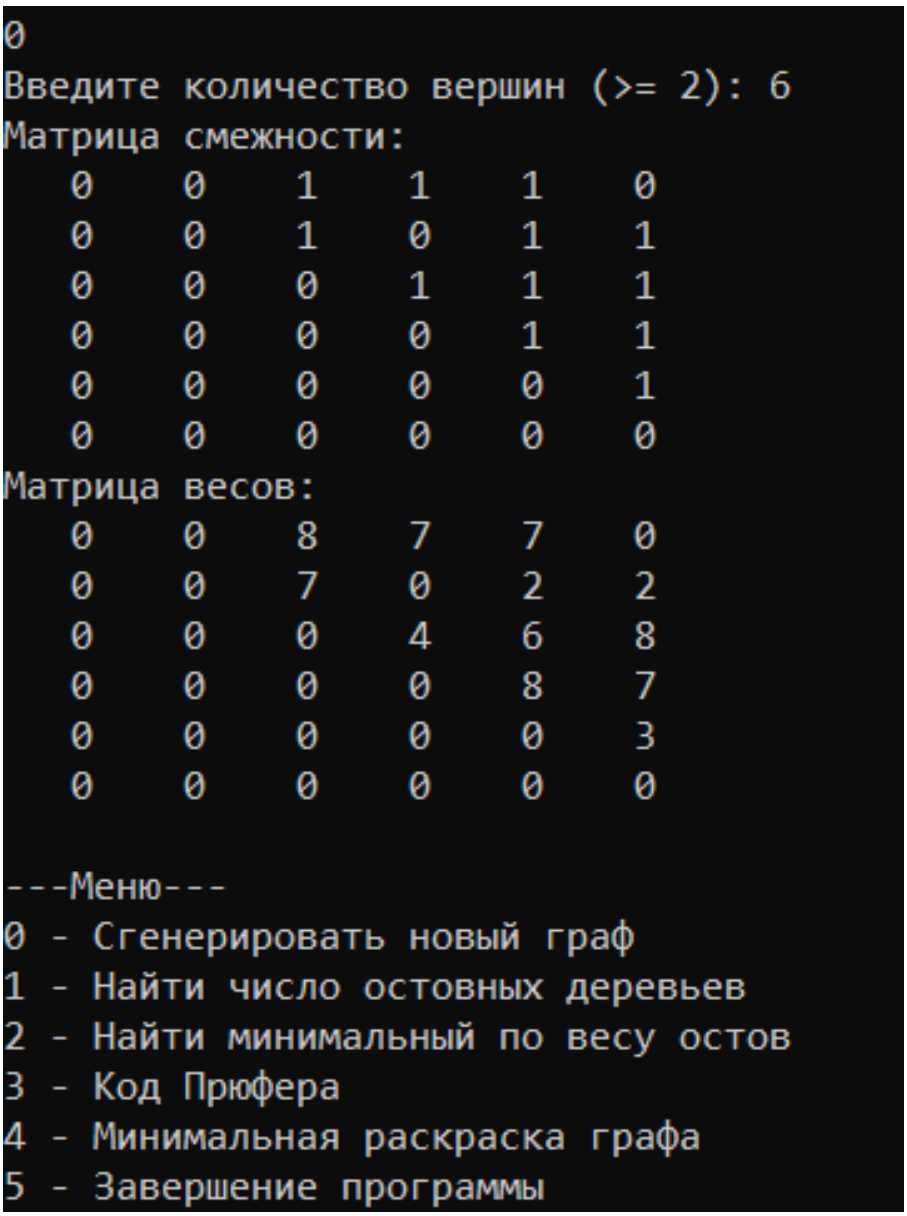

Рис. 7: Генерация нового графа

При выборе 1 выводится число остовных деревьев и матрица Кирхгоффа – рис. 8.

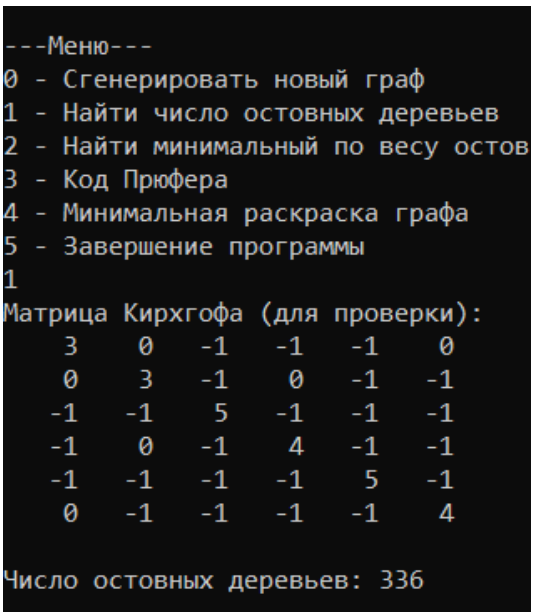

Рис. 8: Число остовных деревьев

При выборе 2 выводится минимальный по весу остов и его вес – рис. 9.

