МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Севастопольский государственный университет»

кафедра Информационные системы

Бариев Эмин Юсуфович

Институт информационных технологий и управления в технических системах

курс 4 группа ИС/б-16-2

09.03.02 Информационные системы и технологии (уровень бакалавриата)

ОТЧЕТ

по лабораторной работе №3

По дисциплине: «ТРСиПВ»

По теме: «Иследование возможностей формирования виртуальных топологий вычислительных кластеров»

Отметка о зачете \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

(дата)

Руководитель практикума

ст.преподаватель Дрозин А. Ю.

(должность) (подпись) (инициалы,фамилия)

Севастополь 2019

1 ЦЕЛЬ РАБОТЫ

Исследовать возможности, предоставляемые MPI по формированию виртуальных топологий.

2 ПОСТАНОВКА ЗАДАЧИ

Вариант – 1

Необходимо реализовать алгоритм перемножения матриц ленточным способом с распределением столбцов.

3 ХОД РАБОТЫ

3.1 Код программы на языке С++

#include <iostream>

#include <mpi.h>

#include <fstream>

#include <iomanip>

#include <cmath>

#include <cstdlib>

**using** **namespace** std;

**static** MPI\_Comm graph;

**static** MPI\_Comm comm;

**static** **int** processRank;

**static** **int** processCount;

**static** **int** \*\*matrixA, \*\*matrixB, \*\*matrixC, \*row, \*column, \*tmpColumn, \*result;

**int** nextProcess();

**void** initBuffers(**int** \*rowsBuf, **int** \*columnsBuf);

**void** createGraph();

**void** initRowAndColumn(**int** \*rowsBuf, **int** \*columnsBuf, **int** \*row, **int** \*column);

**void** iteration(**int** \*row, **int** \*column, **int** \*result, **int** index);

**void** colsExchange(**int** \*column, **int** \*tmpColumn);

**void** tapeMultiplication(**int** \*row, **int** \*column, **int** \*result, **int** \*tmpColumn);

**void** resultRowsGather();

**void** rootProc();

**void** otherProc();

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

MPI\_Init(&argc, &argv);

comm = MPI\_COMM\_WORLD;

MPI\_Comm\_rank(comm, &processRank);

MPI\_Comm\_size(comm, &processCount);

createGraph();

**if** (processRank == 0) {

rootProc();

}

otherProc();

MPI\_Finalize();

**return** 0;

}

**int** nextProcess() {

**int** rankToNextProcess;

MPI\_Graph\_neighbors(graph, processRank, 1, &rankToNextProcess);

**return** rankToNextProcess;

}

**void** initBuffers(**int** \*rowsBuf, **int** \*columnsBuf) {

**int** k = 0;

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

rowsBuf[k] = matrixA[i][j];

columnsBuf[k++] = matrixB[j][i];

}

}

}

**void** createGraph() {

**int** n = processCount;

**int** \*index = **new** **int**[processCount];

**int** \*edges = **new** **int**[processCount];

**for** (**int** i = 1; i <= processCount; i++) {

index[i - 1] = i;

edges[i - 1] = i % processCount;

}

MPI\_Barrier(comm);

MPI\_Graph\_create(comm, n, index, edges, 0, &graph);

MPI\_Comm\_size(graph, &processCount);

MPI\_Comm\_rank(graph, &processRank);

**delete**[] index;

**delete**[] edges;

}

**void** initRowAndColumn(**int** \*rowsBuf, **int** \*columnsBuf, **int** \*row, **int** \*column) {

MPI\_Barrier(graph);

MPI\_Scatter(rowsBuf, processCount, MPI\_INT, row, processCount, MPI\_INT, 0, graph);

MPI\_Barrier(graph);

MPI\_Scatter(columnsBuf, processCount, MPI\_INT, column, processCount, MPI\_INT, 0, graph);

}

**void** iteration(**int** \*row, **int** \*column, **int** \*result, **int** index) {

result[index] = 0;

**for** (**int** i = 0; i < processCount; result[index] += row[i] \* column[i], i++);

}

**void** colsExchange(**int** \*column, **int** \*tmpColumn) {

MPI\_Status status;

**int** next;

next = !processRank ? processCount - 1 : processRank - 1;

**if** (!(processRank % 2)) {

MPI\_Barrier(graph);

MPI\_Send(column, processCount, MPI\_INT, next, 0, graph);

}

MPI\_Barrier(graph);

