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 | 9 | 1,75 | 2,75 |
| 75, 1000 | 86 | 53 | 36 | 67 | 0,929825 | 0,631579 |
| 75, 2500 | 156 | 67 | 52 | 99 | 0,676768 | 0,525253 |
| 90, 300 | 83 | 51 | 42 | 28 | 1,821429 | 1,5 |
| 95, 2500 | 151 | 74 | 52 | 104 | 0,711538 | 0,5 |
| 150, 5000 | 270 | 116 | 91 | 175 | 0,662857 | 0,52 |
| 150, 10000 | 420 | 147 | 145 | 324 | 0,453704 | 0,447531 |
| 300, 5000 | 375 | 155 | 145 | 219 | 0,707763 | 0,6621 |
| 300, 10000 | 510 | 220 | 178 | 361 | 0,609418 | 0,493075 |
| 170, 10000 | 427 | 119 | 157 | 351 | 0,339031 | 0,447293 |
| 250, 14000 | 584 | 224 | 158 | 503 | 0,445328 | 0,314115 |
| 250, 17000 | 660 | 218 | 182 | 573 | 0,380454 | 0,317627 |
| 300, 34000 | 1320 | 341 | 336 | 819 | 0,416361 | 0,410256 |
| 400, 40000 | 1452 | 422 | 411 | 1038 | 0,406551 | 0,395954 |
| Среднее | Среднее | 0,80793 | 0,851056 |
| Максимальное | Максимальное | 1,75 | 3,75 |
| Минимальное | Минимальное | 0,339031 | 0,314115 |

Исходя из результатов тестирования можно сказать, что MPICH реализация данного алгоритма имеет более высокое быстродействие чем линейная, но уступает OpenMP и Therad реализациям. 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 оказалась эффективнее линейной версии, но уступает в быстродействии OpenMP и Therad реализациям.

Приложение А

(обязательное)

Листинг программы

#include <iostream>

#include <algorithm>

#include <fstream>

#include <chrono>

#include <numeric>

#include <vector>

#include <random>

#include <mpi.h>

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; // вершины

size\_t m; is >> m; // ребра

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); // закрашенные вершины

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

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

for (size\_t v : order) {

for (int i = 0; i < graph[v].size(); ++i) {

size\_t to = graph[v][i];

if (to < size) {

if (colored[to] == 1) {

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

}

}

else {

printf("pizda");

}

}

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

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

colors[v] = c; // раскрашиваем вершину

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

}

return 1 + \*max\_element(colors.begin(), colors.end());

}

size\_t MPIColorize(const graph\_t& graph, int world\_rank, int world\_size) {

size\_t size = graph.size();

size\_t orders\_count = 500;

vector<size\_t> order(size);

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

random\_device rd;

mt19937 g(rd());

size\_t chunk\_size = orders\_count / world\_size;

size\_t local\_min = size;

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

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

local\_min = min(local\_min, colorize(graph, order));

}

size\_t global\_min;

MPI\_Reduce(&local\_min, &global\_min, 1, MPI\_UNSIGNED\_LONG, MPI\_MIN, 0, MPI\_COMM\_WORLD); //Среди всех минимумов находим мин и отправляем в поток 0 в global\_min

return global\_min;

}

void broadcast\_graph(graph\_t& graph, int world\_rank) {

size\_t n, m; // n - кол-во вершин, m - кол-во граней

if (world\_rank == 0) { // Задача главного процесса 0

n = graph.size();

m = 0;

for (const auto& neighbors : graph) {

m += neighbors.size();

}

}

MPI\_Bcast(&n, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD); // Процесс 0 передает n всем остальным процессам

MPI\_Bcast(&m, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD); // Процесс 0 передает m всем остальным процессам

if (world\_rank != 0) { // Задача подчиненных процессов

graph.resize(n); // Задаем графу размер кол-во вершин

}

vector<size\_t> edges;

if (world\_rank == 0) { // Задача главного процесса 0

for (size\_t i = 0; i < n; ++i) { // Считываем все грани из графа

for (size\_t j : graph[i]) {

edges.push\_back(i);

edges.push\_back(j);

}

}

}

else {

edges.resize(2 \* m);

}

MPI\_Bcast(edges.data(), edges.size(), MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD); // Процесс 0 передает массив граней всем процессам

if (world\_rank != 0) { // Задача подчиненных процессов

for (size\_t i = 0; i < m; ++i) { // Загружаем грани в граф

size\_t a = edges[2 \* i];

size\_t b = edges[2 \* i + 1];

graph[a].push\_back(b);

}

}

}

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

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

int world\_rank, world\_size;

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

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

graph\_t graph;

size\_t cnt = 0;

ifstream input("input.txt");

ofstream output("output.txt");

if (world\_rank == 0) { // Задача главного процесса 0

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

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

}

MPI\_Bcast(&cnt, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD); // Процесс 0 передает значение cnt всем остальным процессам

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

if (world\_rank == 0) { // Задача главного процесса 0

input >> graph; // Получаем из input граф

}

broadcast\_graph(graph, world\_rank); // Процесс 0 передает граф всем остальным процессам

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

size\_t res = MPIColorize(graph, world\_rank, world\_size);

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

if (world\_rank == 0) { // Задача главного процесса 0

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

ofstream output("output.txt", ios::app);

output << res << endl;

cout << time << endl;

}

}

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

return 0;

}