МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Лабораторная работа № 3 по курсу «Параллельное программирование»

Выполнил студент группы ИВТ-31 _____/Седов М. Д./
Проверил доцент кафедры ЭВМ _____/Долженкова М. Л./

Киров 2020

1 Задание

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

1. Изучить основные принципы создания приложений с использованием библиотеки OpenMP, рассмотреть базовый набор директив компилятора.
2. Выделить в полученной в ходе первой лабораторной работы реализации алгоритма фрагменты кода, выполнение которых может быть разнесено на несколько процессорных ядер.
3. Реализовать многопоточную версию алгоритма с помощью языка C++ и библиотеки OpenMP, использую при этом необходимые примитивы синхронизации.
4. Показать корректность полученной реализации путем осуществления на построенном в ходе первой лабораторной работы наборе тестов.
5. Провести доказательную оценку эффективность OpenMP-реализации алгоритма.

2 Выделение областей для распараллеливания

Для распараллеливания были использованы parallel_section и section в основном цикле поиска возможного хода. Это позволяет выполнять поиск сразу по 4 сторонам параллельно.

3 Тестирование

Тестирование проводилось на ЭВМ под управлением 64-разрядной ОС Linux, с 12 ГБ оперативной памяти, с процессором Intel Core i5 6200U с частотой 2.3 ГГц (4 логических и 2 физических ядра).

Количество строк и столбцов каждой матрицы пятнашек и результаты тестирования приведены в таблице 1.

Таблица 1 – Результаты тестирования

Строки и столбцы матрицы	Линейная реализация, мс	Параллельная реализация, мс	OpenMP,мс	Ускорение
2x3	0,0000734	0,00006502	0,00004302	1,71
3x3	0,0011579	0,00015598	0,00017598	6,58
3x4	0,0126833	0,0036612	0,0020612	6,15
4x4	0,112261	0,008361	0,008197	13,7
4x5	2,50988	0,216	0,204	12,3
5x5	61,3944	4,82699	4,5	13,64
			Среднее	9,01
			Максимальное	13,7
			Минимальное	1,71

4 Вывод

В ходе выполнения лабораторной работы был изучен стандарт OpenMP, его применение и директивы. На основе знаний, полученных в ходе лекционного материала по OpenMP, был разработан параллельный алгоритм поиска разрешающей последовательности в пятнашках. Параллельный алгоритм, реализованный с помощью OpenMP, оказался быстрее алгоритма, реализованного с помощью потоков стандартной библиотеки C++. В среднем, было достигнуто 9-кратное ускорение по сравнению с линейной реализацией.

Приложение А
(обязательное)
Листинг программной реализации

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <vector>
#include "Map.h"
#include "BinTree.h"
#include "State.h"
#include <ctime>
#include <thread>
#include <mutex>
#include <omp.h>

#define TESTS 1
using namespace std;

bool check(Map*);
void printMap(Map*);

Map* generateMap(int lines, int cols) {
int len = lines * cols;
Map* map = new Map(lines, cols);

for (int i = 0; i < len; ++i)
{
map->map[i] = i + 1;
}
map->map[len - 1] = 0;
int i = 0;
int shift_pos;
srand(time(NULL));
while (i <= len * 20) {
int zero = map->find(0);
shift_pos = rand() % 4;
switch (shift_pos) {
case 0:
	if (zero / map->getCols() != 0) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
case 1:
	if (zero % map->getCols() != map->getCols() - 1) {
		map = map->shift(shift_pos);
		i++;
	}
	continue;
```

```
case 2:
		if (zero / map->getCols() != map->getLines() - 1) {
			map = map->shift(shift_pos);
			i++;
		}
		continue;
case 3:
		if (zero % map->getCols() != 0) {
			map = map->shift(shift_pos);
			i++;
		}
		continue;
}
}
return map;
}

void printMap(Map* map) {
cout << endl;
for (int i = 0; i < map->lines; ++i) {
for (int j = 0; j < map->cols; ++j) {
		cout << map->map[i*map->cols + j] << '\t';
}
cout << endl;
}
cout << endl;
}

std::mutex flag_mutex;
bool flag_solution = false;

vector<State*> resultP2;
mutex resultP2_mutex;

vector<State*> thread_func2(Map* map, State* min, BinTree*
close, BinTree* open) {
vector<State*> lol;
omp_set_num_threads(6);
for (; min->getCost() != 0; min = open->min(), close->add(min),
open->del(min))
{
int zero = min->getMap()->find(0);
State* tmp_states[4] = { 0, 0, 0, 0 };