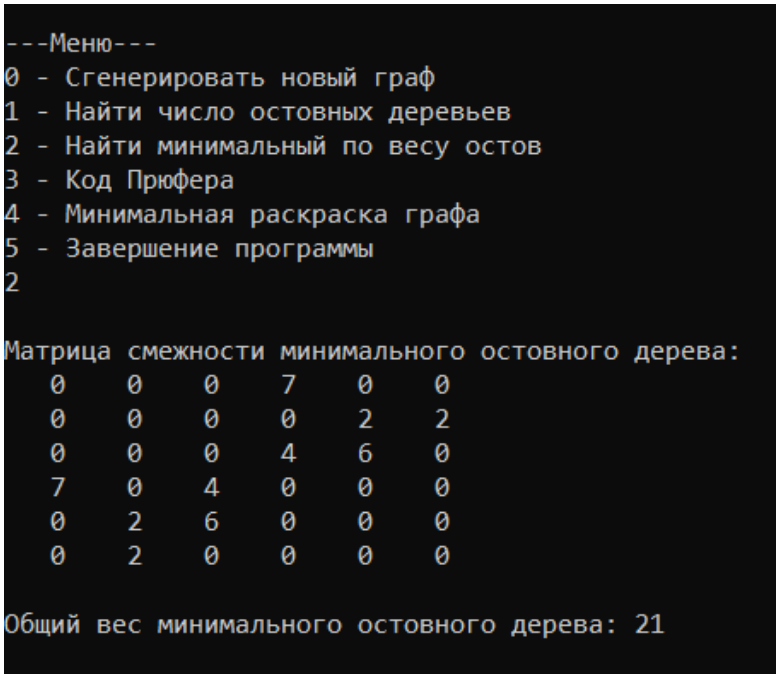

Рис. 9: Минимальное остовное дерево

При выборе 3 выводится код Прюфера для минимального остова – рис. 10.

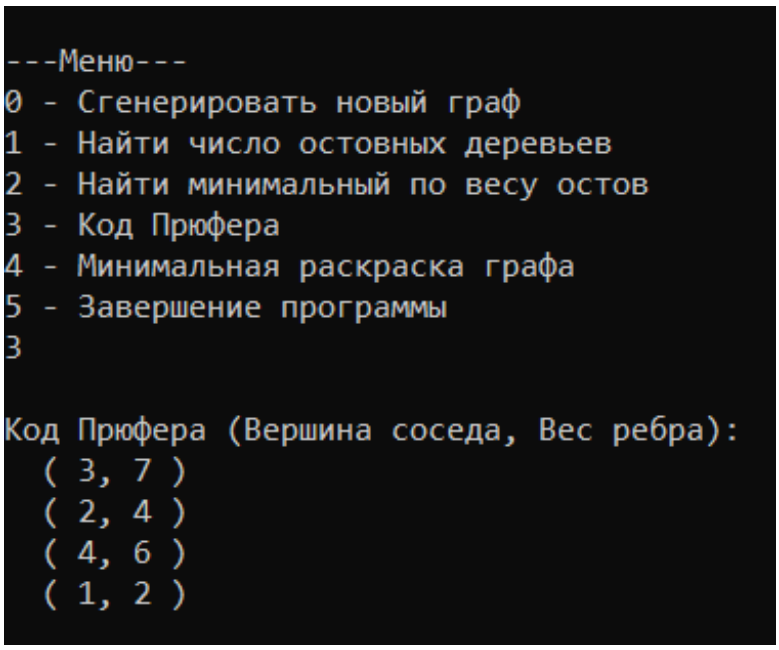

Рис. 10: Код Прюфера

При выборе 4 выводится минимальная раскраска графа – рис. 11.

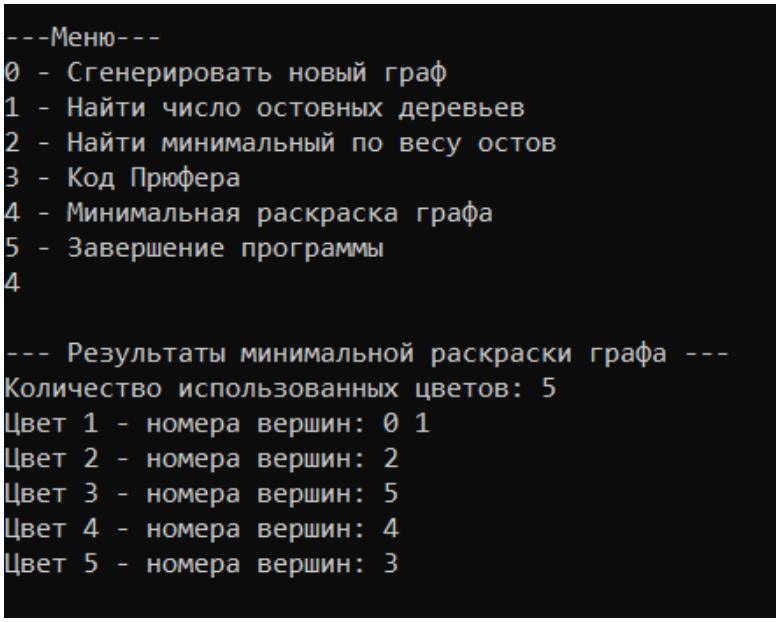

Рис. 11: Минимальная раскраска графа

При вводе 5 программа завершается.

