Министерство образования и науки Российской Федерации

Севастопольский государственный университет

Кафедра ИС

Отчет

По дисциплине: “Теория распределенных систем и параллельных вычислений”

Лабораторная работа №5

“Исследование моделей взаимодействия распределено

выполняющихся процессов”

Выполнил:

ст.гр. ИС/б-17-2

Долженко И.А.

Проверил:

Дрозин А.Ю.

Севастополь

2020

1 ЦЕЛЬ РАБОТЫ

Исследовать алгоритмическое построение методов взаимодействия распределено выполняющихся процессов.

2 ПОСТАНОВКА ЗАДАЧИ

**Вариант №1.** Осуществить построение топологии кластера требуемого вида (рис. 1); выполнить широковещательную рассылку вводимого с клавиатуры сообщения от узла S на все остальные узлы. На узле, инициирующем рассылку, выводить (в виде матрицы) топологию и остовное дерево, на остальных хостах кластера после получения сообщения выводить номер хоста и сам текст сообщения.



Рисунок 1 – Схема каналов взаимодействия процессов в кластере

3 КОД ПРОГРАММЫ

#include <iostream>

#include "mpi.h"

using namespace std;

const int send\_data\_tag = 99;

int main(int argc, char\*\* argv)

{

int rank;

MPI\_Status status;

MPI\_Comm new\_comm;

int processes;

MPI\_Init(&argc, &argv);

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

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

// Проверка на то, что кол-во процессов равно 5

// если нет, то нулевой процесс выводит ошибку и завершает программу

if (processes != 5 && rank == 0)

{

cout << "Procces count <> 5";

(void)getchar();

return 0;

}

/\*

Построение топологии сети, представленной ввиде графа с узлами

Узлы - процессы, ребра - каналы передачи данных

\*/

// Массив, описывающий степени узлов

int index[] = { 2, 5, 7, 10, 12 };

// Массив, описывающий ребра графа

int edges[] = {

1, 3,

0, 2, 4,

1, 3,

0, 2, 4,

1, 3

};

// Сообщение, передаваемое между узлами

int message[300];

int\* temp = new int[300];

// Создание топологии графа

MPI\_Graph\_create(MPI\_COMM\_WORLD, processes, index, edges, false, &new\_comm);

// Узнаем количество соседей текущего процесса в топологии

// Необходимо для того, чтобы знать кол-во узлов, на которые

// Надо посылать "Зонд" и "Эхо"-сообщения при построении топологии

int countNeighbors;

MPI\_Graph\_neighbors\_count(new\_comm, rank, &countNeighbors);

// Получаем список номеров процессов-соседей для текущего процесса

int\* neighbors = new int[countNeighbors];

MPI\_Graph\_neighbors(new\_comm, rank, countNeighbors, neighbors);

MPI\_Comm\_rank(new\_comm, &rank);

int first;

// Все процессы кроме первого начинают свою работу с того, что ожидают зонда

if (rank != 0)

{

// Ожидание зонда

MPI\_Recv(message, 300, MPI\_INT, MPI\_ANY\_SOURCE, send\_data\_tag, new\_comm, &status);

cout << "Process " << rank << "\t Recv zond from process " << message[1] << endl;

// Запоминаем тот процесс, который первым прислал нам зонд.

// Ему нужно будет вернуть финальное эхо

if (message[0] == 1)

{

first = message[1];

}

}

// message[0] - тип сообщения

// 1 - зонд

// 2 - эхо

// message[1] - ID источника соообщения

message[0] = 1;

message[1] = rank;

// Каждый процесс рассылает всем своим соседям зонды

// в коде соседи, которые ждут, получат зонд

// самым первым зонды разошлет 0й процесс

for (int i = 0; i < countNeighbors; i++)

{

cout << "Process " << rank << "\t Send zond to process " << neighbors[i] << endl;

MPI\_Send(message, 300, MPI\_INT, neighbors[i], send\_data\_tag, new\_comm);

}

int from;

// Кол-во сообщений all = кол-во соседей \* 2

// т.е. каждый из соседей пошлет нам зонд и эхо (пустое эхо)

int all = countNeighbors \* 2;

if (rank != 0)

{

all--;

}

for (int i = 0; i < 300; i++)

{

temp[i] = 0;

}

/\*

В матрице temp (и message) хранится инфа, которая будет передаваться в сообщении,

в элементах с индексами от 2 до 27 (25+2) будет храниться последовательность

строк двумерной матрицы 5\*5 связности кластера (всего 25 элементов матрицы)

Т.е. temp[i][j] = 1, если есть ребро соединяющее вершины i и j, иначе элемент равен 0

\*/

// Записываем строку матрицы топологии для процесса rank

// т.е.каждый из процессов записывает строку топологии, которая

// хранит информацию о том, с какими узлами процессами-соседями этот процесс имеет связь

// эта строка будет занимать соответствующую строку в основной матрице топологии

for (int i = 0; i < countNeighbors; i++)

