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 № 4 по курсу «Параллельное программирование»

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

Киров 2020

1. Задание

Знакомство с программным интерфейсом MPI, получение навыков реализации параллельных приложений с использованием библиотеки MPICH.

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 многопоточную версию алгоритма с помощью языка C++ и библиотеки MPICH, используя при этом предлагаемые интерфейсом MPI механизмы и виртуальные топологии (в случае при 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.

2. Словесное описание процесса выделения распараллеливаемых фрагментов

В реализованном алгоритме есть лишь одна процедура, распараллеливание которой может принести выигрыш в скорости нахождения решения поставленной задачи. Данный участок кода отвечает за порождение новых состояний игрового поля за счет сдвига пустой фишки во всех возможных направлениях.

Таким образом, основной процесс отвечает за взаимодействие с пользователем через консоль, а непосредственно решение задачи выполняется несколькими процессами. Количество порождаемых состояний может варьироваться от 2 до 4 включительно, в зависимости от местоположения пустой фишки.

Целесообразным способом распараллеливания является одновременное выполнение сдвига фишка во всех возможных направлениях и запись новых порожденных состояний в дерево.

Каждый процесс порождает новое состояние из изначального путем сдвига пустой фишки в одну из сторон, а затем в каждом процессе происходит поиск решения задачи на основе порожденного состояния.

Как только в одном из процессов было найдено решение, об этом информируются другие процессы и поиск решения завершается.

Исходное состояние сгенерированного игрового поля передается другим процессам из основного с помощью функции MPI_Bcast.

Процессы сигнализируют о нахождении ответа с помощью функций асинхронной передачи MPI_Isend, приём с помощью функции асинхронного приёма MPI_Irecv. Для синхронизации процессов используются функция MPI_Barrier и механизм сообщений.

3. Схема взаимодействия процессов

На рисунке 1 представлена схема, иллюстрирующая взаимодействие процессов.

Рисунок 1 – Схема взаимодействия процеесов

4. 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, с	Параллельная реализация, с	OpenMP,с	MPI, с
2x3	0,0000734	0,00006502	0,00004302	0,0000420
3x3	0,0011579	0,00015598	0,00017598	0,0001730
3x4	0,0126833	0,0036612	0,0020612	0,0020342
4x4	0,112261	0,008361	0,008197	0,0079215
4x5	2,50988	0,216	0,204	0,1723
5x5	61,3944	4,82699	4,5	4,254

5 Вывод

В ходе выполнения лабораторной работы были изучены принципы создания приложения с использованием MPI. Были рассмотрены механизмы передачи сообщений между процессами. Были выделены участки кода, выполнение которых может быть разнесено на несколько процессоров. На их основе был разработан вариант параллельного алгоритма с использованием MPI на языке C++. Разработанный алгоритм хорошо показал себя во время тестирования. На всех размерностях исходного игрового поля он превосходит последовательную версию алгоритма; относительно многопоточной реализации, и реализации на основе openmp, алгоритм показал себя с улучшенным временем работы.

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;
}
```