МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Отчет по лабораторной работе №4 по дисциплине «Параллельное программирование»

Выполнил студент группы ИВТ-32_____/Рзаев А. Э./
Проверил доцент кафедры ЭВМ_____/Чистяков Г. А./

Киров 2018

1 Задание на лабораторную работу

1. Изучить основные принципы работы с интерфейсом MPI, освоить механизм передачи сообщений между процессами.
2. Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессоров.
3. Реализовать параллельную версию алгоритма с помощью языка C++, используя при этом предлагаемые интерфейсом MPI механизмы и виртуальные топологии (в случае применимости).
4. Показать корректность полученной реализации путём осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов.
5. Провести доказательную оценку эффективность MPI-реализации алгоритма.

2 Выделение областей для распараллеливания

Реализация жадного алгоритма из первой лабораторной работы выполняет k итераций раскраски, используя каждый раз разную перестановку вершин графа. Среди всех полученных значений количества цветов для раскраски выбирается наименьшее.

Данную реализацию алгоритма можно ускорить за счет выполнения итераций раскраски независимо друг от друга в несколько процессов, в каждом из которых будет выполнятся раскраски графа по перестановке вершин. Входные данные каждый процесс считывает отдельно из файла.

Листинг MPI-реализации алгоритма на C++ приведен в приложении А.

3 Графическая схема, иллюстрирующая взаимодействие процессов

Графическая схема изображена на рисунке 1.

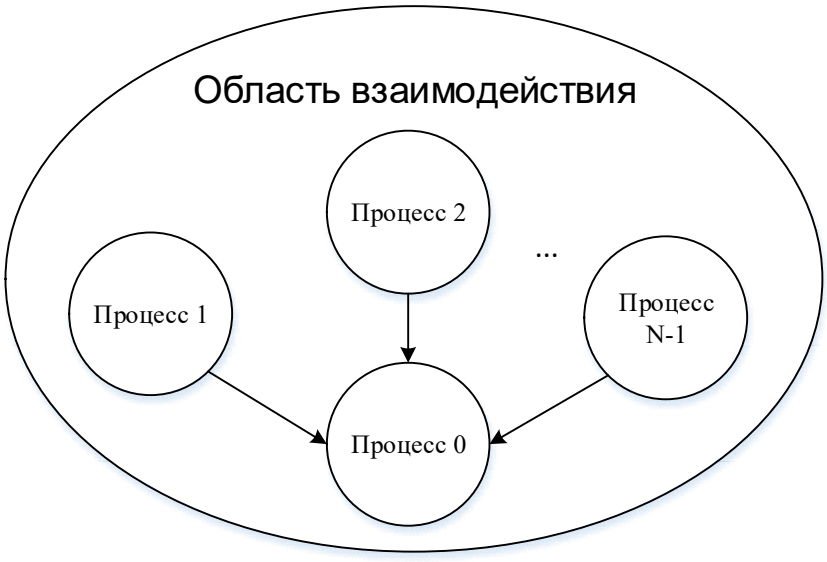

Рисунок 1 – Графическая схема

Нулевой процесс отвечает за сбор результатов работы алгоритма раскраски. Остальные процессы (1, 2, …, N-1) выполняют раскраску графа и отправляют результат нулевому процессу.

4 Тестирование

Тестирование проводилось на ЭВМ под управлением 64-разрядной ОС Windows 10, с 4 ГБ оперативной памяти, с процессором Intel Core i3 6006U с частотой 2.0 ГГц (4 логических и 2 физических ядра).

Среди всех времен выполнения процессов выбиралось максимальное.

В ходе тестирование было установлено, что максимальная производительность достигается при одновременной работе 16 процессов. Ускорение по сравнению с однопоточной реализацией составляет 3.1 раза.

Результаты тестирования приведены в таблице 1.

Таблица 1 — Результаты тестирования (в мс)

N	4	8	14	16	18	Линейная программа
	2559	2233	2219	2138	2339	6516
	6083	5271	5045	4882	5059	15174
	12998	11241	10695	10621	10818	33223
				Среднее		3.10
				Максимальное		3.13
				Минимальное		3.07

5 Вывод

В ходе выполнения лабораторной работы был изучен интерфейс MPI, принципы создания многопроцессных приложений с его использованием. На основе знаний, полученных в ходе лекционного материала по MPI, был разработан параллельный алгоритм жадной раскраски графа.

