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 №5 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 программным пакетом PVM, освоить механизм передачи сообщений между процессами и соединения ЭВМ в вычислительную сеть.
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 PVM механизмы.
4. Показать корректность полученной реализации путём осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов.
5. Провести доказательную оценку эффективности PVM-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 PVM-реализации алгоритма на C++ приведен в приложении А.

3 Графическая схема, иллюстрирующая взаимодействие процессов

Графическая схема изображена на рисунке 1.

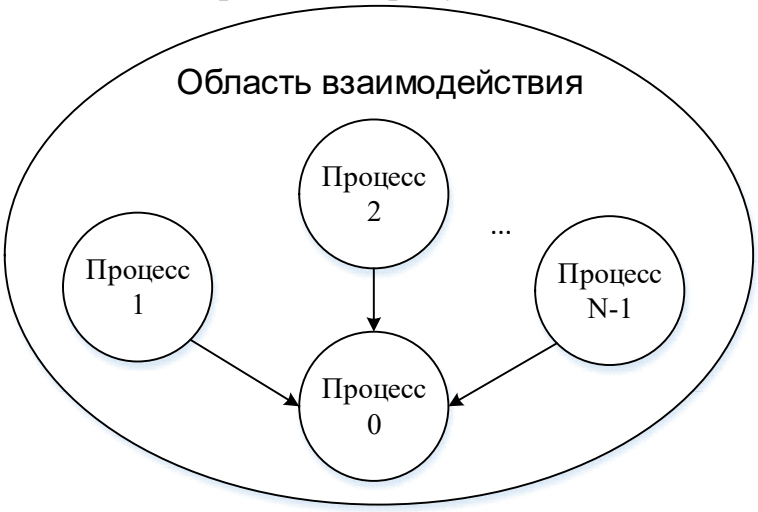

Рисунок 1 – Графическая схема

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 сети из 2 и из 3 ЭВМ. Для упрощения процесса развертывания вычислительной сети тестируемая программа запускалась на виртуальных машинах с 64-разрядной ОС Ubuntu 16.04.4 с 1 ГБ ОЗУ. Виртуальной машине выделялись все доступные ядра процессора хост-системы. Для подключения виртуальных машин в одну локальную сеть использовался сетевой мост, хост-системы подключались к общей WiFi-сети.

Сеть из 2 ЭВМ:

- Процессор: Intel Core i5, 2.3 ГГц, 2 ядра
- Процессор: Intel Core i3, 2 ГГц, 2 физических и 4 логических ядра

Среди всех времен выполнения процессов выбиралось среднее.

В ходе тестирования было установлено, что максимальная производительность достигается при одновременной работе 14 процессов. Ускорение по сравнению с однопоточной реализацией составляет 5.8 раз. Процессорных ядер — 6.

Результаты тестирования приведены в таблице 1.

Таблица 1 — Результаты тестирования (в мс)

N	8	10	12	14	16	Линейная программа
1680 1058400	1165	1154	1150	1136	1164	6516
2568 2472984	2733	2740	2694	2621	2750	15174
3782 5363821	5954	5712	5670	5662	6423	33223
				Среднее		5.8
				Максимальное		5.83
				Минимальное		5.78

Сеть из 3 ЭВМ:

- Процессор: Intel Core i5, 2.3 ГГц, 2 ядра
- Процессор: Intel Core i3, 2 ГГц, 2 физических и 4 логических ядра
- Процессор: AMD A6-9210, 2.4 ГГц, 2 ядра

В ходе тестирования было установлено, что максимальная производительность достигается при одновременной работе 16 процессов.

Ускорение по сравнению с однопоточной реализацией составляет 7.2 раз. Процессорных ядер — 8.

Результаты тестирования приведены в таблице 2.

Таблица 2 — Результаты тестирования (в мс)

N	10	12	14	16	18	Линейная программа
1680 1058400	1031	977	993	948	964	6516
2568 2472984	2393	2219	2257	2207	2214	15174
3782 5363821	5163	5193	4723	4521	4526	33223
				Среднее		7.1
				Максимальное		7.34
				Минимальное		6.92

5 Вывод

В ходе выполнения лабораторной работы был изучен программный пакет PVM, принципы создания мнопроцессных приложений для запуска в вычислительной сети. Получены навыки в построении вычислительных сетей. Разработан параллельный алгоритм жадной раскраски графа.

Данная многопроцессная реализация показала наилучший результат: получено почти линейное увеличение производительности при увеличении количества процессорных ядер — в среднем 0.93n, где n — количество ядер.

Наибольшее ускорение достигнуто при количестве процессов большем, чем количество ядер процессоров из-за особенностей ядра Linux.

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_master.cpp

```
#include <pvm3.h>
#include <iostream>
#include <memory>
#include <fstream>
#include <iomanip>
#include <vector>
#include <limits>
#include <chrono>
#include <cstdlib>

using namespace std::chrono;

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

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

    std::vector<int> results(cnt, std::numeric_limits<int>::max()), times(cnt, 0);

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

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

        for (int rank = 0; rank < ws; ++rank) {
            pvm_recv(chs[rank], -1);
            pvm_upkint(rank_res, 2, 1);

            int time = rank_res[0];
            int res = rank_res[1];
```

```
            results[i] = std::min(results[i], res);
            times[i] += time;
        }

        auto g_stop = std::chrono::system_clock::now();
        auto g_time = duration_cast<milliseconds>(g_stop - g_start).count();

        output << std::setw(4) << results[i] << ' '
               << times[i] / ws << std::endl;
        std::cout << g_time << ' ' << times[i] / ws << std::endl;
    }
}

int get_tasks_count(int argc, char** argv) {
    if (argc < 2) {
        return 4;
    }

    int cnt = atoi(argv[1]);

    if (cnt < 0 || cnt > 25) {
        return 4;
    } else {
        return cnt;
    }
}

int main(int argc, char** argv) {
    int cnt = get_tasks_count(argc, argv);
    int* children = new int[cnt];

    std::cout << "Children count: " << cnt << std::endl;

    pvm_mytid();
    pvm_spawn("main_slave", &argv[1], PvmTaskDefault, "", cnt, children);

    master_routine(cnt, children);

    pvm_exit();
    delete[] children;
}
```

main_slave.cpp

```
#include <pvm3.h>
#include <iostream>
#include <memory>
#include <fstream>
#include <iomanip>
#include <vector>
#include <numeric>

#include "algo.h"

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

    graph_t graph;
    size_t cnt;
    input >> cnt;

    for (size_t i = 0; i < cnt; ++i) {
```

```
        auto start = std::chrono::system_clock::now();
        input >> graph;
        auto stop = std::chrono::system_clock::now();

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

        auto time = (int)(duration_cast<milliseconds>(stop - start).count());

        int buffer[2] = {time, result};

        pvm_initsend(PvmDataRaw);
        pvm_pkint(buffer, 2, 1);
        pvm_send(pvm_parent(), 1);
    }
}

int get_tasks_count(int argc, char** argv) {
    if (argc < 2) {
        return 4;
    }

    int cnt = atoi(argv[1]);

    if (cnt < 0 || cnt > 25) {
        return 4;
    } else {
        return cnt;
    }
}

int main(int argc, char** argv) {
    int wr = pvm_mytid();
    int cnt = get_tasks_count(argc, argv);

    slave_routine(wr, cnt);
    pvm_exit();
}
```