Заключение

В ходе выполнения научно-исследовательской работы были изучены и применены методы параллельного программирования с использованием технологий OpenMP и MPI. Реализовано приложение на языке C++, настроена сеть виртуальных машин для запуска распределённых вычислений. Созданы 3 виртуальные машины, изучены основы взаимодействия с ОС Linux на примере Slackware. Реализованы параллельные версии алгоритмов теории графов: подсчёт количества остовных деревьев, поиск минимального остова по алгоритму Прима, кодирование через код Прюфера, нахождение минимальной раскраски графа.

Список литературы

[1] Секция «Телематика». — Текст : электронный // tema.spbstu.ru : [сайт]. — URL: https://tema.spbstu.ru/tgraph/ (дата обращения: 29.06.2025).

[2] The Slackware Linux Project `http://www.slackware.com/` (дата обращения: 20.06.2025)

[3] OpenMPI [Электронный ресурс] / Lawrence Livermore National Laboratory. — Режим доступа: `https://hpc-tutorials.llnl.gov/mpi/`, свободный. — Дата обращения: 28.06.2025.

[4] OpenMP [Электронный ресурс] / Lawrence Livermore National Laboratory. — Режим доступа: `https://hpc-tutorials.llnl.gov/openmp/`, свободный. — Дата обращения: 28.06.2025.

[5] MPI: A Message-Passing Interface Standard. Version 3.1. — High Performance Computing Center Stuttgart, 2015. — 866 с.

[6] OpenMP Application Program Interface. Version 5.0. — OpenMP Architecture Review Board, 2020. — 530 с.

[7] Форсайт Дж., Малькольм М., Моулер К. Машинные методы математических вычислений. — М.: Мир, 1980. — 280 с.

