Министерство образования и науки Российской Федерации Федеральное агентство по образованию

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

**Многопоточная реализация вычислительно сложного алгоритма с применением библиотеки OpenMP**

Лабораторная работа №3

по курсу «Параллельное программирование»

Вариант 9

Выполнил студент группы ИВТб-31 /Категов А. Д./ Проверил преподаватель /Долженкова М. Л./

Киров 2024

1. Задание

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

Этапы работы:

1. Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора
2. Выделить в полученной в ходе выполнения первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер
3. Реализовать многопоточную версию алгоритма с помощью языка С++ и библиотеки OpenMP, используя при этом необходимые примитивы синхронизации
4. Показать корректность полученной реализации путём осуществления на построенном в ходе первой лабораторной работы наборе тестов
5. Провести доказательную оценку эффективности OpenMP-реализации алгоритма
6. 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. Порядок, по которому потоки будут брать перестановки, определяется директивами OpenMP.

Следовательно, в качестве областей участков для распараллеливания при помощи OpenMP были выбраны те же участки, что и при простом распараллеливании. Это было сделано для наиболее точного сравнения результатов тестов.

1. Программная реализация

Листинг программной OpenMP-реализации алгоритма приведен в приложении А.

1. Тестирование

Тестирование проводилось на ЭВМ под управлением 64-разрядной OC Windows 10, с 8 ГБ оперативной памяти, с процессором AMD Ryzen 3 4300U 2,70GHz (4 физических ядра, 4 потока).

Результаты тестирования и сравнения с последовательной реализацией приведены в таблице 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Граф (кол-во вершин, кол-во ребер) | Последовательная реализация, мс | Параллельная реализация, мс | OpenMP-реализация, мс | Ускорение (последовательная) | Ускорение (Thread) |
| 5, 7 | 8 | 11 | 19 | 0,421053 | 0,578947 |
| 75, 1000 | 86 | 53 | 36 | 2,388889 | 1,472222 |
| 75, 2500 | 156 | 67 | 52 | 3,000000 | 1,288462 |
| 90, 300 | 83 | 51 | 42 | 1,97619 | 1,214286 |
| 95, 2500 | 151 | 74 | 52 | 2,903846 | 1,423077 |
| 150, 5000 | 270 | 116 | 91 | 2,967033 | 1,274725 |
| 150, 10000 | 420 | 147 | 145 | 2,896552 | 1,013793 |
| 300, 5000 | 375 | 155 | 145 | 2,586207 | 1,068966 |
| 300, 10000 | 510 | 220 | 178 | 2,865169 | 1,235955 |
| 170, 10000 | 427 | 119 | 157 | 2,719745 | 0,757962 |
| 250, 14000 | 584 | 224 | 158 | 3,696203 | 1,417722 |
| 250, 17000 | 660 | 218 | 182 | 3,626374 | 1,197802 |
| 300, 34000 | 1320 | 341 | 336 | 3,928571 | 1,014881 |
| 400, 40000 | 1452 | 422 | 411 | 3,532847 | 1,026764 |
| Среднее | 2,822048 | 1,141826 |
| Максимальное | 3,928571 | 1,472222 |
| Минимальное | 0,421053 | 0,578947 |

Исходя из результатов тестирования можно сказать, что в реализации данного алгоритма OpenMP дает небольшое увеличение быстродействия по сравнению с параллельной реализацией. Значительное ускорение, что не удивительно, присутствует при сравнении с последовательной реализацией алгоритма.

Распараллеливание более эффективно на больших значениях входных данных. При малых данных параллельная реализация проигрывает в быстродействии из-за дополнительных затрат времени на организацию параллельных вычислений и координацию потоков.

**Вывод**: В ходе выполнения лабораторной работы была реализована многопоточная версия жадного алгоритма закраски графа на языке С++ с использованием OpenMP. OpenMP версия оказалась эффективнее параллельной версии и намного эффективнее линейной версии на больших данных.

