МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**ОТЧЕТ ПО ИНДИВИДУАЛЬНОМУ ЗАДАНИЮ № 1**

**по дисциплине  
 «Параллельное и низкоуровневое программирование»**

Выполнил студент группы 25/2                                       А.А. Козин

Направление подготовки  02.03.03  Математическое обеспечение и администрирование информационных систем

Курс    2

Отчет принял доктор физико-математических наук, профессор                                                                                       А.И. Миков

Краснодар

2022 г.

**Задача 1.**

Дан обыкновенный граф: K– (из полного графа из n вершин удалено 3n/4 случайных ребер).

Эффективно распараллелить с использованием потоков thread и других низкоуровневых средств решение задачи «Вычисления диаметра графа».

**Решение**:

В программе функции генерации случайного графа с заданным числом вершин и рёбер, функцию поиска диаметра графа и поиска диаметра графа в многопоточном режиме.

Поиск диаметра осуществляется применением к матрице смежности графа алгоритма Беллмана-Форда с попутным сохранением максимального среди всех минимальных путей в графе, который и будет являться его диаметром.

Для работы в многопоточном режиме создана вспомогательная функция, обрабатывающая заданную часть матрицы. При этом переменная диаметра задана как атомарная, что гарантирует корректный доступ к ней нескольких потоков.

В main будет вызывать обе функции для графов с количеством вершин от 16 до 1024 с шагом n \*= 2. При этом в конце каждого цикла будем сравнивать ответы, полученные в результате разных способов вычислений и выводить сообщение об ошибке, если они различны.

**Код программы:**

#include "profile.h"

#include <iostream>

#include <vector>

#include <random>

#include <atomic>

#include <thread>

using namespace std;

#define INF INT16\_MAX

vector<vector<int32\_t>> gen\_rand\_graph(size\_t vertex\_num, size\_t edges\_num) {

vector<vector<int32\_t>> adj\_matrix(vertex\_num, vector<int32\_t>(vertex\_num, INF));

if (edges\_num <= 0)

return adj\_matrix;

random\_device dev;

mt19937 rng(dev());

uniform\_int\_distribution<mt19937::result\_type> dist(0, 10);

for (int i = 0; i < vertex\_num; ++i) {

adj\_matrix[i][i] = 0;

}

for (int i = 0; i < vertex\_num; ++i) {

for (int j = i; j < vertex\_num; ++j) {

if (i == j)

continue;

else if (!(dist(rng) % 4)) {

adj\_matrix[i][j] = 1;

adj\_matrix[j][i] = 1;

edges\_num--;

}

else {

adj\_matrix[i][j] = INF;

}

if (edges\_num <= 0)

break;

}

if (edges\_num <= 0)

break;

}

return adj\_matrix;

}

int64\_t find\_diameter(vector<vector<int32\_t>> d) {

int64\_t diameter = -1;

for (size\_t k = 0; k < d.size(); ++k)

for (size\_t i = 0; i < d.size(); ++i)

for (size\_t j = 0; j < d.size(); ++j) {

d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

if (d[i][j] != INF && d[i][j] > diameter)

diameter = d[i][j];

}

return diameter;

}

void process\_data\_part(vector<vector<int32\_t>> d, atomic<int64\_t>& diameter, size\_t range\_begin, size\_t ragne\_end) {

for (size\_t k = range\_begin; k < ragne\_end; ++k)

for (size\_t i = 0; i < d.size(); ++i)

for (size\_t j = 0; j < d.size(); ++j) {

d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

if (d[i][j] != INF && d[i][j] > diameter)

diameter = d[i][j];

}

}

int64\_t find\_diameter\_multithreading(const vector<vector<int32\_t>>& adj\_matrix) {

atomic<int64\_t> diameter = -1;

vector<thread> threads;

int part\_size = adj\_matrix.size() / thread::hardware\_concurrency();

vector<int> parallel\_data\_parts\_sizes(thread::hardware\_concurrency(), part\_size);

for (int i = adj\_matrix.size() - part\_size \* thread::hardware\_concurrency(); i > 0; --i) {

parallel\_data\_parts\_sizes[i]++;

}

int t = 0;

for (int i = 0; i < thread::hardware\_concurrency(); ++i) {

threads.emplace\_back(process\_data\_part, ref(adj\_matrix), ref(diameter), t, t + parallel\_data\_parts\_sizes[i]);

t += parallel\_data\_parts\_sizes[i];

}

for (thread& thread : threads) {

thread.join();

}

return diameter;

}

int main() {

size\_t n;

int64\_t singleThreadAns, multithreadingAns;

for (n = 8; n <= 1024; n \*= 2) {

cerr << "Graph with " << n << " verteces and " << n / 4 << " edges" << endl;

auto g = gen\_rand\_graph(n, n / 4);

