Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Поиск кратчайших путей из одной вершины (алгоритм Дейкстры)»**

**Выполнил:**

студент группы 381606-1

Ситкин Д.А.

**Проверил:**

к.т.н,

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 2](#_Toc531367741)

[Метод решения 3](#_Toc531367742)

[Схема распараллеливания 4](#_Toc531367743)

[Описание программной реализации 5](#_Toc531367744)

[Подтверждение корректности 6](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 8](#_Toc531367747)

[Приложение 9](#_Toc531367748)

# Постановка задачи

Реализовать параллельный поиск кратчайших путей из одной вершины (алгоритм Дейкстры).  
Изначально мы имеем матрицу смежности и стартовую вершину.  
Результатом работы программы будет массив кратчайших путей.

# Метод решения

Пусть, есть граф G = (V, E),где V – множество вершин, а список ребер графа задается множеством E.

Граф будем представлять в виде матрицы смежности.

Матрица смежности

Где

Каждой вершине из ***V*** сопоставим метку — минимальное известное расстояние от этой вершины до ***a***. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

**Инициализация**. Метка самой вершины ***a*** полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от ***a*** до других вершин пока неизвестны. Все вершины графа помечаются как не посещённые.

**Шаг алгоритма**. Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина ***u***, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых ***u*** является предпоследним пунктом. Вершины, в которые ведут рёбра из ***u***, назовём *соседями* этой вершины. Для каждого соседа вершины ***u***, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки ***u*** и длины ребра, соединяющего ***u*** с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину ***u*** как посещённую и повторим шаг алгоритма.

# Схема распараллеливания

У нас будет один управляющий процессор (процессор 0), который будет разбивать матрицы смежности графа на части и пересылать их на соответствующие процессоры, выбирать стартовую вершину и рассылать ее всем процессорам, инициализировать и рассылать массив кратчайших путей от стартовой вершины до всех остальных вершин графа. Также управляющий процессор будет принимать вершины и метки от функциональных процессов, обновлять результирующий массив меток и класть принятые вершины в очередь.

Повторение итераций до тех пор, пока не посетим все вершины.

Функциональные процессоры будут принимать от процессора 0 соответствующего блока матрицы смежности, стартовые вершины и части массива кратчайших путей.

Далее будет рассматриваться новая длина пути, которая равна сумме метки и длина ребра, соединяющего с вершиной из переданной части. Это повторяется для каждой вершины.

Если полученная длина меньше метки вершины, то оправляем новую метку и саму вершину управляющему процессу.

