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 № 2 по курсу «Параллельное программирование»

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

Киров 2020

# 1 Задание

Изучить средства работы с потоками операционной системы, получить навыки реализации многопоточных приложений.

1. Выделить в полученной в ходе первой лабораторной работы реализации поиска разрешающей последовательности ходов в пятнашках произвольного размера фрагменты кода, выполнение которых может быть распределено на несколько процессорных ядер.
2. Реализовать многопоточную версию алгоритма с помощью языка C++ и потоков стандартной библиотеки C++, использую при этом необходимые примитивы синхронизации.
3. Показать корректность полученной реализации, запустив ее на наборе тестов, построенных в ходе первой лабораторной.
4. Провести доказательную оценку эффективности многопоточной реализации алгоритма.

# 2 Метод распараллеливания алгоритма

Многопоточная реализация алгоритма предполагает нахождение из начального состояния всех соседних состояний. Данные состояния становятся начальными для каждого из N потоков, каждый из которых реализует работу последовательного алгоритма из лабораторной работы №1. Таким образом, обработка ветвей дерева состояний игрового поля производится параллельно. Как только один из потоков найдет разрешающую последовательность ходов, все потоки завершают свою работу и выдается ответ.

Листинг многопоточной реализации алгоритма на C++ приведен в приложении А.

# 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 | Ускорение |
|---|---|---|---|
| 2x3 | 0,0000734 | 0,00006502 | 1,13 |
| 3x3 | 0,0011579 | 0,00015598 | 7,42 |
| 3x4 | 0,0126833 | 0,0036612 | 3,46 |
| 4x4 | 0,112261 | 0,008361 | 13,43 |
| 4x5 | 2,50988 | 0,216 | 11,62 |
| 5x5 | 61,3944 | 4,82699 | 12,72 |
| | | Среднее | 8,3 |
| | | Максимальное | 13,43 |
| | | Минимальное | 1,13 |

## 4 Вывод

В ходе лабораторной работы была разработана многопоточная версия алгоритма поиска разрешающей последовательности ходов в пятнашках произвольной размерности с использованием потоков стандартной библиотеки C++. Многопоточный алгоритм оказался быстрее однопоточного на всех тестовых входных данных; в среднем ускорение составило 9,7 раза. Исходя из этого можно предположить, что многопоточная реализация будет быстрее при любых входных данных.

# 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>

#define TESTS 50
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(0));
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_func(Map* map, State* min, BinTree*
close, BinTree* open) {
vector<State*> lol;

for (; min->getCost() != 0; min = open->min(), close->add(min),
open->del(min))
{
int zero = min->getMap()->find(0);

if (zero / map->getCols() != 0) {
		State* s = new State(min->getMap()->shift(0), min);
		if ((open->find(s) == NULL) && (close->find(s) ==
NULL)) {
			open->add(s);
		}
}

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)) {
			open->add(s);
		}
}

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)) {
			open->add(s);
		}
}

if (zero % map->getCols() != 0) {
		State* s = new State(min->getMap()->shift(3), min);
		if ((close->find(s) == NULL) && (open->find(s) ==
NULL)) {
			open->add(s);
		}
}
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*> aPar(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) < open_branch.size()-1; i++) {
threads.emplace_back(thread_func, 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;

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

```
double tP = 0;
double tPar1 = 0;
double tPar2 = 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();
ans = aPar(map);
time = clock() - time;

for(int i = ans.size() - 1; i >= 0; i--)
        printMap(ans[i]->getMap());

cout << "\n" << "Answer Par(Time = " << (double)time /
CLOCKS_PER_SEC << "): ";
printMap(ans[0]->getMap());
tPar2 += (double)time / CLOCKS_PER_SEC;
cout << "-------------------------------------------------------";
}

cout << "\n" << "*********************************" << endl;
cout << "* Average time " << tPar2 / TESTS << " *" << endl;
cout << "*********************************" << endl;

system("pause");
return 0;
}}
```