Приложение А

(обязательное)

Листинг программы

#include <vector>

#include <iostream>

#include <algorithm>

#include <fstream>

#include <chrono>

#include <thread>

#include <numeric>

using namespace std;

using namespace chrono;

using graph\_t = vector<vector<size\_t>>;

istream& operator>>(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;

}

size\_t \_mex(const vector<size\_t>& set) {

return static\_cast<size\_t>(find(set.begin(), set.end(), 0) - set.begin());

}

size\_t LineColorize(const graph\_t& graph) {

size\_t size = graph.size();

vector<size\_t> order(size);

size\_t orders\_count = 500;

vector<vector<size\_t>> orders(orders\_count);

/\*for (size\_t i = 0; i < size; ++i) { //O(size)

order[i] = i;

}\*/

iota(order.begin(), order.end(), 0);

for (auto& ord : orders) { //O(orders\_count)

random\_shuffle(order.begin(), order.end());

ord = order;

}

size\_t min = graph.size();

for (const auto& ord : orders) { //O(orders\_count)

vector<size\_t> colored(size, 0); //закрашеные вершины

vector<size\_t> colors(size, 0); //цвета вершин

vector<size\_t> used\_colors(size, 0); //использованые цвета

for (size\_t v : ord) { //O(size)

for (auto to : graph[v]) { //O(size)

if (colored[to]) {

used\_colors[colors[to]] = 1; //этот цвет занят

}

}

size\_t c = \_mex(used\_colors);//ищем незанятый цвет

colored[v] = 1; //помечаем вершину как раскрашеную

colors[v] = c; //раскрашиваем вершину

used\_colors.assign(size, 0); //сброс всех цветов

}

size\_t colors\_count = 1 + \*max\_element(colors.begin(), colors.end());

min = std::min(min, colors\_count);

}

return min; //O(size + orders\_count + orders\_count \* (size \* size)) = O(orders\_count \* size^2)

}

size\_t colorize\_chunk(const graph\_t& graph, const vector<vector<size\_t>>& orders, size\_t start, size\_t end) {

size\_t min = graph.size();

size\_t size = graph.size();

for (size\_t i = start; i < end; ++i) {

const auto& ord = orders[i];

vector<size\_t> colored(size, 0);

vector<size\_t> colors(size, 0);

vector<size\_t> used\_colors(size, 0);

for (size\_t v : ord) {

for (auto to : graph[v]) {

if (colored[to]) {

used\_colors[colors[to]] = 1;

}

}

size\_t c = \_mex(used\_colors);

colored[v] = 1;

colors[v] = c;

used\_colors.assign(size, 0);

}

size\_t colors\_count = 1 + \*max\_element(colors.begin(), colors.end());

min = std::min(min, colors\_count);

}

return min;

}

size\_t ParallelColorize(const graph\_t& graph) {

size\_t size = graph.size();

vector<size\_t> order(size);

size\_t orders\_count = 500;

vector<vector<size\_t>> orders(orders\_count);

iota(order.begin(), order.end(), 0);

for (auto& ord : orders) {

random\_shuffle(order.begin(), order.end());

ord = order;

}

size\_t min = graph.size();

size\_t num\_threads = 4;

vector<thread> threads;

vector<size\_t> mins(num\_threads, graph.size());

size\_t chunk\_size = orders\_count / num\_threads; //опр размер подзадачи

for (size\_t t = 0; t < num\_threads; ++t) { //цикл запуска потоков

size\_t start = t \* chunk\_size;

size\_t end = start + chunk\_size; //size\_t end = (t == num\_threads - 1) ? orders\_count : start + chunk\_size;

threads.emplace\_back([&graph, &orders, &mins, t, start, end]() { //запуск потока

mins[t] = colorize\_chunk(graph, orders, start, end); //функция потока

});

}

for (auto& thread : threads) { //ожидаем завершения всех потоков

thread.join();

}

min = \*min\_element(mins.begin(), mins.end());

return min;

}

int main() {

ifstream input("input.txt");

ofstream output("output.txt");

graph\_t graph;

size\_t cnt;

input >> cnt;

output << cnt << endl;

for (size\_t i = 0; i < cnt; ++i) {

input >> graph;

auto start = chrono::system\_clock::now();

auto res = LineColorize(graph);

auto stop = chrono::system\_clock::now();

auto time = duration\_cast<milliseconds>(stop - start).count();

output << res << endl;

cout << time << endl;

}

input.seekg(0, ios::beg);

input >> cnt;

for (size\_t i = 0; i < cnt; ++i) {

input >> graph;

auto start = chrono::system\_clock::now();

auto res = ParallelColorize(graph);

auto stop = chrono::system\_clock::now();

auto time = duration\_cast<milliseconds>(stop - start).count();

output << res << endl;

cout << time << endl;

}

return 0;

}