Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота №6
Технології паралельних обчислень

Виконав студент групи ІТ-03: Чабан А.Є.		Перевірив: Дифучина О.Ю
		Дата:
		Оцінка:

Київ 2023

Завдання:

5.6 Завдання до комп'ютерного практикуму 6 «Розробка паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності»

1. Ознайомитись з методами блокуючого та неблокуючого обміну повідомленнями типу point-to-point (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями (лістинг 1). **30 балів.**

3. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями. **30 балів.**
4. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями. **40 балів.**

Хід виконання:

Змінимо код з лістингу на MPJ Express та реалізуємо не блокуючу версію алгоритму. Виконаємо експерименти по дослідженню швидкодії алгоритмів

#	Number of workers					
	2		4		5	
	Matrix Size					
	1000	2000	1000	2000	1000	2000
blocking	6112	63595	3798	48577	3553	39364
non-blocking	5019	60551	3695	39841	3478	37988
Speed-up	1,217772	1,050272	1,027876	1,219272	1,021564	1,036222

У блокуючій реалізації взаємодія між процесами відбувається за допомогою методів Send та Recv з блокуванням виконання до завершення передачі даних. MASTER процес розсилає задачі worker'ам і отримує результати обчислень. В цій реалізації, коли майстер процес надсилає дані, він затримується доти, доки воркери не закінчать обчислення та не надішлють результати назад. Така реалізація є простою, але блокує виконання майстер процесу до завершення сплікування з усіма воркерами.

Неблокуюча версія використовує неблокуючі операції, які не зупиняють виконання процесу до завершення передачі. У цьому випадку процес MASTER використовує методи Isend для неблокуючої передачі даних, воркери використовують методи Irecv для неблокуючого отримання даних.

Блокуючий:

```
package lab6;

import mpi.*;

import java.util.Arrays;

public class Main {

    static final int N = 1000;
    static final int MASTER = 0;
    static final int FROM_MASTER = 1;
    static final int FROM_WORKER = 5;

    public static void main(String[] args) {
        {
            MPI.Init(args);

            int currentProcess = MPI.COMM_WORLD.Rank();
            int processesCount = MPI.COMM_WORLD.Size();

            int workersCount = processesCount - 1;
            if (N % workersCount != 0) {
                if (currentProcess == MASTER) {
                    System.out.println("It is impossible to allocate the specified number
```

```
of calls to "
                            + processesCount + " threads!");
                }

                MPI.Finalize();
                return;
            }
            int rowsPerProcess = N / workersCount;

            double[][] matrixB = new double[N][N];

            if (currentProcess == MASTER) {
                double[][] matrixA = new double[N][N];
                double[][] matrixC = new double[N][N];

                long startTime = System.currentTimeMillis();