{

temp[rank \* 5 + neighbors[i] + 2] = 1;

temp[neighbors[i] \* 5 + rank + 2] = 1;

}

// Пока не примем все сообщения от соседей (зонд и эхо)

for (int i = 0; i < all; i++)

{

// Принимаем сообщение от соседа

MPI\_Recv(message, 300, MPI\_INT, MPI\_ANY\_SOURCE, send\_data\_tag, new\_comm, &status);

cout << "Process " << rank << "\t Recv message from process " << message[1];

// Если зонд

if (message[0] == 1)

{

cout << "\t It is zond " << endl;

// Узнаем, от кого пришло сообщение

from = message[1];

message[0] = 2;

message[1] = rank;

// Признак пустого эхо

message[99] = -1;

cout << "Process " << rank << "\t Send empty echo to process " << from << endl;

// Отправляем на этот процесс пустое эхо

MPI\_Send(message, 300, MPI\_INT, from, send\_data\_tag, new\_comm);

}

// Если эхо

else if (message[0] == 2)

{

cout << "\t It is echo " << endl;

// Если мы приняли не пустой эхо ответ

if (message[99] != -1)

{

// Добавляем полученную часть топологии

// т.е. если от процесса пришло не пустое эхо, значит между ними есть связь

// поэтому добавляем его в топологию

for (int i = 2; i < 300; i++)

{

if (message[i] == 1)

{

temp[i] = 1;

}

}

}

}

}

if (rank != 0)

{

cout << "Process " << rank << "\t Send final echo to process " << first << endl;

// Метка о том, что это финальный эхо ответ

temp[99] = 1;

temp[0] = 2;

temp[1] = rank;

// Отправляем на процесс, который первым прислал нам зонд

// финальный эхо-ответ, в котором передадим

// информацию о локальной топологии соответствующего процесса

MPI\_Send(temp, 300, MPI\_INT, first, send\_data\_tag, new\_comm);

}

// По завершению отправки всех финальных эхо ответов на корневом процессе

// все локальные топологии собираются воедино и образуется общая топология

// кластера. Выводим ее на экран

if (rank == 0)

{

(void)getchar();

cout << "Cluster topology:" << endl;

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 5; j++)

{

cout << temp[5 \* i + j + 2] << "\t";

}

cout << endl;

}

cout << endl;

}

/\*

Получаем из топологии остов дерева.

На каждом шаге алгоритма добавляем по одному ребру

т.о., чтобы одна вершина ребра была среди уже добавленных,

а вторая - нет. Эту вторую вершину, добавляем в список добавленных.

Вершина на первом шаге добавляется произвольно.

\*/

int ostov[5][5];

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 5; j++)

{

ostov[i][j] = 0;

}

}

// Остановное дерево строим на корневом процессе

if (rank == 0)

{

// used[i] = true - узел включен в остов, иначе нет

// по умолчанию все узлы не включены, кроме первого

bool used[5];

for (int i = 0; i < 5; i++)

{

used[i] = false;

}

// добавляем первый узел в остов по умолчанию

used[0] = true;

// пока все узлы не будут добавленны в остов, повторяем итерации

// всего осталось добавить 4 узла

for (int i = 0; i < 4; i++)

{

bool done = false;

// идем по все узлам, уже находящимся в остове

for (int j = 0; j < 5; j++)

{

// проверка на то, что узел находится в остове

if (used[j] == true)

{

// идем по все соседним узлам текущего рассматриваемого узла

for (int k = 0; k < 5; k++)

{

// если узел еще не включен в остов и

// существует канал связи между узлом в остове и текущим

// рассматриваемым узлом, то добавляем узел в остов

if ((used[k] == false) && (temp[5 \* j + k + 2] == 1))

{

ostov[j][k] = 1;

ostov[k][j] = 1;

done = true;

used[k] = true;

break;

}

}

}

// если на текущей итерации был добавлен узел, то переходим на следующую

if (done)

{

break;

}

}

}

// Выводим полученное остовное дерево

cout << endl << "Minimum spanning tree is:" << endl;

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 5; j++)

{

cout << ostov[i][j] << "\t";

}

cout << endl;

}

cout << endl << "Enter the message: " << endl;

}

char s[250];

/\*

Считываем строку. Сообщение будем передавать в массиве temp

temp[0] - длина строки

далее идет temp[0] элементов массива temp, определяющих строку

каждый элемент - очередная буква строки

после строки 25 элементов матрицы temp - остовное дерево

\*/

if (rank == 0)

{

scanf\_s("%s", &s, sizeof(s));

temp[0] = strlen(s);

cout << endl;

for (int i = 0; i < temp[0]; i++)

{

temp[i + 1] = (int)s[i];

}

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 5; j++)

{

temp[temp[0] + 1 + 5 \* i + j] = ostov[i][j];

}

}

}

// Все процессы кроме первого принимают одно сообщение

if (rank != 0)

