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

Лабораторная работа № 2 по курсу «Параллельное программирование»

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

Киров 2018

1 Задание

Изучить средства работы с потоками операционной системы, получить навыки реализации многопоточных приложений.

1. Выделить в полученной в ходе первой лабораторной работы реализации жадного алгоритма раскраски графа фрагменты кода, выполнение которых может быть распределено на несколько процессорных ядер.
2. Реализовать многопоточную версию алгоритма с помощью языка C++ и потоков стандартной библиотеки C++, использую при этом необходимые примитивы синхронизации.
3. Показать корректность полученной реализации, запустив ее на наборе тестов, построенных в ходе первой лабораторной.
4. Провести доказательную оценку эффективности многопоточной реализации алгоритма.

2 Метод распараллеливания алгоритма

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 в несколько потоков, в каждом из которых создается заданное количество перестановок вершин графа и по ней выполняется его раскраска.

Листинг многопоточной реализации алгоритма на C++ приведен в приложении А.

3 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).

Количество вершин и ребер каждого графа и результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты тестирования

Граф (вершины, ребра)	Линейная реализация, мс	Параллельная реализация, мс	Ускорение
200, 15000	1540	689,8	2,233
550, 113437	12275	5320,0	2,307
750, 168750	17808	7970,5	2,234
930, 324337	36047	15161,8	2,377
1250, 585937	66092	27390,0	2,413
1420, 756150	83561	35139,0	2,378
1680, 1058400	114252	48823,5	2,340
		Среднее	2,326
		Максимальное	2,413
		Минимальное	2,233

4 Вывод

В ходе лабораторной работы была разработана многопоточная версия жадного алгоритма раскраски графа с использованием потоков стандартной библиотеки C++. Многопоточный алгоритм оказался быстрее однопоточного на всех тестовых входных данных; в среднем ускорение составило 2.3 раза. Исходя из этого можно предположить, что многопоточная реализация будет быстрее при любых входных данных.

Приложение А
(обязательное)
Листинг программной реализации

algo.h

```
#include <vector>
#include <iostream>
#include <cstdint>
#include <set>
#include <algorithm>
#include <future>

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;
}

std::ostream& operator<<(std::ostream& os, const graph_t& graph) {
    std::set<std::pair<size_t, size_t>> edges;
    for (size_t v = 0; v < graph.size(); ++v) {
        for (size_t to : graph[v]) {
            edges.emplace(std::minmax(v, to));
        }
    }
    os << graph.size() << ' ' << edges.size() << std::endl;
    for (const auto& p : edges) {
        os << p.first << ' ' << p.second << std::endl;
    }

    return os;
}

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

    size_t colorize(const graph_t& graph) {
        size_t orders_count = 500;

        const size_t thread_count = 4;
        size_t results[thread_count] = { 0 };
```

```
    auto routine = [&graph](size_t* r, size_t first, size_t last) {
        std::vector<size_t> order(graph.size());
        for (size_t i = 0; i < graph.size(); ++i) {
            order[i] = i;
        }
        size_t min = graph.size();
        for (size_t i = first; i < last; ++i) {
            std::random_shuffle(order.begin(), order.end());
            min = std::min(min, _colorize(graph, order));
        }
        *r = min;
    };
    std::thread threads[thread_count];
    for (size_t i = 0; i < thread_count; ++i) {
        size_t len = orders_count / thread_count;
        threads[i] = std::thread(routine, results + i, i * len, (i + 1) * len);
    }
    for (auto& thread : threads) {
        thread.join();
    }
    return *std::min_element(results, results + thread_count);    }

bool _check_coloring(const graph_t& graph, const std::vector<size_t>& coloring) {
    for (int v = 0; v < graph.size(); ++v) {
        for (size_t to : graph[v]) {
            if (coloring[v] == coloring[to]) {
                return false;
            }
        }
    }
    return true;
}

size_t _mex(const std::set<size_t>& set) {
    if (set.empty()) {
        return 0;
    }
    auto max = *set.rbegin();
    for (size_t c = 0; c <= max + 1; ++c) {
        if (set.find(c) == set.end()) {
            return c;
        }
    }
}

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);
    for (size_t i = 0; i < size; ++i) {
        size_t v = order[i];
        if (!colored[v]) {
            std::set<size_t> used_colors;
            for (auto to : graph[v]) {
```

```
                    if (colored[to]) {
                        used_colors.emplace(colors[to]);
                    }
                }
                auto color = _mex(used_colors);
                colored[v] = 1;
                colors[v] = color;
            }
        }
        if (_check_coloring(graph, colors)) {
            std::set<size_t> unique_colors(colors.cbegin(), colors.cend());
            return unique_colors.size();
        } else {
            return graph.size();
        }
    }
}
```

main_improved.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <iomanip>

#include "algo.h"

using namespace std::chrono;

int main() {
    std::ifstream input("../input.txt");
    std::ofstream output("../output.txt");

    graph_t graph; size_t cnt;
    input >> cnt;
    output << cnt << std::endl;
    for (size_t i = 0; i < cnt; ++i) {
        input >> graph;
        auto start = std::chrono::system_clock::now();
        auto res = improved::colorize(graph);
        auto stop = std::chrono::system_clock::now();
        auto time = duration_cast<milliseconds>(stop - start).count();
        output << std::setw(3) << graph.size() << ' '
               << std::setw(4) << res << ' '
               << time << std::endl;
    }
    return 0;
}
```