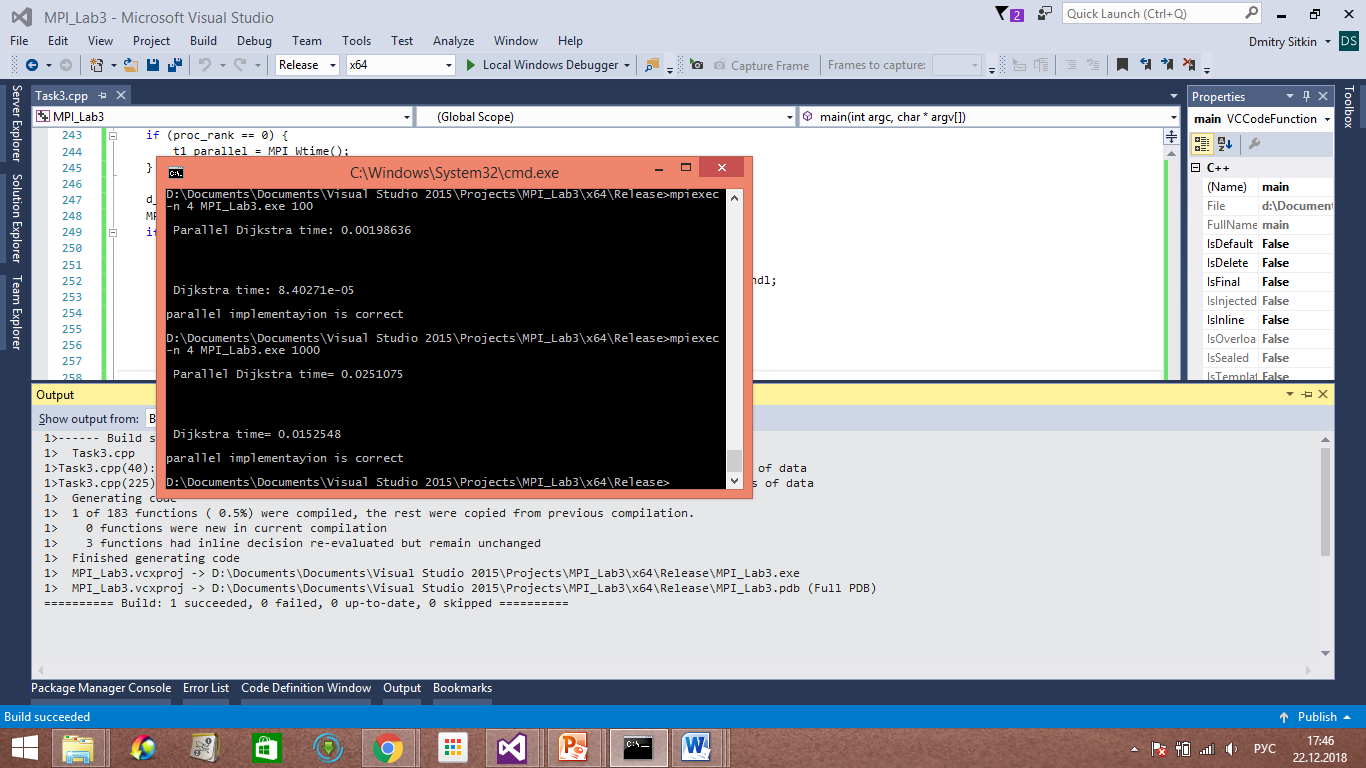
# Описание программной реализации

**Руководство пользователя**

Для запуска программы откройте консоль в директории с приложением, и введите:

**mpiexec –n N program.exe V**

Где N- количество процессов, V-количество вершин в графе



**Руководство программиста**

В программе реализовано две основные функции.

* **int\* dijkstra(int\* graph, int start, int count\_vertex)** – функция реализующая последовательный алгоритм Дейкстры. На вход поступает граф, представленный матрицей смежности, а также стартовая вершина и количество вершин. Возвращает массив кратчайших расстояний от стартовой вершины до всех остальных вершин графа.
* **int\* parallel\_dijkstra(int\* graph, int start, int count\_vertex)** – функция реализующая параллельный алгоритм Дейкстры. На вход также поступает граф, представленный матрицей смежности, стартовая вершина и количество вершин. Возвращает массив кратчайших расстояний от стартовой вершины до всех остальных вершин графа.

Код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

# Подтверждение корректности

Для подтверждения корректности в программе были использованы графы, для которых несложно найти кратчайшие расстояния, например:

Цикл Звезда

Путь

# Результаты экспериментов

По данным экспериментов видно, что последовательный алгоритм работает быстрее параллельного. Это объясняется тем, что основное время работы параллельного алгоритма составляет распределение матрицы смежности между процессами, и многочисленные пересылки между нулевым и функциональными процессами.

|  |  |  |  |
| --- | --- | --- | --- |
| Точек | 200 | 500 | 1000 |
| Последовательно | 0,00027 | 0,0017 | 0,015 |
| 2 процесса | 0,0021 | 0,007 | 0,032 |
| 4 процесса | 0,0036 | 0,0089 | 0,024 |

# Заключение

В ходе выполнения лабораторной работы, я изучил алгоритм поиска кратчайших путей (алгоритм Дейкстры), а также научился из последовательных алгоритмов делать параллельные, правда это не всегда необходимо, потому что в нашем случае последовательный алгоритм быстрее параллельного.