                MatrixTools.fillMatrix(matrixA, 8);
                MatrixTools.fillMatrix(matrixB, 8);
                double[] matrixB1D = MatrixTools.convert2DTo1D(matrixB);

                for (int dest = 1; dest <= workersCount; dest++) {
                    int offSet = (dest - 1);
                    int startRow = offSet * rowsPerProcess;
                    int endRow = startRow + rowsPerProcess;

                    double[] subMatrixA1D =
MatrixTools.convert2DTo1D(Arrays.copyOfRange(matrixA, startRow, endRow));

                    MPI.COMM_WORLD.Send(new int[]{offSet}, 0, 1, MPI.INT, dest,
FROM_MASTER);
                    MPI.COMM_WORLD.Send(subMatrixA1D, 0, rowsPerProcess * N, MPI.DOUBLE,
dest, FROM_MASTER + 1);
                    MPI.COMM_WORLD.Send(matrixB1D, 0, N * N, MPI.DOUBLE, dest, FROM_MASTER
+ 2);
                }

                for (int source = 1; source <= workersCount; source++) {
                    int[] offset = new int[1];
                    MPI.COMM_WORLD.Recv(offset, 0, 1, MPI.INT, source, FROM_WORKER);

                    double[] resultMatrix1D = new double[rowsPerProcess * N];
                    MPI.COMM_WORLD.Recv(resultMatrix1D, 0, rowsPerProcess * N, MPI.DOUBLE,
source, FROM_WORKER + 1);

                    MatrixTools.fillMatrixWithOffset(matrixC, resultMatrix1D, offset[0] *
rowsPerProcess);
                }

                System.out.println("Execution time of blocking: " +
(System.currentTimeMillis() - startTime) + " ms" +
                        " for " + workersCount + " workers");
//                MatrixTools.printMatrix(matrixC);
            } else {
                int[] offset = new int[1];
                MPI.COMM_WORLD.Recv(offset, 0, 1, MPI.INT, MASTER, FROM_MASTER);

                double[] subMatrixA1D = new double[rowsPerProcess * N];
                MPI.COMM_WORLD.Recv(subMatrixA1D, 0, rowsPerProcess * N, MPI.DOUBLE,
MASTER, FROM_MASTER + 1);
                double[][] subMatrixA = MatrixTools.convert1DTo2D(subMatrixA1D,
rowsPerProcess, N);

                double[] matrixB1D = new double[N * N];
                MPI.COMM_WORLD.Recv(matrixB1D, 0, N * N, MPI.DOUBLE, MASTER, FROM_MASTER +
2);
                matrixB = MatrixTools.convert1DTo2D(matrixB1D, N, N);
```

```
                double[][] resultMatrix = new double[rowsPerProcess][N];
                for (int i = 0; i < rowsPerProcess; i++) {
                    for (int j = 0; j < N; j++) {
                        for (int k = 0; k < N; k++) {
                            resultMatrix[i][j] += subMatrixA[i][k] * matrixB[k][j];
                        }
                    }
                }

                MPI.COMM_WORLD.Send(offset, 0, 1, MPI.INT, MASTER, FROM_WORKER);
                MPI.COMM_WORLD.Send(MatrixTools.convert2DTo1D(resultMatrix), 0,
rowsPerProcess * N, MPI.DOUBLE, MASTER, FROM_WORKER + 1);
            }

            MPI.Finalize();
        }
    }
}
```

