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

Отчет по лабораторной работе №3
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

Познакомиться со стандартом OpenMP, получить навыки реализации многопоточных SPMD-приложений с применением OpenMP.

1. Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора.
2. Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер.
3. Реализовать многопоточную версию алгоритма с помощью языка C++ и библиотеки OpenMP, использую при этом необходимые примитивы синхронизации.
4. Показать корректность полученной реализации путем осуществления на построенном в ходе первой лабораторной работы наборе тестов.
5. Провести доказательную оценку эффективность OpenMP-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 потоков, в каждом из которых будет выполнятся раскраски графа по перестановке вершин. Порядок, по которому потоки будут брать перестановки, определяется директивами OpenMP.

Листинг OpenMP-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).

В ходе тестирования было проверено три варианта планирования: static, dynamic и guided. В большинстве случаев быстрее всего оказался метод guided при размере параметра chunk равном либо 1, либо k/N, где N – количество потоков.

Результаты тестирования приведены в таблице 1.

Таблица 1 — Результаты тестирования (в мс)

chunk\schedule	1	10	50	100	125 = k/N	Линейная программа
static	2248	2297	2464	3086	2261	6516
	5155	5292	5729	7126	5221	15174
	11028	11434	12308	15396	11123	33223
dynamic	2230	2275	2482	3086	2231	
	5148	5245	5700	7087	5099	
	11184	11373	12212	15453	11035	
guided	2221	2243	2402	2753	2221	
	5143	5150	5584	6360	5132	
	10993	11144	12011	13702	10977	
				Среднее		2.98
				Максимальное		3.00
				Минимальное		2.96

4 Вывод

В ходе выполнения лабораторной работы был изучен стандарт OpenMP, его применение и директивы. На основе знаний, полученных в ходе лекционного материала по OpenMP, был разработан параллельный алгоритм жадной раскраски графа.

Параллельный алгоритм, реализованный с помощью OpenMP, оказался быстрее алгоритма, реализованного с помощью потоков стандартной библиотеки C++ (ускорение в 2.3 раза). Было достигнуто почти 3-кратное ускорение по сравнению с линейной реализацией.

Приложение А
(обязательное)
Листинг программной реализации

algo.h

```
#include <omp.h>
#include <vector>
#include <iostream>
#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 {

    constexpr size_t _thread_count = 3;

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

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

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

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

#pragma omp parallel for num_threads(_thread_count) schedule(guided, 125)
        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());
    }

}
```