# Приложение

#include <mpi.h>

#include <cstdlib>

#include <time.h>

#include <Windows.h>

#include <queue>

#include <iostream>

#include <string>

#define INFINITI 10000000

#define WEIGHT 5

using namespace std;

int\* init\_graph(int countEdge, int countVertex) {

int rank, procNum;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &procNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int\* one\_graph = new int[countVertex \* countVertex];

int\*\* graph = new int\*[countVertex];

for (int i = 0; i < countVertex; i++) {

graph[i] = new int[countVertex];

}

srand(time(NULL));

for (int i = 0; i < countVertex; i++) {

for (int j = 0; j < countVertex; j++) {

if (i == j) {

graph[i][j] = 0;

}

else {

graph[i][j] = rand() % 3;

if (graph[i][j] == 0) {

graph[i][j] = INFINITI;

}

}

//cout << G[i][j] << " ";

}

//cout << endl;

}

for (int i = 0, t = 0; i < countVertex; i++) {

for (int j = 0; j < countVertex; j++) {

one\_graph[t] = graph[j][i];

t++;

}

}

return one\_graph;

}

int\* init\_d(int size) {

int\* d = new int[size];

for (int i = 0; i < size; i++) {

d[i] = INFINITI;

}

return d;

}

int\* dijkstra(int\* graph, int start, int count\_vertex) {

int\* d = init\_d(count\_vertex);

d[start] = 0;

priority\_queue<pair<int, int>> queue;

queue.push(make\_pair(0, start));

while (!queue.empty()) {

int v = queue.top().second, cur\_d = queue.top().first;

queue.pop();

if (cur\_d > d[v]) continue;

for (int i = 0; i < count\_vertex; ++i) {

int to = i, len = graph[i \* count\_vertex + v];

if (d[v] + len < d[to]) {

d[to] = d[v] + len;

queue.push(make\_pair(d[to], to));

}

}

}

return d;

}

int\* parallel\_dijkstra(int\* graph, int start, int count\_vertex) {

int\* d = nullptr;

int rank = 0, proc\_num = 0;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int\* dist\_d = new int[proc\_num];

int\* dist\_graph = new int[proc\_num];

int\* count\_element\_d = new int[proc\_num];

int\* count\_element\_graph = new int[proc\_num];

int part\_size = count\_vertex / (proc\_num - 1);

count\_element\_d[0] = 0; //send to 0

dist\_d[0] = 0;

count\_element\_graph[0] = 0;

dist\_graph[0] = 0;

for (int i = 1; i < proc\_num - 1; i++) {

count\_element\_d[i] = part\_size;

dist\_d[i] = (i - 1) \* part\_size;

count\_element\_graph[i] = count\_vertex \* part\_size;

dist\_graph[i] = (i - 1) \* count\_vertex \* part\_size;

}

dist\_d[proc\_num - 1] = (proc\_num - 2) \* part\_size;

count\_element\_d[proc\_num - 1] = count\_vertex - (proc\_num - 2) \* part\_size;

dist\_graph[proc\_num - 1] = (proc\_num - 2) \* (count\_vertex \* part\_size);

count\_element\_graph[proc\_num - 1] = count\_vertex \* count\_vertex - (proc\_num - 2) \* (count\_vertex\*part\_size);

int\* part\_graph = new int[count\_element\_graph[rank]];

int\* part\_d = new int[count\_element\_d[rank]];

int flag = 1;

int flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank = 1;

pair<int, int> current\_vertex;

pair<int, int> current\_vertex\_with\_current\_min\_destation;

MPI\_Status st;

if (rank == 0) {

MPI\_Scatterv(graph, count\_element\_graph, dist\_graph, MPI\_INT, part\_graph, count\_element\_graph[rank], MPI\_INT, 0, MPI\_COMM\_WORLD); //send graph

d = init\_d(count\_vertex);

d[start] = 0;

priority\_queue<pair<int, int>> queue;

queue.push(make\_pair(0, start));

int flag\_rank\_0 = 1;

while (!queue.empty()) {

int v = queue.top().second;

int dest = queue.top().first;

queue.pop();

if (dest > d[v])

continue;

MPI\_Bcast(&flag, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

current\_vertex = make\_pair(dest, v);

MPI\_Bcast(&current\_vertex, 1, MPI\_2INT, 0, MPI\_COMM\_WORLD); //send vertex and dist