Неблокуючий:

```
package lab6;

import mpi.*;

import java.util.Arrays;

public class Main {

    static final int N = 1000;
    static final int MASTER = 0;
    static final int FROM_MASTER = 1;
    static final int FROM_WORKER = 5;

    public static void main(String[] args) {
        {
            MPI.Init(args);

            int currentProcess = MPI.COMM_WORLD.Rank();
            int processesCount = MPI.COMM_WORLD.Size();

            int workersCount = processesCount - 1;
            if (N % workersCount != 0) {
                if (currentProcess == MASTER) {
                    System.out.println("It is impossible to allocate the specified number
of calls to "
                            + processesCount + " threads!");
                }

                MPI.Finalize();
                return;
            }
            int rowsPerProcess = N / workersCount;

            double[][] matrixB = new double[N][N];

            if (currentProcess == MASTER) {
                double[][] matrixA = new double[N][N];
                double[][] matrixC = new double[N][N];

                long startTime = System.currentTimeMillis();

                MatrixTools.fillMatrix(matrixA, 8);
                MatrixTools.fillMatrix(matrixB, 8);
                double[] matrixB1D = MatrixTools.convert2DTo1D(matrixB);

                for (int dest = 1; dest <= workersCount; dest++) {
                    int offSet = (dest - 1);
                    int startRow = offSet * rowsPerProcess;
                    int endRow = startRow + rowsPerProcess;

                    double[] subMatrixA1D =
MatrixTools.convert2DTo1D(Arrays.copyOfRange(matrixA, startRow, endRow));

                    MPI.COMM_WORLD.Isend(new int[]{offSet}, 0, 1, MPI.INT, dest,
FROM_MASTER);
                    MPI.COMM_WORLD.Isend(subMatrixA1D, 0, rowsPerProcess * N, MPI.DOUBLE,
dest, FROM_MASTER + 1);
                    MPI.COMM_WORLD.Isend(matrixB1D, 0, N * N, MPI.DOUBLE, dest, FROM_MASTER
+ 2);
                }

                for (int source = 1; source <= workersCount; source++) {
                    int[] offset = new int[1];
                    double[] resultMatrix1D = new double[rowsPerProcess * N];

                    var offsetRequest = MPI.COMM_WORLD.Irecv(offset, 0, 1, MPI.INT, source,
FROM_WORKER);
```

```
                    MPI.COMM_WORLD.Recv(resultMatrix1D, 0, rowsPerProcess * N, MPI.DOUBLE,
source, FROM_WORKER + 1);
                    offsetRequest.Wait();

                    MatrixTools.fillMatrixWithOffset(matrixC, resultMatrix1D, offset[0] *
rowsPerProcess);
                }

                System.out.println("Execution time of non-blocking: " +
(System.currentTimeMillis() - startTime) + " ms" +
                        " for " + workersCount + " workers");
//                MatrixTools.printMatrix(matrixC);
            } else {
                int[] offset = new int[1];
                var offsetRequest = MPI.COMM_WORLD.Irecv(offset, 0, 1, MPI.INT, MASTER,
FROM_MASTER);

                double[] subMatrixA1D = new double[rowsPerProcess * N];
                var subMatrixRequest = MPI.COMM_WORLD.Irecv(subMatrixA1D, 0, rowsPerProcess
* N, MPI.DOUBLE, MASTER, FROM_MASTER + 1);

                double[] matrixB1D = new double[N * N];
                MPI.COMM_WORLD.Recv(matrixB1D, 0, N * N, MPI.DOUBLE, MASTER, FROM_MASTER +
2);

                offsetRequest.Wait();
                subMatrixRequest.Wait();

                double[][] subMatrixA = MatrixTools.convert1DTo2D(subMatrixA1D,
rowsPerProcess, N);
                matrixB = MatrixTools.convert1DTo2D(matrixB1D, N, N);

                double[][] resultMatrix = new double[rowsPerProcess][N];
                for (int i = 0; i < rowsPerProcess; i++) {
                    for (int j = 0; j < N; j++) {
                        for (int k = 0; k < N; k++) {
                            resultMatrix[i][j] += subMatrixA[i][k] * matrixB[k][j];
                        }
                    }
                }

                MPI.COMM_WORLD.Isend(offset, 0, 1, MPI.INT, MASTER, FROM_WORKER);
                MPI.COMM_WORLD.Isend(MatrixTools.convert2DTo1D(resultMatrix), 0,
rowsPerProcess * N, MPI.DOUBLE, MASTER, FROM_WORKER + 1);
            }

            MPI.Finalize();
        }
    }
}
```

Допоміжний клас MatrixTools:

```
package lab6;

public class MatrixTools {
    public static void printMatrix(double[][] matrix) {
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }

    public static void fillMatrix(double[][] arr, double value) {

        int rows = arr.length;
```

```
        int cols = arr[0].length;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                arr[i][j] = arr[i][j] = value;
            }
        }
    }

    public static double[] convert2DTo1D(double[][] arr2D) {
        int rows = arr2D.length;
        int cols = arr2D[0].length;
        double[] arr1D = new double[rows * cols];
        int index = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                arr1D[index++] = arr2D[i][j];
            }
        }

        return arr1D;
    }

    public static double[][] convert1DTo2D(double[] arr1D, int rows, int cols) {
        double[][] arr2D = new double[rows][cols];
        int index = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                arr2D[i][j] = arr1D[index++];
            }
        }

        return arr2D;
    }

    public static void fillMatrixWithOffset(double[][] matrix, double[] array, int
offsetByRows) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        int index = 0;

        for (int i = offsetByRows; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if(index == array.length)
                    return;
                matrix[i][j] = array[index];
                index++;
            }
        }
    }
}
```