Приложение А. Исходный код code.cpp

```
#include <iostream>
#include <vector>
#include <random>
#include <iomanip>
#include <mpi.h>
#include <omp.h>
#include <cstdlib>
#include <ctime>
#include <stdexcept>
#include <climits>
#include <set>
#include <numeric>
#include <algorithm>
#include <thread>

#define OUTPUTPHRASE "\nМеню------\n0 - Сгенерировать новый граф\n1 - Найти число остовных деревьев\n2 - Найти минимальный по весу остов\n3 - Код Прюфера\n4 - Минимальная раскраска графа\n5 - Завершение программы\n"

class OrientedGraph
{
protected:
int numVertices;
int numEdges;
int world_rank;
std::vector<std::vector<int>> adjacencyMatrix;
std::vector<std::vector<int>> weightMatrix;

int factorial(int n)
{
return (n <= 1) ? 1 : n * factorial(n - 1);
}

int erlangRandomDegree()
{
const double lambda = 3;
double sum = 0.0;
for (int i = 0; i < 1; ++i)
{
        sum += pow(lambda, i) / factorial(i);
```

```
39 }
40 double result = 1.0 - exp(-lambda) * sum;
41 return static_cast<int>(round(result * numVertices));
42 }
43
44 public:
45 OrientedGraph(int vertices, int rank) : numVertices(
     vertices), numEdges(0), world_rank(rank)
46 {
47 adjacencyMatrix.resize(vertices, std::vector<int>(
     vertices, 0));
48 weightMatrix.resize(vertices, std::vector<int>(vertices
     , 0));
49 }
50
51 void printAdjacencyMatrix() const
52 {
53 std::cout << "Матрица␣смежности:\n";
54 for (const auto &row : adjacencyMatrix)
55 {
56         for (int val : row)
57         {
58                 std::cout << std::setw(4) << val << "␣"
     ;
59         }
60         std::cout << '\n';
61 }
62 }
63
64 void printWeightMatrix() const
65 {
66 std::cout << "Матрица␣весов:\n";
67 for (const auto &row : weightMatrix)
68 {
69         for (int val : row)
70         {
71                 std::cout << std::setw(4) << val << "␣"
     ;
72         }
73         std::cout << '\n';
74 }
75 }
76
```

```
77 void generateRandomGraph()
78 {
79 std::srand(std::time(0) + omp_get_thread_num());
80 #pragma omp parallel for
81 for (int i = 0; i < numVertices; ++i)
82 {
83         int edgesFromVertex = erlangRandomDegree();
84         edgesFromVertex = std::min(edgesFromVertex,
   numVertices - i - 1);
85
86         for (int j = 0; j < edgesFromVertex; ++j)
87         {
88                 int targetVertex = i + 1 + std::rand()
   % (numVertices - i - 1);
89
90                 if (adjacencyMatrix[i][targetVertex] ==
    0 && targetVertex < numVertices)
91                 {
92                         adjacencyMatrix[i][targetVertex
   ] = 1;
93                         #pragma omp atomic
94                         numEdges++;
95                 }
96         }
97 }
98
99 generateWeightMatrix();
100 }
101
102 void generateWeightMatrix()
103 {
104 std::srand(std::time(0) + omp_get_thread_num());
105 #pragma omp parallel for
106 for (int i = 0; i < numVertices; i++)
107 {
108         for (int j = 0; j < numVertices; j++)
109         {
110                 if (adjacencyMatrix[i][j] == 1)
111                 {
112                         weightMatrix[i][j] = std::rand
   () % 9 + 1;
113                 }
114         }
```

```
115 }
116 }
117
118 int getNumVertices() const { return numVertices; }
119
120 int kirghoff()
121 {
122 int world_size, world_rank;
123 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
124 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
125
126 std::vector<std::vector<int>> kirghoff_matrix(
    numVertices, std::vector<int>(numVertices, 0));
127 #pragma omp parallel for
128 for (int i = 0; i < numVertices; i++)
129 {
130         int degree = 0;
131         for (int j = 0; j < numVertices; j++)
132         {
133                 if (adjacencyMatrix[i][j] ||
    adjacencyMatrix[j][i])
134                 {
135                         kirghoff_matrix[i][j] = -1;
136                         degree++;
137                 }
138         }
139         kirghoff_matrix[i][i] = degree;
140 }
141 std::vector<std::vector<double>> a(numVertices - 1, std
    ::vector<double>(numVertices - 1));
142 #pragma omp parallel for
143 for (int i = 0; i < numVertices - 1; i++)
144 {
145         for (int j = 0; j < numVertices - 1; j++)
146         {
147                 a[i][j] = kirghoff_matrix[i][j];
148         }
149 }
150 if (world_rank == 0)
151 {
152         std::cout << "Матрица␣Кирхгофа␣для␣(␣проверки):\n";
153         for (int i = 0; i < numVertices; i++)
154         {
```

```
155                     for (int j = 0; j < numVertices; j++)
156                     {
157                             std::cout << std::setw(5) <<
    kirghoff_matrix[i][j];
158                     }
159                     std::cout << std::endl;
160             }
161             std::cout << "\Числоn␣остовных␣деревьев:␣";
162     }
163     double determinant = det(a);
164     return static_cast<int>(round(determinant));
165     }
166
167     double det(const std::vector<std::vector<double>> &
    matrix)
168     {
169     size_t n = matrix.size();
170     for (const auto &row : matrix)
171     {
172             if (row.size() != n)
173             {
174                     throw std::invalid_argument("Матрица␣
    должна␣быть␣квадратной\n");
175             }
176     }
177     if (n == 1)
178     {
179             return matrix[0][0];
180     }
181     if (n == 2)
182     {
183             return matrix[0][0] * matrix[1][1] - matrix
    [0][1] * matrix[1][0];
184     }
185     double determinant = 0;
186     #pragma omp parallel for reduction(+ : determinant)
187     for (size_t col = 0; col < n; col++)
188     {
189             std::vector<std::vector<double>> minor;
190             for (size_t i = 1; i < n; i++)
191             {
192                     std::vector<double> row;
193
```

```
                for (size_t j = 0; j < n; j++)
                {
                        if (j != col)
                        {
                                row.push_back(matrix[i][j]);
                        }
                }