MPI\_Scatterv(d, count\_element\_d, dist\_d, MPI\_INT, part\_d, count\_element\_d[rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

for (int i = 1; i < proc\_num; i++) {

flag\_rank\_0 = 1;

while (flag\_rank\_0 != 0) {

MPI\_Recv(&flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, &st);

flag\_rank\_0 = flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank;

if (flag\_rank\_0 == 0) {

continue;

}

MPI\_Recv(&current\_vertex\_with\_current\_min\_destation, 1, MPI\_2INT, i, 0, MPI\_COMM\_WORLD, &st);

d[current\_vertex\_with\_current\_min\_destation.second] = current\_vertex\_with\_current\_min\_destation.first;

queue.push(make\_pair(current\_vertex\_with\_current\_min\_destation.first, current\_vertex\_with\_current\_min\_destation.second));

}

}

flag\_rank\_0 = 1;

}

flag = 0;

MPI\_Bcast(&flag, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

}

else {

MPI\_Scatterv(graph, count\_element\_graph, dist\_graph, MPI\_INT, part\_graph, count\_element\_graph[rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

while (flag) {

MPI\_Bcast(&flag, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (flag == 0) {

continue;

}

MPI\_Bcast(&current\_vertex, 1, MPI\_2INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(d, count\_element\_d, dist\_d, MPI\_INT, part\_d, count\_element\_d[rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

for (int j = 0; j < count\_element\_d[rank]; j++) {

int to = j, len = part\_graph[j \* count\_vertex + current\_vertex.second];

if (current\_vertex.first + len < part\_d[to]) {

flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank = 1;

MPI\_Send(&flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

current\_vertex\_with\_current\_min\_destation.first = current\_vertex.first + len;

current\_vertex\_with\_current\_min\_destation.second = to + dist\_d[rank];

MPI\_Send(&current\_vertex\_with\_current\_min\_destation, 1, MPI\_2INT, 0, 0, MPI\_COMM\_WORLD);

}

}

flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank = 0;

MPI\_Send(&flag\_finish\_find\_current\_min\_distation\_in\_sigment\_of\_rank, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

}

return d;

}

bool is\_correct\_implementation(int\* d1, int\* d2, int size) {

int count\_mistakes = 0;

for (int i = 0; i < size; i++) {

if (d1[i] != d2[i]) {

count\_mistakes++;

return false;

}

}

if (count\_mistakes == 0) {

return true;

}

}

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

int proc\_num, proc\_rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

const int count\_vertex = stoi(std::string(argv[1]));

srand(time(NULL));

const int count\_edge = (count\_vertex - 1) + rand() % ((count\_vertex \* (count\_vertex - 1)) / 2);

int\* d\_step = nullptr;

int\* d\_parallel = nullptr;

int\* graph = nullptr;

double t1\_step = 0.0, t2\_step = 0.0, t1\_parallel = 0.0, t2\_parallel = 0.0;

int start;

if (proc\_rank == 0) {

start = rand() % (count\_vertex - 1);

}

if (proc\_rank == 0) {

graph = init\_graph(count\_edge, count\_vertex);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

if (proc\_rank == 0) {

t1\_parallel = MPI\_Wtime();

}

d\_parallel = parallel\_dijkstra(graph, start, count\_vertex);

MPI\_Barrier(MPI\_COMM\_WORLD);

if (proc\_rank == 0) {

t2\_parallel = MPI\_Wtime();

cout << endl << " Parallel Dijkstra time= " << t2\_parallel - t1\_parallel << endl << endl;

cout << endl << endl;

t1\_step = MPI\_Wtime();

d\_step = dijkstra(graph, start, count\_vertex);

t2\_step = MPI\_Wtime();

cout << endl << " Dijkstra time= " << t2\_step - t1\_step << endl << endl;

if (is\_correct\_implementation(d\_step, d\_parallel, count\_vertex)) {

cout << "parallel implementayion is correct" << endl;

}

else {

cout << "parallel implementayion isn`t correct" << endl;

}

}

MPI\_Finalize();

}