НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

КАФЕДРА АВТОМАТИЗАЦІЇ СИСТЕМ ОБРОБКИ ІНФОРМАЦІЇ ТА УПРАВЛІННЯ

Лабораторна робота №8

з дисципліни **«**Паралельне програмування**»**

Виконав:

студент 3 курсу

ФІОТ гр. ІП-31

Кахерський О.І.

Перевірив:

Корочкін О. В.

Київ – 2016 р.

**Тема:** Програмування для комп’ютерних систем з локальною пам’яттю. Бібліотека MPI.

1. **Технічне завдання:** Розробити програму для розв’язання ПКС із СП (структура на рис. 1) математичної задачі: =

**Бібліотека: MPI.**



Рис.1 Структурна схема ПКС

**Виконання роботи**

**Етап 1.** Побудова паралельного алгоритму

1. =

**Етап 2. Розробка алогоритмів роботи кожного процесу**

i = (1 ,3.. 11)

**Задача Т0**

* 1. Введення МB, MC, МK, a, e;
  2. Упакувати , ,MC, a, ;
  3. Надіслати пакети всім процесам (враховуючи топологію);
  4. Обчислити =
  5. Надіслати процесу Т2 (враховуючи топологію).

**Задача Т2**

1. Отримати упаковані, надіслані T0;
2. Розпакувати дані;
3. Обчислити =
4. Отримати від процесів Т0,T1,Т3-10 (враховуючи топологію).
5. Виведення MA

**Задача Ті**

1. Отримати упаковані, надіслані T0;
2. Розпакувати дані;
3. Обчислити =
4. Надіслати процесу Т2 (враховуючи топологію)

**Етап 4 . Лістинг програми  
import** mpi.Graphcomm;  
**import** mpi.MPI;  
  
**import** java.util.Arrays;  
*/\*  
PP-2  
Lab-8.MPI  
Kobylynskiy D.A.  
 \*/***public class** Main {  
  
 **private static int**[][] *MB*, *MC*, *MK*;  
 **public static int** *a*, *e*, *N*, *H*;  
 **private static int** *p*;  
  
 **public static void** main(String[] args) {  
  
 MPI.*Init*(args);  
  
 **long** startTime = System.*currentTimeMillis*();  
 **int** rank = MPI.*COMM\_WORLD*.Rank();  
 **int** size = MPI.*COMM\_WORLD*.Size();  
  
 System.***out***.println(**"Process "** + rank + **" is started"**);  
  
 *//определяем размерность матриц - передана как последний параметр командной строки  
 N* = Integer.*parseInt*(args[args.**length** - 1]);  
 *H* = *N* / size;  
 *p* = size - 1;  
  
 **int**[] index = {1,3,7,9,13,15,16,17,18,19,20};  
 **int**[] edges = {1,0,2,1,3,4,5,2,6,2,8,9,10,4,7,6,5,5,5};  
 Graphcomm graphcomm = MPI.*COMM\_WORLD*.Create\_graph(index, edges, **false**);  
 *initData*(rank);  
  
 **int**[] indexCount = *getEachCount*(size);  
 **int**[] displacement = *getEachOffset*(size);  
  
 **if** (rank == 0) {DataPackBuilder.*setMetadata*(size, *getEachCount*(size), *getEachOffset*(size));  
 }  
DataPack[] sendPacks;  
 **if** (rank == 0)  
 sendPacks = DataPackBuilder.*packData*(*MB*, *MC*, *MK*, *a*, **true**);  
 **else** sendPacks = **new** DataPack[0];  
  
 **int**[] sendCount = **new int**[size];  
 **int**[] sendDisplacement = **new int**[size];  
  
 **if** (rank == 0) {  
  
 Arrays.*fill*(sendCount, 1);  
 **for** (**int** i = 0; i < sendDisplacement.**length**; i++) {  
 sendDisplacement[i] = i;  
 }  
 }  
  
 **int**[] receiveCount = **new int**[size];  
 receiveCount[0] = 1;  
 receiveCount[*p*] = 1;  
  
 **int**[] receiveDisplacement = **new int**[size];  
 receiveDisplacement[0] = 0;  
 receiveDisplacement[*p*] = 1;  
  
 DataPack[] receivePacks = **new** DataPack[2];  
  
 graphcomm.Alltoallv(sendPacks, 0, sendCount, sendDisplacement, MPI.*OBJECT*,  
 receivePacks, 0, receiveCount, receiveDisplacement, MPI.*OBJECT*);  
  
 *MC* = receivePacks[0].getMatrix2();  
 *MB* = receivePacks[0].getMatrix1();  
 *a* = receivePacks[0].getConstant();*e* = receivePacks[0].getConstant2();  
  
 *MK* = receivePacks[0].getMatrix3();  
 *MA* = MatrixOperations.*addMatrix*(  
 MatrixOperations.*multMatrix*(*MO*, *MB*), *MK*, *a*, *a*);  
  