{

LOG\_DURATION("Single thread");

singleThreadAns = find\_diameter(g);

}

{

LOG\_DURATION("Multithreading");

multithreadingAns = find\_diameter\_multithreading(g);

}

try {

if (singleThreadAns != multithreadingAns) {

throw runtime\_error("DIFFERENT ANSWERS");

}

}

catch (runtime\_error& e) {

cerr << e.what() << endl;

}

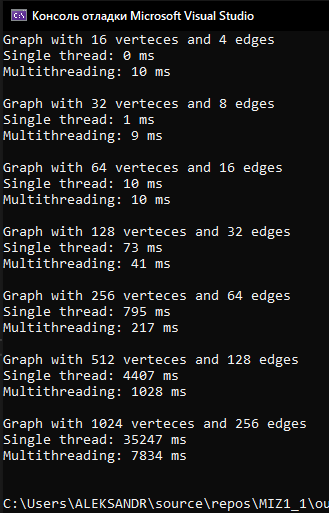
cout << endl;

}

return 0;

}

**Примеры вывода программы с консоль:**



**Вывод:**

Использование параллельных вычислений позволяет уменьшить время расчётов, в данном случае, примерно в 4.5 раза для графа с 1024 вершинами.

**Задача 2.**

В земных недрах геофизическими методами можно отыскивать залежи тех или иных полезных ископаемых. На заданном участке поверхности m×m километров, глубиной 10 км отыскать газоносное месторождение и вычислить его объем.

**Решение:**

Зададим данные по условию параметры размеров земной тверди, а также параметры генерации случайных месторождений в земле.

Для простоты генерации месторождения будут иметь форму куба.

Напишем алгоритм обхода в ширину для трёхмерного пространства. При этом будем подсчитывать количество обработанных клеток, как объём и обходить только клетки с таким же типом ресурса, как и в стартовой, чтобы не начать подсчитывать объём смежного месторождения другого типа.

Функция поиска месторождений в каждой точке поверхности будет просматривать все точки до максимальной заданной глубины, и, если обнаружит там ресурсы, вызовет BFS от этой точки как от стартовой.

BFS отметит всё месторождение как посещённое и вернёт его объём. Если объём оказался больше объёмов найденных ранее месторождений, обновим максимум. Также увеличим счётчик найденных месторождений.

Из функции вернём число газовых месторождений на участке земли и объём самого большого газового месторождения.

**Код программы:**

#include "profile.h"

#include <omp.h>

#include <ppl.h>

#include <iostream>

#include <vector>

#include <random>

#include <atomic>

#include <queue>

#include <algorithm>

using namespace std;

#define INF INT16\_MAX

#ifdef max

#undef max

#endif

#ifdef min

#undef min

#endif

const size\_t m = 1000;

const size\_t depth = 100;

int8\_t min\_num\_of\_fields = 10;

int8\_t max\_num\_of\_fields = 50;

int8\_t min\_size\_of\_field = 10;

int8\_t max\_size\_of\_field = 50;

enum resourse\_type {

NOTHING,

GAS,

OIL,

COAL,

METAL,

};

#define rt resourse\_type

vector<vector<vector<rt>>> ground(m, vector<vector<rt>>(m, vector<rt>(depth)));

void gen\_fields() {

random\_device dev;

mt19937 rng(dev());

uniform\_int\_distribution<mt19937::result\_type> position(0, m - 1);

uniform\_int\_distribution<mt19937::result\_type> depth\_(0, depth - 1);

uniform\_int\_distribution<mt19937::result\_type> num\_of\_fields\_(min\_num\_of\_fields, max\_num\_of\_fields);

uniform\_int\_distribution<mt19937::result\_type> size\_of\_field\_(min\_size\_of\_field, max\_size\_of\_field);

uniform\_int\_distribution<mt19937::result\_type> res\_type\_(0, 4);

cerr << "I'm working" << endl << endl;

int32\_t num\_of\_fields = num\_of\_fields\_(rng);

int32\_t pos\_x, pos\_y, pos\_z, sz1, sz2, sz3, res\_type;

while (num\_of\_fields) {

pos\_x = position(rng);

pos\_y = position(rng);

pos\_z = depth\_(rng);

sz1 = size\_of\_field\_(rng);

sz2 = size\_of\_field\_(rng);

sz3 = size\_of\_field\_(rng);

res\_type = res\_type\_(rng);

for (int32\_t i = pos\_x; i < min(pos\_x + sz1, (int)m); i++)

for (int32\_t j = pos\_y; j < min(pos\_y + sz2, (int)m); j++)

for (int32\_t k = pos\_z; k < min(pos\_z + sz3, (int)depth); k++)

ground[i][j][k] = (rt)res\_type;

num\_of\_fields--;