MPI\_Recv(tmpColumn, processCount, MPI\_INT, nextProcess(), MPI\_ANY\_TAG, graph, &status);

**if** (processRank % 2) {

MPI\_Barrier(graph);

MPI\_Send(column, processCount, MPI\_INT, next, 0, graph);

}

}

**void** tapeMultiplication(**int** \*row, **int** \*column, **int** \*result, **int** \*tmpColumn) {

**int** index = processRank;

**for** (**int** i = 0; i < processCount; i++) {

iteration(row, column, result, index);

**if** (i != processCount - 1) {

colsExchange(column, tmpColumn);

**for** (**int** j = 0; j < processCount; column[j] = tmpColumn[j], j++);

}

index = (index + 1) % processCount;

}

}

**void** resultRowsGather() {

**int** \*tmpBuf;

**if** (processRank == 0) {

tmpBuf = **new** **int**[processCount \* processCount];

} **else** {

tmpBuf = **new** **int**[1];

}

MPI\_Barrier(graph);

MPI\_Gather(result, processCount, MPI\_INT, tmpBuf, processCount, MPI\_INT, 0, graph);

**if** (processRank == 0) {

**int** k = 0;

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; matrixC[i][j++] = tmpBuf[k++]);

}

}

}

**void** rootProc() {

matrixA = **new** **int** \*[processCount];

matrixB = **new** **int** \*[processCount];

matrixC = **new** **int** \*[processCount];

**for** (**int** i = 0; i < processCount; i++) {

matrixA[i] = **new** **int**[processCount];

matrixB[i] = **new** **int**[processCount];

matrixC[i] = **new** **int**[processCount];

}

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

matrixA[i][j] = rand() % 9;

}

}

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

matrixB[i][j] = rand() % 9;

}

}

result = **new** **int**[processCount];

row = **new** **int**[processCount];

column = **new** **int**[processCount];

tmpColumn = **new** **int**[processCount];

**int** \*rowsBuf = **new** **int**[processCount \* processCount];

**int** \*columnsBuf = **new** **int**[processCount \* processCount];

initBuffers(rowsBuf, columnsBuf);

initRowAndColumn(rowsBuf, columnsBuf, row, column);

tapeMultiplication(row, column, result, tmpColumn);

resultRowsGather();

cout<<"matrixA \* matrixB = matrixC"<<endl;

cout<<endl;

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

cout<<matrixA[i][j]<<" ";

}

cout<<endl;

}

cout << "\t\*" << endl;

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

cout<<matrixB[i][j]<<" ";

}

cout<<endl;

}

cout << "\t=" << endl;

**for** (**int** i = 0; i < processCount; i++) {

**for** (**int** j = 0; j < processCount; j++) {

cout<<matrixC[i][j]<<" ";

}

cout<<endl;

}

**delete** matrixA;

**delete** matrixB;

**delete** matrixC;

**delete** result;

**delete** row;

**delete** column;

**delete** tmpColumn;

**delete**[] rowsBuf;

**delete**[] columnsBuf;

}

**void** otherProc() {

result = **new** **int**[processCount];

row = **new** **int**[processCount];

column = **new** **int**[processCount];

tmpColumn = **new** **int**[processCount];

**int** \*rowsBuf = **new** **int**[1];

**int** \*columnsBuf = **new** **int**[1];

initRowAndColumn(rowsBuf, columnsBuf, row, column);

tapeMultiplication(row, column, result, tmpColumn);

resultRowsGather();

**delete**[] result;

**delete**[] row;

**delete**[] column;

**delete**[] tmpColumn;

**delete**[] rowsBuf;

**delete**[] columnsBuf;

}

3.2 Результаты выполнения программы

**matrixA \* matrixB = matrixC**

**4 7 5 2 4**

**8 6 8 8 1**

**0 8 6 8 0**

**2 7 7 7 1**

**6 1 0 2 4**

**\***

**6 6 0 8 7**

**7 2 5 1 8**

**0 1 1 0 3**

**1 2 1 0 2**

**2 2 4 6 4**

**=**

**83 55 58 63 119**

**100 86 50 76 148**

**64 38 54 8 98**

**70 49 53 29 109**

**53 50 23 73 70**

ВЫВОДЫ

В ходе выполнения лабораторной работы были исследованы возможности, предоставляемые MPI по формированию виртуальных топологий.

Была реализована программа, реализующая алгоритм перемножения матриц ленточным способом с распределением столбцов.