 **int**[][] result = **new int**[*N*][];  
 graphcomm.Gatherv(*MA*, 0, indexCount[rank], MPI.*OBJECT*, result, 0, indexCount, displacement, MPI.*OBJECT*, 2);  
  
 **f** (rank == 2) {  
 **if** (result.**length** <= 16)  
 System.***out***.println(MatrixOperations.*formattedDeepToString*(result));  
 System.***out***.println(**"Computation time (sec): "** + ((System.*currentTimeMillis*() - startTime) / 1000.0));  
 }  
 System.***out***.println(**"Process "** + rank + **" is finished"**);  
 MPI.*Finalize*();  
 }  
  
 **private static void** initData(**int** rank) {  
 **if** (rank == 0) {  
 *MB* = MatrixOperations.*inputMatrix*(*N*);  
 *MO* = MatrixOperations.*inputMatrix*(*N*);  
 *a* = MatrixOperations.*inputConstant*();  
  
 *MC* = MatrixOperations.*inputMatrix*(*N*);  
 *MK* = MatrixOperations.*inputMatrix*(*N*);  
 *e* = MatrixOperations. *inputConstant*();  
 }  
 }  
  
 **private static int**[] getEachCount(**int** processCount) {  
 **int**[] result = **new int**[processCount];  
 Arrays.*fill*(result, 1, result.**length**, *H*);  
result[0] = *N* - (result.**length** - 1) \* *H*;  
 **return** result;  
 }  
  
 **private static int**[] getEachOffset(**int** processCount) {  
 **int**[] result = **new int**[processCount];  
 result[0] = 0;  
 result[1] = *N* - (result.**length** - 1) \* *H*;  
 **for** (**int** i = 2; i < result.**length**; i++) {  
 result[i] = result[i - 1] + *H*;  
 }  
 **return** result;  
 }  
}

**DataPack.java**

**public class** DataPack **implements** Serializable {  
  
 **private int**[][] **matrix1**;  
  
 **private int**[][] **matrix2**;  
  
  
 **private int**[][] **matrix3**;  
  
 **private int constant**;  
  
 **public int** getConstant2() {  
 **return constant2**;  
 }  
  
 **public void** setConstant2(**int** constant2) {  
 **this**.**constant2** = constant2;  
 }  
  
 **private int constant2**;  
 **public** DataPack() {  
 **matrix1** = **new int**[0][0];  
 **matrix2** = **new int**[0][0];  
 **matrix3** = **new int**[0][0];  
 }  
  
 **public int** getConstant() {  
 **return constant**;  
 }  
  
 **public void** setConstant(**int** constant) {  
 **this**.**constant** = constant;  
 }  
  
  
 **public int**[][] getMatrix2() {  
 **return matrix2**;  
 }  
  
 **public void** setMatrix2(**int**[][] matrix2) {  
 **this**.**matrix2** = matrix2;  
 }  
  
 **public int**[][] getMatrix1() {  
 **return matrix1**;  
 }  
  
 **public void** setMatrix1(**int**[][] matrix1) {  
 **this**.**matrix1** = matrix1;  
 }  
  
 **public int**[][] getMatrix3() {  
 **return matrix3**;  
 }  
  
 **public void** setMatrix3(**int**[][] matrix3) {  
 **this**.**matrix3** = matrix3;  
 }  
  
  
  
}

**DataPackBuilder.java**

**import** java.util.Arrays;  
  
*/\*\*  
 \* Created by Oleh Kakherskyi, student of the KPI, FICT, IP-31 group (olehkakherskiy@gmail.com) on 18.04.2016.  
 \*/***public class** DataPackBuilder {  
  
 **private static** DataPack[] *preparedPacks*;  
  
 **public static int**[] *elementsCount*;  
  
 **public static int**[] *dataOffset*;  
  
 **public static void** setMetadata(**int** processCount, **int**[] elemCount, **int**[] offset) {  
 *preparedPacks* = **new** DataPack[processCount];  
 **for** (**int** i = 0; i < *preparedPacks*.**length**; i++) {  
 *preparedPacks*[i] = **new** DataPack();  
 }  
 *elementsCount* = elemCount;  
 *dataOffset* = offset;  
 }  
  
 **public static** DataPack[] packData(**int**[][] MO, **int**[][] MT, **int**[][] MR, **int** e1, **int** e) {  
 **for** (**int** i = 0; i < *preparedPacks*.**length**; i++) {  
 *preparedPacks*[i].setMatrix1(Arrays.*copyOfRange*(MO, *dataOffset*[i], *dataOffset*[i] + *elementsCount*[i]));  
 *preparedPacks*[i].setMatrix2(MR);  
 *preparedPacks*[i].setMatrix3(Arrays.*copyOfRange*(MT, *dataOffset*[i], *dataOffset*[i] + *elementsCount*[i]));  
 *preparedPacks*[i].setConstant(e);  
 *preparedPacks*[i].setConstant2(e);  
 }  
 **return** *preparedPacks*;  
 }  
}