                minor.push_back(row);
        }

        double cofactor = matrix[0][col] * det(minor);

        if (col % 2 == 1)
        {
                cofactor = -cofactor;
        }

        determinant += cofactor;
}

return determinant;
}

void broadcast_matrices()
{
std::vector<int> flat_adj(numVertices * numVertices);
std::vector<int> flat_weight(numVertices * numVertices);
if (world_rank == 0)
{
        for (int i = 0; i < numVertices; ++i)
        {
                for (int j = 0; j < numVertices; ++j)
                {
                        flat_adj[i * numVertices + j] = adjacencyMatrix[i][j];
                        flat_weight[i * numVertices + j] = weightMatrix[i][j];
                }
        }
}
```

```
233 MPI_Bcast(flat_adj.data(), numVertices * numVertices,
      MPI_INT, 0, MPI_COMM_WORLD);
234 MPI_Bcast(flat_weight.data(), numVertices * numVertices
      , MPI_INT, 0, MPI_COMM_WORLD);
235 if (world_rank != 0)
236 {
237         for (int i = 0; i < numVertices; ++i)
238         {
239                 for (int j = 0; j < numVertices; ++j)
240                 {
241                         adjacencyMatrix[i][j] =
      flat_adj[i * numVertices + j];
242                         weightMatrix[i][j] =
      flat_weight[i * numVertices + j];
243                 }
244         }
245 }
246 }
247
248 void find_mst_parallel()
249 {
250 int world_size;
251 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
252 const int INF = std::numeric_limits<int>::max();
253 std::vector<std::vector<int>> g(numVertices, std::
      vector<int>(numVertices));
254 #pragma omp parallel for
255 for (int i = 0; i < numVertices; ++i)
256 {
257         for (int j = 0; j < numVertices; ++j)
258         {
259                 if (adjacencyMatrix[i][j] ||
      adjacencyMatrix[j][i])
260                 {
261                         int w = (weightMatrix[i][j] !=
      0) ? weightMatrix[i][j] : weightMatrix[j][i];
262                         g[i][j] = w;
263                 }
264                 else
265                 {
266                         g[i][j] = INF;
267                 }
268         }
```

```
269 }
270
271 std::vector<int> counts(world_size);
272 std::vector<int> displs(world_size);
273 int chunk_size = numVertices / world_size;
274 for (int i = 0; i < world_size; ++i)
275 {
276         counts[i] = chunk_size;
277 }
278 counts[world_size - 1] += numVertices % world_size;
279 displs[0] = 0;
280 for (int i = 1; i < world_size; ++i)
281 {
282         displs[i] = displs[i - 1] + counts[i - 1];
283 }
284 int start_v = displs[world_rank];
285 int end_v = start_v + counts[world_rank];
286 std::vector<int> min_e(numVertices, INF);
287 std::vector<int> sel_e(numVertices, -1);
288 std::vector<bool> used(numVertices, false);
289 min_e[0] = 0;
290 struct
291 {
292         int value;
293         int index;
294 } local_min, global_min;
295
296 for (int i = 0; i < numVertices; ++i)
297 {
298         local_min.value = INF;
299         local_min.index = -1;
300
301         #pragma omp parallel
302         {
303                 struct
304                 {
305                         int val;
306                         int idx;
307                 } thread_min = {INF, -1};
308                 #pragma omp for
309                 for (int j = start_v; j < end_v; ++j)
310                 {
```

```
311                         if (!used[j] && min_e[j] <
    thread_min.val)
312                         {
313                                 thread_min.val = min_e[
    j];
314                                 thread_min.idx = j;
315                         }
316                 }
317                 #pragma omp critical
318                 {
319                         if (thread_min.val < local_min.
    value)
320                         {
321                                 local_min.value =
    thread_min.val;
322                                 local_min.index =
    thread_min.idx;
323                         }
324                 }
325         }
326
327         MPI_Allreduce(&local_min, &global_min, 1,
    MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
328         int u = global_min.index;
329         if (u == -1 || global_min.value == INF)
330         {
331                 if (world_rank == 0 && i < numVertices
    - 1)
332                 std::cout << "Граф не связный, невозможно
    построить MST!\n";
333                 break;
334         }
335
336         used[u] = true;
337         #pragma omp parallel for
338         for (int v = start_v; v < end_v; ++v)
339         {
340                 if (!used[v] && g[u][v] < min_e[v])
341                 {
342                         min_e[v] = g[u][v];
343                         sel_e[v] = u;
344                 }
345         }
```

```
346         std::vector<int> local_min_e_slice(counts[
    world_rank]);
347         for (int k = 0; k < counts[world_rank]; ++k)
348         local_min_e_slice[k] = min_e[start_v + k];
349         MPI_Allgatherv(local_min_e_slice.data(), counts
    [world_rank], MPI_INT, min_e.data(), counts.data(),
    displs.data(), MPI_INT, MPI_COMM_WORLD);
350         std::vector<int> local_sel_e_slice(counts[
    world_rank]);
351         for (int k = 0; k < counts[world_rank]; ++k)
352         local_sel_e_slice[k] = sel_e[start_v + k];
353         MPI_Allgatherv(local_sel_e_slice.data(), counts
    [world_rank], MPI_INT, sel_e.data(), counts.data(),
    displs.data(), MPI_INT, MPI_COMM_WORLD);
354 }
355
356 if (world_rank == 0)
357 {
358         long long total_weight = 0;
359         std::vector<std::vector<int>> mst_adj_matrix(
    numVertices, std::vector<int>(numVertices, 0));
360         for (int i = 1; i < numVertices; ++i)
361         {
362                 if (sel_e[i] != -1)
363                 {
364                         int u_res = sel_e[i];
365                         int v_res = i;
366                         int weight = g[u_res][v_res];
367                         if (weight != INF)
368                         {
