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 MPICH**

Лабораторная работа №4

по курсу «Параллельное программирование»

Вариант 9

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

Киров 2024

1. Задание

Цель лабораторной работы: знакомство с программным интерфейсом MPI, получение навыков реализации параллельных приложений с использованием библиотеки MPICH.

Этапы работы:

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 алгоритм с помощью языка С++ и библиотеки MPICH, используя при этом предлагаемые интерфейсом MPI механизмы и виртуальные топологии (в случае применимости).
4. Показать корректность полученной реализации путем осуществления тестирования на построенном в ходе первой лабораторной работы наборе тестов.
5. Провести доказательную оценку эффективности MPI-реализации алгоритма, в том числе с использованием инструментов профилирования.
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 MPICH были выбраны те же участки, что и при простом распараллеливании. Это было сделано для наиболее точного сравнения результатов тестов.

1. Программная реализация

Листинг программной MPICH-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 | Линейная реализация, мс | Therad, мс | OpenMP, мс | MPICH | Ускорение (Therad) | Ускорение (OpenMP) |
| 5, 7 | 8 | 11 | 19 | 8 | 1,3750 | 2,3750 |
| 75, 1000 | 86 | 53 | 36 | 29 | 1,8275 | 1,2413 |
| 75, 2500 | 156 | 67 | 52 | 54 | 1,2407 | 0,9629 |
| 90, 300 | 83 | 51 | 42 | 28 | 1,8214 | 1,5000 |
| 95, 2500 | 151 | 74 | 52 | 53 | 1,3962 | 0,9811 |
| 150, 5000 | 270 | 116 | 91 | 94 | 1,2340 | 0,9680 |
| 150, 10000 | 420 | 147 | 145 | 162 | 0,9074 | 0,8950 |
| 300, 5000 | 375 | 155 | 145 | 158 | 0,9810 | 0,9177 |
| 300, 10000 | 510 | 220 | 178 | 210 | 1,0476 | 0,8476 |
| 170, 10000 | 427 | 119 | 157 | 180 | 0,6611 | 0,8722 |
| 250, 14000 | 584 | 224 | 158 | 232 | 0,9655 | 0,6810 |
| 250, 17000 | 660 | 218 | 182 | 290 | 0,7517 | 0,6275 |
| 300, 34000 | 1320 | 341 | 336 | 402 | 0,8482 | 0,8358 |
| 400, 40000 | 1452 | 422 | 411 | 458 | 0,9213 | 0,8973 |
| Среднее | Среднее | 1,1413 | 1,0430 |
| Максимальное | Максимальное | 1,8275 | 2,3750 |
| Минимальное | Минимальное | 0,6611 | 0,6275 |

Исходя из результатов тестирования можно сказать, что MPICH реализация данного алгоритма имеет значительно более высокое быстродействие чем линейная реализация. Также в среднем немного выигрывает у OpenMP и Therad реализаций. Однако, выигрыш, в основном, наблюдается на небольшом объеме данных, при большем объеме MPICH версия начинает уступать другим параллельным алгоритмам. Это связанно с необходимостью обмена данными между разделенной памятью процессов.

**Вывод**: В ходе выполнения лабораторной работы была реализована многопоточная версия жадного алгоритма закраски графа на языке С++ с использованием MPICH. MPICH версия в среднем оказалась эффективнее других реализаций алгоритма.

Приложение А

(обязательное)

Листинг программы

#include <vector>

#include <random>

#include <mpi.h>

#include <iostream>

#include <fstream>

#include <iomanip>

#include <numeric>

using namespace std;

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

const int MASTER\_RANK = 0;

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;

}

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

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

}

size\_t colorize(const graph\_t& graph, const vector<size\_t>& order) {

size\_t size = graph.size();

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

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

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 + \*max\_element(colors.begin(), colors.end());

}

size\_t TaskForSlave(const graph\_t& graph, size\_t task\_count) {

vector<size\_t> order(graph.size());

iota(order.begin(), order.end(), 0); //Задаем прямой порядок вершин

vector<vector<size\_t>> perms(task\_count);

mt19937 g(static\_cast<unsigned int>(time(nullptr)));

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

shuffle(order.begin(), order.end(), g);

perms[i] = order;

}

vector<size\_t> results(task\_count, graph.size());

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

results[i] = colorize(graph, perms[i]); //Покраска графа

}

return \*min\_element(results.begin(), results.end());

}

void slave\_routine(int wr, int ws) {

size\_t chunk = 500u / (ws - 1); //определяем объем работы подчиненного

ifstream input("input.txt");

graph\_t graph;

size\_t cnt;

input >> cnt; //получаем из input кол-во тестов

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

input >> graph;

auto start = MPI\_Wtime() \* 1000.0; //начало отсчета

size\_t result = TaskForSlave(graph, chunk); //выполнение задачи

auto stop = MPI\_Wtime() \* 1000.0; //конец отсчета

auto time = static\_cast<size\_t>(stop - start); //вычисление времени выполнения задачи

size\_t buffer[2] = { time, result };

MPI\_Send(&buffer, 2, MPI\_UNSIGNED\_LONG, MASTER\_RANK, static\_cast<int>(i), MPI\_COMM\_WORLD); //Отправка результата в главный процесс

}

}

void master\_routine(int ws) {

ifstream input("input.txt");

ofstream output("output.txt");

size\_t cnt;

input >> cnt; // Получаем из input кол-во тестовых графов

output << cnt << endl; // Выводим в output кол-во тестовых графов

vector<size\_t> results(cnt, numeric\_limits<size\_t>::max()); //вектор результатов тестов

vector<size\_t> times(cnt, 0); //вектор времени тестов

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

size\_t rank\_res[2]; //отдельный результат подчиненного потока

for (int rank = 1; rank < ws; ++rank) {

MPI\_Recv(&rank\_res, 2, MPI\_UNSIGNED\_LONG, rank, static\_cast<int>(i), MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE); //ждем когда подчиненные отработают, и получаем их результат

size\_t time = rank\_res[0];

size\_t res = rank\_res[1];

results[i] = std::min(results[i], res); //запись результата

times[i] += time; //сумма времени работы всех подчиненных?

}

output << setw(4) << results[i] << endl; //вывод в файл результата теста

cout << times[i] / (ws - 1) << endl; //вывод в консоль времени теста

}

}

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv); /\* Инициализируем библиотеку \*/

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank); /\* Процесс узнает свой номер: от 0 до (size-1) \*/

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size); /\* Процесс узнает количество задач в запущенном приложении \*/

if (world\_rank == 0) {

master\_routine(world\_size); // Задача главного процесса 0

}

else {

slave\_routine(world\_rank, world\_size); // Задача подчиненных процессов

}

MPI\_Finalize(); /\* Все процессы завершают выполнение \*/

}