**MatrixOperations.java**

**import** java.util.Arrays;  
  
*/\*\*  
 \* Created by oleg on 03.10.15.  
 \*/***public class** MatrixOperations {  
  
 */\*\*  
 \* Генерує матрицю, заповнює одиницями  
 \*  
 \** ***@param n*** *розмірність  
 \*/* **public static int**[][] inputMatrix(**int** n) {  
 **int**[][] result = **new int**[n][n];  
 **for** (**int** i = 0; i < n; i++)  
 **for** (**int** j = 0; j < n; j++) {  
 result[i][j] = 1;  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* Генерує вектор, заповнює одиницями  
 \*  
 \** ***@param n*** *розмірність  
 \*/* **public static int**[] inputVector(**int** n) {  
 **int**[] result = **new int**[n];  
 **int** start = 4;  
 **for** (**int** i = 0; i < result.**length**; i++) {  
 result[i] = start + i;  
 }  
 **return** result;  
 }  
  
 **public static int** inputConstant() {  
 **return** 1;  
 }  
  
 */\*\*  
 \* Додає матриці  
 \*  
 \** ***@param param1*** *матриця 1  
 \** ***@param param2*** *матриця 2  
 \** ***@return*** *сума матриць  
 \*/* **public static int**[][] addMatrix(**int**[][] param1, **int**[][] param2, **int** const1, **int** const2) {  
 **if** (param1.**length** != param2.**length**) {  
 System.***out***.println(**"Нельзя суммировать матрицы с разным количеством строк"**);  
 **return null**;  
 }  
 **int**[][] result = **new int**[param1.**length**][param1[0].**length**];  
 **for** (**int** i = 0; i < result.**length**; i++) {  
 result[i] = *addVectors*(param1[i], param2[i], const1, const2);  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* Додає вектори  
 \*  
 \** ***@param p1*** *вектор1  
 \** ***@param p2*** *вектор2  
 \** ***@return*** *сума векторів  
 \*/* **public static int**[] addVectors(**int**[] p1, **int**[] p2, **int** const1, **int** const2) {  
 **int**[] result = **new int**[p1.**length**];  
 **for** (**int** i = 0; i < result.**length**; i++) {  
 result[i] = p1[i] \* const1 + p2[i] \* const2;  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* Перемножає матриці  
 \*  
 \** ***@param param1*** *матриця-множник  
 \** ***@param param2*** *матриця-множене  
 \** ***@return*** *добуток матриць, розмірність - nxn  
 \*/* **public static int**[][] multMatrix(**int**[][] param1, **int**[][] param2) {  
 **if** (param1[0].**length** != param2.**length**) {  
 System.***out***.println(**"Нельзя умножать матрицы, количество элементов в строке которой не равно кол-ву столбцов в другой"**);  
 **return null**;  
 }  
 **int**[][] result = **new int**[param1.**length**][param1[0].**length**];  
 **for** (**int** k = 0; k < param1.**length**; k++) {  
 **for** (**int** i = 0; i < param1[0].**length**; i++) {  
 **for** (**int** j = 0; j < param2.**length**; j++) {  
 result[k][i] += param1[k][j] \* param2[j][i];  
 }  
 }  
 }  
 **return** result;  
 }  
  
 **public static** String formattedDeepToString(**int**[][] matrix) {  
 StringBuilder result = **new** StringBuilder();  
 **for** (**int** i = 0; i < matrix.**length**; i++) {  
 result.append(Arrays.*toString*(matrix[i])).append(**"\n"**);  
 }  
 **return** result.toString();  
 }  
  
 **public static int** min(**int**[] vector, **int** startIndex, **int** endIndex) {  
 **int** result = vector[startIndex];  
 **for** (**int** i = startIndex + 1; i < endIndex; i++) {  
 result = result > vector[i] ? vector[i] : result;  
 }  
 **return** result;  
 }  
  
 */\*\*  
 \* Метод выполняет обрезку строк матрицы. Возвращает строки в matrixToTruncate начиная с offset количеством rowCount  
 \*  
 \** ***@param matrixToTruncate*** *матрица, с которой будут вырезаны строки  
 \** ***@param offset*** *номер строки, с которой будет выполнена обрезка  
 \** ***@param rowCount*** *количество строк  
 \** ***@return*** *матрица, количество строк которой = rowCount и строки идентичны строкам в matrixToTruncate начиная с  
 \* offset позиции  
 \*/* **public static int**[][] truncateMatrix(**int**[][] matrixToTruncate, **int** offset, **int** rowCount) {  
 **int** result[][] = **new int**[rowCount][];  
 System.*arraycopy*(matrixToTruncate, offset, result, 0, rowCount);  
 **return** result;  
 }  
  
}