#pragma omp parallel sections
{
#pragma omp section
		{
			if (zero / map->getCols() != 0) {
				State* s = new State(min->getMap()-
>shift(0), min);
				if ((open->find(s) == NULL) &&
(close->find(s) == NULL)) {
```

```
                                        tmp_states[0] = s;
                                }
                        }
                }
#pragma omp section
                {
                        if (zero % map->getCols() != map->getCols() -
1) {
                                State* s = new State(min->getMap()-
>shift(1), min);
                                if ((open->find(s) == NULL) &&
(close->find(s) == NULL)) {
                                        tmp_states[1] = s;
                                }
                        }
                }
#pragma omp section
                {
                        if (zero / map->getCols() != map->getLines() -
1) {
                                State* s = new State(min->getMap()-
>shift(2), min);
                                if ((open->find(s) == NULL) &&
(close->find(s) == NULL)) {
                                        tmp_states[2] = s;
                                }
                        }
                }
#pragma omp section
                {
                        if (zero % map->getCols() != 0) {
                                State* s = new State(min->getMap()-
>shift(3), min);
                                if ((close->find(s) == NULL) &&
(open->find(s) == NULL)) {
                                        tmp_states[3] = s;
                                }
                        }
                }
        }
        for (int i = 0; i < 4; i++) {
                if (tmp_states[i]) {
                        open->add(tmp_states[i]);
                }
        }
        flag_mutex.lock();
        if (flag_solution == true)
        {
                flag_mutex.unlock();
                return lol;
        }
        flag_mutex.unlock();
```

```
}
flag_mutex.lock();
flag_solution = true;
flag_mutex.unlock();
State* s = min;
vector <State*> solution;

do
{
solution.push_back(s);
s = s->getParent();
} while (s != NULL);

resultP2_mutex.lock();
resultP2 = solution;
resultP2_mutex.unlock();

return lol;
}

vector<State*> aPar2(Map* map) {
BinTree* open = new BinTree();
BinTree* close = new BinTree(new State(map, NULL));
State* min = close->min();

vector<thread> threads;
vector<BinTree*> open_branch;
vector<BinTree*> close_branch;

int zero = min->getMap()->find(0);
int index = 0;

if (zero / map->getCols() != 0) {
open_branch.emplace_back(new BinTree(new State(min-
>getMap()->shift(0), NULL)));
close_branch.emplace_back(new BinTree());
}

if (zero % map->getCols() != map->getCols() - 1) {
open_branch.emplace_back(new BinTree(new State(min-
>getMap()->shift(1), NULL)));
close_branch.emplace_back(new BinTree());
}

if (zero / map->getCols() != map->getLines() - 1) {
open_branch.emplace_back(new BinTree(new State(min-
>getMap()->shift(2), NULL)));
close_branch.emplace_back(new BinTree());
}

if (zero % map->getCols() != 0) {
open_branch.emplace_back(new BinTree(new State(min-
>getMap()->shift(3), NULL)));
```

```
close_branch.emplace_back(new BinTree());
}

for (int i = 0; abs(i) < 1; i++) {
threads.emplace_back(thread_func2, open_branch[i]->min()-
>getMap(), open_branch[i]->min(), close_branch[i],
open_branch[i]);
}

for (auto &thread_ : threads) {
thread_.join();
}

return resultP2;

}

int main(int argc, char const *argv[]) {
int lines, cols;
Map* map;

std::cout << "Enter field sizes: " << endl;
cin >> lines >> cols;

double tParNew = 0;
vector<State*> ans;

for (int i = 0; i < TESTS; i++) {
srand(i);
map = generateMap(lines, cols);

cout << "\n" << "-----------------------------------------------------";
cout << "\n" << "Case #" << i + 1 << ": ";
printMap(map);

clock_t time = clock();

time = clock();
ans = aPar2(map);
time = clock() - time;

cout << "\n" << "Time of NEW PARALLEL= " << (double)time /
CLOCKS_PER_SEC;
printMap(ans[0]->getMap());
tParNew += (double)time / CLOCKS_PER_SEC;
}

cout << "\n" << "--------------" << endl;
cout << "Average time NEW PARALLEL = " << tParNew /
TESTS << endl;
cout << "--------------" << endl;

system("pause");
```

```
return 0;
}
```