369                                 total_weight += weight;
370                                 mst_adj_matrix[u_res][
    v_res] = weight;
371                                 mst_adj_matrix[v_res][
    u_res] = weight;
372                         }
373                 }
374         }
375
376         std::cout << "\Матрица\n смежности минимального 
    остовного дерева:\n";
377         for (const auto &row : mst_adj_matrix)
378         {
```

```
379                 for (int val : row)
380                 {
381                         std::cout << std::setw(4) <<
    val << " ";
382                 }
383                 std::cout << '\n';
384         }
385         std::cout << "\Общийn вес минимального остовного
    дерева: " << total_weight << std::endl;
386 }
387 }
388
389 std::vector<std::vector<int>> get_mst_parallel()
390 {
391 int world_size;
392 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
393 const int INF = std::numeric_limits<int>::max();
394 std::vector<std::vector<int>> g(numVertices, std::
    vector<int>(numVertices));
395 #pragma omp parallel for
396 for (int i = 0; i < numVertices; ++i)
397 {
398         for (int j = 0; j < numVertices; ++j)
399         {
400                 if (adjacencyMatrix[i][j] ||
    adjacencyMatrix[j][i])
401                 {
402                         int w = (weightMatrix[i][j] !=
    0) ? weightMatrix[i][j] : weightMatrix[j][i];
403                         g[i][j] = w;
404                 }
405                 else
406                 {
407                         g[i][j] = INF;
408                 }
409         }
410 }
411
412 std::vector<int> counts(world_size), displs(world_size)
    ;
413 int chunk_size = numVertices / world_size;
414 for (int i = 0; i < world_size; ++i)
415 counts[i] = chunk_size;
```

```
416 counts[world_size - 1] += numVertices % world_size;
417 displs[0] = 0;
418 for (int i = 1; i < world_size; ++i)
419 displs[i] = displs[i - 1] + counts[i - 1];
420 int start_v = displs[world_rank], end_v = start_v +
      counts[world_rank];
421 std::vector<int> min_e(numVertices, INF), sel_e(
      numVertices, -1);
422 std::vector<bool> used(numVertices, false);
423 min_e[0] = 0;
424 struct
425 {
426         int value;
427         int index;
428 } local_min, global_min;
429
430 for (int i = 0; i < numVertices; ++i)
431 {
432         local_min.value = INF;
433         local_min.index = -1;
434         #pragma omp parallel
435         {
436                 struct
437                 {
438                         int val;
439                         int idx;
440                 } thread_min = {INF, -1};
441                 #pragma omp for
442                 for (int j = start_v; j < end_v; ++j)
443                 {
444                         if (!used[j] && min_e[j] <
      thread_min.val)
445                         {
446                                 thread_min.val = min_e[
      j];
447                                 thread_min.idx = j;
448                         }
449                 }
450                 #pragma omp critical
451                 if (thread_min.val < local_min.value)
452                 local_min = {thread_min.val, thread_min
      .idx};
453         }
```

```
454
455         MPI_Allreduce(&local_min, &global_min, 1,
    MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
456         int u = global_min.index;
457         if (u == -1 || global_min.value == INF)
458         {
459                 if (world_rank == 0)
460                 {
461                         bool is_mst_complete = true;
462                         for (int vert_idx = 0; vert_idx
     < numVertices; ++vert_idx)
463                         if (!used[vert_idx])
464                         is_mst_complete = false;
465                         if (!is_mst_complete)
466                         std::cout << "Граф не связный,
    невозможно построить MST!\n";
467                 }
468                 break;
469         }
470         used[u] = true;
471         #pragma omp parallel for
472         for (int v = start_v; v < end_v; ++v)
473         {
474                 if (!used[v] && g[u][v] < min_e[v])
475                 {
476                         min_e[v] = g[u][v];
477                         sel_e[v] = u;
478                 }
479         }
480         std::vector<int> local_min_e(counts[world_rank
    ]), local_sel_e(counts[world_rank]);
481         std::copy(min_e.begin() + start_v, min_e.begin
    () + end_v, local_min_e.begin());
482         std::copy(sel_e.begin() + start_v, sel_e.begin
    () + end_v, local_sel_e.begin());
483         MPI_Allgatherv(local_min_e.data(), counts[
    world_rank], MPI_INT, min_e.data(), counts.data(),
    displs.data(), MPI_INT, MPI_COMM_WORLD);
484         MPI_Allgatherv(local_sel_e.data(), counts[
    world_rank], MPI_INT, sel_e.data(), counts.data(),
    displs.data(), MPI_INT, MPI_COMM_WORLD);
485 }
```

```
std::vector<std::vector<int>> mst_adj_matrix(numVertices, std::vector<int>(numVertices, 0));
if (world_rank == 0)
{
        bool connected = true;
        for (int i = 0; i < numVertices; ++i)
        if (!used[i])
        connected = false;
        if (connected)
        {
                for (int i = 1; i < numVertices; ++i)
                {
                        if (sel_e[i] != -1)
                        {
                                int u_res = sel_e[i], v_res = i, weight = g[u_res][v_res];
                                mst_adj_matrix[u_res][v_res] = weight;
                                mst_adj_matrix[v_res][u_res] = weight;
                        }
                }
        }
}
return mst_adj_matrix;
}

void prufer_code_parallel()
{
std::vector<std::vector<int>> tree = get_mst_parallel();

std::vector<int> flat_tree(numVertices * numVertices);
if (world_rank == 0)
{
        bool is_empty = true;
        for (const auto &row : tree)
        for (int val : row)
        if (val != 0)
        is_empty = false;
        if (is_empty)
        return;