{

MPI\_Recv(temp, 300, MPI\_INT, MPI\_ANY\_SOURCE, send\_data\_tag, new\_comm, &status);

for (int i = 0; i < temp[0]; i++)

{

s[i] = (char)temp[i + 1];

}

s[temp[0]] = '\0';

cout << "Process " << rank << "\t\tReceived message: " << s << endl;

}

// Весь столбец остовной матрицы, соотв. процессу rank, обнуляется

// Для того, чтобы снова не отсылать ему сообщение, которое он сам разослал

for (int i = 0; i < 5; i++)

{

temp[temp[0] + 1 + 5 \* i + rank] = 0;

}

// Отсылаем сообщения каждому соседу, только если не получили от него сообщение сами

for (int i = 0; i < 5; i++)

{

if (temp[temp[0] + 1 + 5 \* rank + i] == 1)

{

cout << "Process " << rank << " send message to process " << i << endl;

MPI\_Send(temp, 300, MPI\_INT, i, send\_data\_tag, new\_comm);

}

}

if (!rank)

{

cin.ignore();

cin.clear();

(void)getchar();

}

MPI\_Finalize();

return 0;

}

4 РЕЗУЛЬТАТЫ

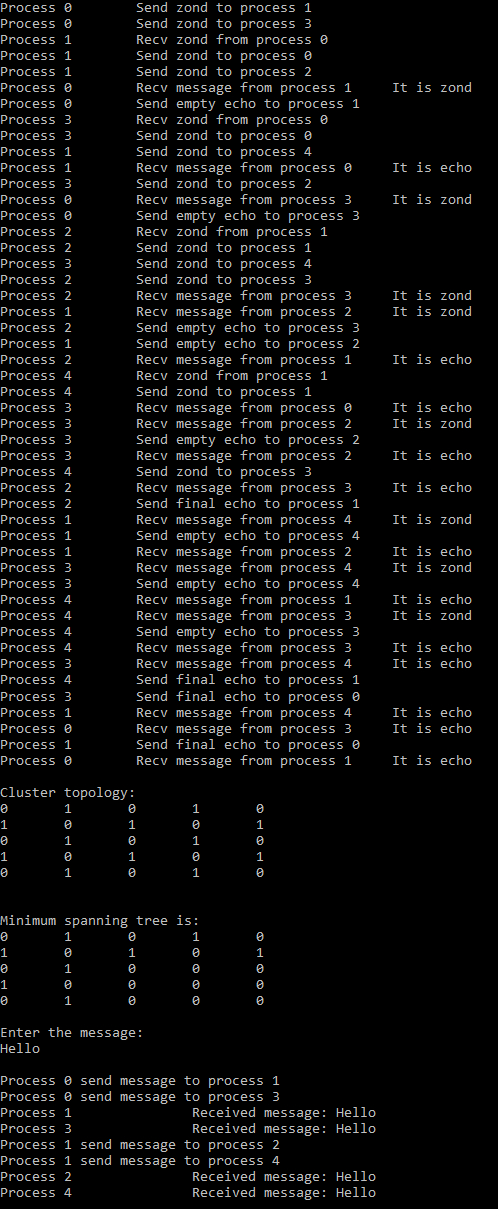


Рисунок 2 – Результаты выполнения программы

Графическое пояснение к полученной топологии представлено на рисунке 3. Номера узлов соответствуют номерам процессов.

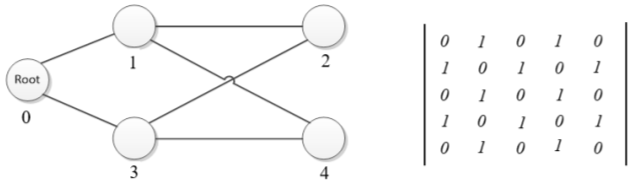


Рисунок 3 – Графическое отображение топологии в виде графа и виде матрицы

Отличие полученного остовного дерева от топологии исходного кластера изображено на рисунке 4.

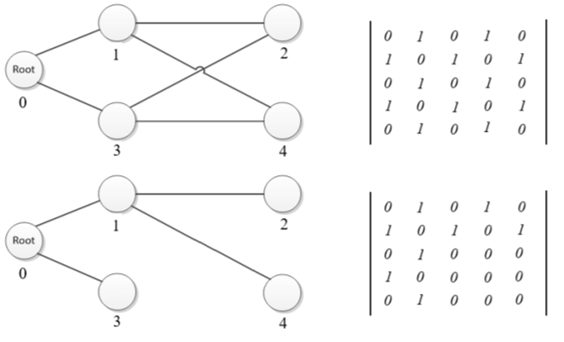


Рисунок 4 – Отличие исходной топологии от остовного дерева

ВЫВОДЫ

В ходе выполнения данной лабораторной работы было исследовано алгоритмическое построение взаимодействий распределено выполняющихся процессов. С помощью построения остовного дерева мы добились отсутствия избыточных повторных передач сообщения, которые возникли бы, если бы мы не использовали информацию об остовном дереве.