Параллельный алгоритм, реализованный с помощью MPI, оказался несколько быстрее алгоритма, реализованного с помощью OpenMP (3.1 против 3.0). Было достигнуто более чем 3-кратное ускорение по сравнению с линейной реализацией.

Приложение А
(обязательное)
Листинг программной реализации

algo.h

```
#include <vector>
#include <iostream>
#include <cstdint>
#include <algorithm>
#include <random>
#include <chrono>
#include <numeric>

using namespace std::chrono;

using graph_t = std::vector<std::vector<size_t>>;

std::istream& operator>>(std::istream& is, graph_t& graph) {
    size_t n; is >> n; // vertexes
    size_t m; is >> m; // edges
    graph.clear();
    graph.resize(n);

    for (size_t i = 0; i < m; ++i) {
        size_t a, b; is >> a >> b;
        graph[a].push_back(b);
        graph[b].push_back(a);
    }

    return is;
}

namespace improved {

    size_t _colorize(const graph_t &, const std::vector<size_t> &);

    size_t colorize(const graph_t &graph, size_t perm_count) {
        std::vector<size_t> order(graph.size());
        std::iota(order.begin(), order.end(), 0);

        std::vector<std::vector<size_t>> perms(perm_count);

        std::mt19937 g(static_cast<unsigned int>(time(nullptr)));
        for (size_t i = 0; i < perm_count; ++i) {
            std::shuffle(order.begin(), order.end(), g);
            perms[i] = order;
        }

        std::vector<size_t> results(perm_count, graph.size());

        for (size_t i = 0; i < perm_count; i++) {
            results[i] = _colorize(graph, perms[i]);
        }

        return *std::min_element(results.begin(), results.end());
    }

    inline size_t _mex(const std::vector<size_t> &set) {
        return std::find(set.begin(), set.end(), 0) - set.begin();
    }
```

```
    size_t _colorize(const graph_t &graph, const std::vector<size_t> &order) {
        size_t size = graph.size();

        std::vector<size_t> colored(size, 0);
        std::vector<size_t> colors(size, 0);
        std::vector<size_t> used_colors(size, 0);

        for (size_t v : order) {
            if (!colored[v]) {
                for (auto to : graph[v]) {
                    if (colored[to]) {
                        used_colors[colors[to]] = 1;
                    }
                }
                colored[v] = 1;

                auto color = _mex(used_colors);
                colors[v] = color;
                used_colors.assign(size, 0);
            }
        }

        return 1 + *std::max_element(colors.begin(), colors.end());
    }

}
```

main.cpp

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>

#include "algo.h"

const int MASTER_RANK = 0;

void master_routine(int ws) {
    std::ifstream input("../input.txt");
    std::ofstream output("../output.txt");

    size_t cnt;
    input >> cnt;
    output << cnt << std::endl;

    std::vector<size_t> results(cnt, 1000000u), times(cnt, 0);

    for (size_t i = 0; i < cnt; ++i) {
        size_t rank_res[2];

        for (int rank = 1; rank < ws; ++rank) {
            MPI_Recv(&rank_res, 2, MPI_UNSIGNED_LONG, rank, i, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            auto [time, res] = rank_res;

            results[i] = std::min(results[i], res);
            times[i] = std::max(times[i], time);
        }
        output << std::setw(4) << results[i] << ' ' << times[i] << std::endl;
        std::cout << times[i] << std::endl;
    }
}
```

```
void slave_routine(int wr, int ws) {
    size_t perm_count = 500u / (ws - 1);
    std::ifstream input("../input.txt");

    graph_t graph;
    size_t cnt;
    input >> cnt;

    for (size_t i = 0; i < cnt; ++i) {
        input >> graph;

        auto start = std::chrono::system_clock::now();

        size_t result = improved::colorize(graph, perm_count);

        auto stop = std::chrono::system_clock::now();
        auto time = duration_cast<milliseconds>(stop - start).count();

        size_t buffer[2] = {static_cast<size_t>(time), result};

        MPI_Send(&buffer, 2, MPI_UNSIGNED_LONG, MASTER_RANK, i, MPI_COMM_WORLD);
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank == 0) {
        master_routine(world_size);
    } else {
        slave_routine(world_rank, world_size);
    }

    MPI_Finalize();
}
```