```

```
524             for (int i = 0; i < numVertices; ++i)
525             for (int j = 0; j < numVertices; ++j)
526             flat_tree[i * numVertices + j] = tree[i][j];
527     }
528     MPI_Bcast(flat_tree.data(), numVertices * numVertices,
          MPI_INT, 0, MPI_COMM_WORLD);
529     if (world_rank != 0)
530     {
531             for (int i = 0; i < numVertices; ++i)
532             for (int j = 0; j < numVertices; ++j)
533             tree[i][j] = flat_tree[i * numVertices + j];
534     }
535     std::vector<int> degrees(numVertices);
536     #pragma omp parallel for
537     for (int i = 0; i < numVertices; ++i)
538     {
539             int d = 0;
540             for (int j = 0; j < numVertices; ++j)
541             {
542                     if (tree[i][j] != 0)
543                     d++;
544             }
545             degrees[i] = d;
546     }
547     std::vector<std::pair<int, int>> prufer_sequence;
548     if (world_rank == 0)
549     prufer_sequence.reserve(numVertices - 2);
550     for (int k = 0; k < numVertices - 2; ++k)
551     {
552             int min_leaf = std::numeric_limits<int>::max();
553             #pragma omp parallel for reduction(min :
          min_leaf)
554             for (int i = 0; i < numVertices; ++i)
555             {
556                     if (degrees[i] == 1)
557                     {
558                             min_leaf = std::min(min_leaf, i
          );
559                     }
560             }
561             int neighbor = -1;
562             for (int j = 0; j < numVertices; ++j)
563             {
```

```
564                     if (tree[min_leaf][j] != 0)
565                     {
566                             neighbor = j;
567                             break;
568                     }
569             }
570
571             if (world_rank == 0)
572             {
573                     prufer_sequence.push_back({neighbor,
    tree[min_leaf][neighbor]});
574             }
575             degrees[min_leaf] = 0;
576             degrees[neighbor]--;
577             tree[min_leaf][neighbor] = 0;
578             tree[neighbor][min_leaf] = 0;
579     }
580     if (world_rank == 0)
581     {
582             std::cout << "\Код␣Прюфера␣Вершина(␣соседа,␣Вес␣
    ребра):␣" << std::endl;
583             for (const auto &p : prufer_sequence)
584             {
585                     std::cout << "␣␣(␣" << p.first << ",␣"
    << p.second << "␣)" << std::endl;
586             }
587     }
588     }
589
590     void graphColoring_parallel()
591     {
592     int world_size;
593     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
594     std::vector<std::vector<int>> graph(numVertices, std::
    vector<int>(numVertices, 0));
595     #pragma omp parallel for
596     for (int i = 0; i < numVertices; i++)
597     {
598             for (int j = 0; j < numVertices; j++)
599             {
600                     if (adjacencyMatrix[i][j] != 0 ||
    adjacencyMatrix[j][i] != 0)
601                     {
```

```
602                             graph[i][j] = 1;
603                     }
604             }
605     }
606     std::vector<int> priorities(numVertices);
607     if (world_rank == 0)
608     {
609             std::iota(priorities.begin(), priorities.end(),
        0);
610             std::random_device rd;
611             std::mt19937 g(rd());
612             std::shuffle(priorities.begin(), priorities.end
       (), g);
613     }
614     MPI_Bcast(priorities.data(), numVertices, MPI_INT, 0,
       MPI_COMM_WORLD);
615     std::vector<int> colors(numVertices, -1);
616     long long uncolored_nodes = numVertices;
617     int chunk_size = numVertices / world_size;
618     int start_v = world_rank * chunk_size;
619     int end_v = (world_rank == world_size - 1) ?
       numVertices : start_v + chunk_size;
620
621     while (uncolored_nodes > 0)
622     {
623             std::vector<int> local_newly_colored_indices;
624             std::vector<int> local_newly_colored_colors;
625             #pragma omp parallel
626             {
627                     std::vector<int> thread_indices;
628                     std::vector<int> thread_colors;
629                     #pragma omp for nowait
630                     for (int u = start_v; u < end_v; ++u)
631                     {
632                             if (colors[u] != -1)
633                             continue;
634                             bool can_be_colored = true;
635                             for (int v = 0; v < numVertices
       ; ++v)
636                             {
637                                     if (u == v || graph[u][
       v] == 0 || colors[v] != -1)
638                                     continue;
```

```
639
640                                 if (priorities[v] >
    priorities[u] || (priorities[v] == priorities[u] &&
    v > u))
641                                 {
642                                         can_be_colored
    = false;
643                                         break;
644                                 }
645                         }
646                         if (can_be_colored)
647                         {
648                                 std::vector<bool>
    available_colors(numVertices, true);
649                                 for (int v = 0; v <
    numVertices; ++v)
650                                 {
651                                         if (graph[u][v]
     && colors[v] != -1)
652                                         {
653
    available_colors[colors[v]] = false;
654                                         }
655                                 }
656
657                                 int chosen_color = 0;
658                                 for (chosen_color = 0;
    chosen_color < numVertices; ++chosen_color)
659                                 {
660                                         if (
    available_colors[chosen_color])
661                                         break;
662                                 }
663
664                                 thread_indices.
    push_back(u);
665                                 thread_colors.push_back
    (chosen_color);
666                         }
667                 }
668
669                 #pragma omp critical
670                 {
```

```
671                         local_newly_colored_indices.
insert(local_newly_colored_indices.end(),
thread_indices.begin(), thread_indices.end());
672                         local_newly_colored_colors.
insert(local_newly_colored_colors.end(),
thread_colors.begin(), thread_colors.end());
673                 }
674         }