}

}

int BFS(int stx, int sty, int stz, vector<vector<vector<bool>>>& visited) {

int field\_volume = 0;

visited[stx][sty][stz] = true;

rt this\_field\_type = ground[stx][sty][stz];

queue<int>q;

q.push(stx);

q.push(sty);

q.push(stz);

while (!q.empty()) {

int f1, f2, f3;

f1 = q.front();

q.pop();

f2 = q.front();

q.pop();

f3 = q.front();

q.pop();

for (int8\_t i = -1; i < 2; i++) {

for (int8\_t j = -1; j < 2; j++) {

for (int8\_t k = -1; k < 2; k++) {

int v1, v2, v3;

v1 = f1 + i;

v2 = f2 + j;

v3 = f3 + k;

if ((v1 >= 0 && v1 < m && v2 >= 0 && v2 < m && v3 >= 0 && v3 < depth)

&& !visited[v1][v2][v3] && ground[v1][v2][v3] == this\_field\_type)

{

field\_volume++;

q.push(v1);

q.push(v2);

q.push(v3);

visited[v1][v2][v3] = true;

}

}

}

}

}

return field\_volume;

}

//Найдем количество газовых месторождений и объем самого большого из них

pair<int, int> find\_filds() {

vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(m, vector<bool>(depth, false)));

int32\_t max\_volume = 0, fields\_num = 0;

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

for (size\_t j = 0; j < m; j++)

for (size\_t k = 0; k < depth; k++)

if (ground[i][j][k] == GAS && !visited[i][j][k]) {

max\_volume = max(BFS(i, j, k, visited), max\_volume);

fields\_num++;

}

return { fields\_num, max\_volume };

}

pair<int, int> find\_filds\_omp() {

vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(m, vector<bool>(depth, false)));

int32\_t max\_volume = 0, fields\_num = 0;

#pragma omp parallel for shared(visited)

for (int64\_t i = 0; i < m; i++)

for (size\_t j = 0; j < m; j++)

for (size\_t k = 0; k < depth; k++)

if (ground[i][j][k] == GAS && !visited[i][j][k]) {

max\_volume = max(BFS(i, j, k, visited), max\_volume);

fields\_num++;

}

return { fields\_num, max\_volume };

}

pair<int, int> find\_filds\_ppl() {

vector<vector<vector<bool>>> visited(m, vector<vector<bool>>(m, vector<bool>(depth, false)));

int32\_t max\_volume = 0, fields\_num = 0;

concurrency::parallel\_for((size\_t)0, m, [&](size\_t i) {

for (size\_t j = 0; j < m; j++)

for (size\_t k = 0; k < depth; k++)

if (ground[i][j][k] == GAS && !visited[i][j][k]) {

max\_volume = max(BFS(i, j, k, visited), max\_volume);

fields\_num++;

}

});

return { fields\_num, max\_volume };

}

int main() {

gen\_fields();

pair<int, int> answer, answerOMP, answerPPL;

{

LOG\_DURATION("Single core");

answer = find\_filds();

}

{

LOG\_DURATION("OMP");

answerOMP = find\_filds\_omp();

}

{

LOG\_DURATION("PPL");

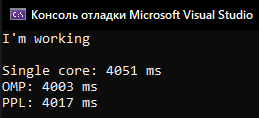
answerPPL = find\_filds\_ppl();

}

return 0;

}

**Вывод программы в консоль:**



Из-за ограниченного ресурса параллелизма алгоритма обхода в ширину мы не получаем линейного ускорения, соразмерного количеству процессоров, выполняющих задачу в многопоточном режиме.

**profile.h** для **задачи 1 и 2**:

#include <chrono>

#include <iostream>

#include <string>

using namespace std;

class LogDuration {

public:

explicit LogDuration(const string& message = "")

: messages(message + ": ")

, start(chrono::steady\_clock::now())

{

}

~LogDuration() {

auto finish = chrono::steady\_clock::now();

auto duration = finish - start;

cerr << messages

<< chrono::duration\_cast<chrono::milliseconds>(duration).count()

<< " ms" << endl;

}

private:

string messages;

chrono::steady\_clock::time\_point start;

};

#define UNIQ\_ID\_IMPL(lineno) \_a\_local\_var\_##lineno

#define UNIQ\_ID(lineno) UNIQ\_ID\_IMPL(lineno)

#define LOG\_DURATION(messages) \

LogDuration UNIQ\_ID(\_\_LINE\_\_){messages};