675         int local_count = local_newly_colored_indices.
size();
676         std::vector<int> all_counts(world_size);
677         std::vector<int> displs(world_size, 0);
678
679         MPI_Allgather(&local_count, 1, MPI_INT,
all_counts.data(), 1, MPI_INT, MPI_COMM_WORLD);
680
681         int total_newly_colored = 0;
682         for (int i = 0; i < world_size; ++i)
683         {
684                 displs[i] = total_newly_colored;
685                 total_newly_colored += all_counts[i];
686         }
687
688         if (total_newly_colored == 0 && uncolored_nodes
 > 0)
689         {
690                 if (world_rank == 0)
691                 {
692                         std::cout << "Не␣удалось␣
окрасить␣оставшиеся␣вершины.␣Возможно,␣граф␣несвязный.\n";
693                 }
694                 break;
695         }
696
697         std::vector<int> global_new_indices(
total_newly_colored);
698         std::vector<int> global_new_colors(
total_newly_colored);
699
700         MPI_Allgatherv(local_newly_colored_indices.data
(), local_count, MPI_INT,
701         global_new_indices.data(), all_counts.data(),
displs.data(), MPI_INT, MPI_COMM_WORLD);
```

```
702         MPI_Allgatherv(local_newly_colored_colors.data
    (), local_count, MPI_INT,
703         global_new_colors.data(), all_counts.data(),
    displs.data(), MPI_INT, MPI_COMM_WORLD);
704         #pragma omp parallel for
705         for (int i = 0; i < total_newly_colored; ++i)
706         {
707                 colors[global_new_indices[i]] =
    global_new_colors[i];
708         }
709
710         uncolored_nodes -= total_newly_colored;
711 }
712 if (world_rank == 0)
713 {
714         std::cout << "\n---␣Результаты␣минимальной␣
    раскраски␣графа␣---\n";
715         int max_color = 0;
716         for (int c : colors)
717         {
718                 if (c > max_color)
719                 {
720                         max_color = c;
721                 }
722         }
723         int num_colors_used = (uncolored_nodes == 0) ?
    (max_color + 1) : 0;
724         std::cout << "Количество␣использованных␣цветов:␣" <<
     num_colors_used << std::endl;
725
726         for (int c = 0; c < num_colors_used; ++c)
727         {
728                 std::cout << "Цвет␣" << (c + 1) << "␣-␣
    номера␣вершин:␣";
729                 for (int i = 0; i < numVertices; ++i)
730                 {
731                         if (colors[i] == c)
732                         {
733                                 std::cout << i << "␣";
734                         }
735                 }
736                 std::cout << std::endl;
737         }
```

```
738 }
739 }
740 };
741
742 int main(int argc, char **argv)
743 {
744 MPI_Init(&argc, &argv);
745 int world_size;
746 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
747 int world_rank;
748 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
749
750 {
751 const int MAX_ALLOWED_THREADS = 20;
752 const char *omp_env = getenv("OMP_NUM_THREADS");
753 if (omp_env != nullptr)
754 {
755         try
756         {
757                 int requested_threads = std::stoi(
    omp_env);
758                 if (requested_threads >
    MAX_ALLOWED_THREADS)
759                 {
760                         if (world_rank == 0)
761                         {
762                                 std::cout << "
    ПРЕДУПРЕЖДЕНИЕ[] Запрошено " << requested_threads
763                                 << " потоков. Максимально
    допустимое значение ограничено до "
764                                 << MAX_ALLOWED_THREADS
    << "." << std::endl;
765                         }
766                         omp_set_num_threads(
    MAX_ALLOWED_THREADS);
767                 }
768         }
769         catch (const std::exception &e)
770         {
771         }
772 }
773 }
774
```

```
775 int vertices;
776 OrientedGraph *graph = nullptr;
777 while (true)
778 {
779 if (world_rank == 0)
780 {
781         do
782         {
783                 std::cout << "Введите количество вершин (>= 2): ";
784                 std::cin >> vertices;
785                 if (std::cin.fail() || vertices < 2)
786                 {
787                         std::cout << "Некорректный ввод. Попробуйте снова.\n";
788                         std::cin.clear();
789                         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
790                         vertices = 0;
791                 }
792         } while (vertices < 2);
793 }
794 MPI_Bcast(&vertices, 1, MPI_INT, 0, MPI_COMM_WORLD);
795 delete graph;
796 graph = new OrientedGraph(vertices, world_rank);
797 if (world_rank == 0)
798 {
799         graph->generateRandomGraph();
800         graph->printAdjacencyMatrix();
801         graph->printWeightMatrix();
802 }
803 graph->broadcast_matrices();
804 while (true)
805 {
806         if (world_rank == 0)
807         {
808                 std::cout << OUTPUTPHRASE;
809         }
810         int symbol;
811         if (world_rank == 0)
812         {
813                 std::cin >> symbol;
814                 if (std::cin.fail())
```

```
815                 {
816                         std::cin.clear();
817                         std::cin.ignore(std::
    numeric_limits<std::streamsize>::max(), '\n');
818                         symbol = -1;
819                 }
820         }
821         MPI_Bcast(&symbol, 1, MPI_INT, 0,
    MPI_COMM_WORLD);
822         if (symbol == 0)
823         {
824                 break;
825         }
826         else if (symbol == 5)
827         {
828                 delete graph;
829                 MPI_Finalize();
830                 return 0;
831         }
832         MPI_Barrier(MPI_COMM_WORLD);
833         switch (symbol)
834         {
835                 case 1:
836                 if (world_rank == 0)
837                 {
838                         int result = graph->kirghoff();
839                         std::cout << result << std::
    endl;
840                 }
841                 break;
842                 case 2:
843                 graph->find_mst_parallel();
844                 break;
845                 case 3:
846                 graph->prufer_code_parallel();
847                 break;
848                 case 4:
849                 graph->graphColoring_parallel();
850                 break;
851                 default:
852                 if (world_rank == 0)
853                 std::cout << "Неверный пункт меню.\n";
854                 break;
```

```
855             }
856             MPI_Barrier(MPI_COMM_WORLD);
857 }
858 }
859 delete graph;
860 MPI_Finalize();
861 return 0;
862 }
863
```