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

Звіт

з комп’ютерного практикуму №6 з дисципліни
«Технології паралельних обчислень»

«Розробка паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності»

Виконав(ла) *ІП-11 Прищепа В. С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучин А. Ю.* _____
(прізвище, ім'я, по батькові)

Київ 2024

Завдання

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.

Виконання

Лістинг

Blocking.java

```
package org.example;

import mpi.*;

import java.util.Random;

public class Blocking {
    private static final int MASTER = 0;
    private static final int TAG_OFFSET = 0;
    private static final int TAG_ROWS = 1;
    private static final int TAG_MATRIX_A = 2;
    private static final int TAG_MATRIX_B = 3;
    private static final int TAG_RESULT = 4;
    private static final int MATRIX_SIZE = 5;

    public static void main(String[] args) {
        int offset;
        int rows;
```

```
        int[] A = new int[MATRIX_SIZE * MATRIX_SIZE];
        int[] B = new int[MATRIX_SIZE * MATRIX_SIZE];
        int[] C = new int[MATRIX_SIZE * MATRIX_SIZE];

        MPI.Init(args);

        int me = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();

        if (size < 2) {
            MPI.COMM_WORLD.Abort(1);
            throw new RuntimeException("Need at least two MPI tasks. Quitting...\n");
        }

        int numWorkers = size - 1;
        if (me == MASTER) {
            System.out.printf("MPI has started with %d tasks.\n", size);

            Random random = new Random();
            for (int i = 0; i < MATRIX_SIZE; i++) {
                for (int j = 0; j < MATRIX_SIZE; j++) {
                    A[MATRIX_SIZE * i + j] = random.nextInt(10);
                }
            }
            for (int i = 0; i < MATRIX_SIZE; i++) {
                for (int j = 0; j < MATRIX_SIZE; j++) {
                    B[MATRIX_SIZE * i + j] = random.nextInt(10);
                }
            }
            //printMatrix(A, "A");
            //printMatrix(B, "B");

            long startTime = System.currentTimeMillis();

            int avgRow = MATRIX_SIZE / numWorkers;
            int extra = MATRIX_SIZE % numWorkers;
            offset = 0;
            for (int dest = 1; dest <= numWorkers; dest++) {
                rows = (dest <= extra) ? avgRow + 1 : avgRow;
                MPI.COMM_WORLD.Send(new int[] { offset * MATRIX_SIZE}, 0, 1,
MPI.INT, dest, TAG_OFFSET);
                MPI.COMM_WORLD.Send(new int[] { rows }, 0, 1, MPI.INT, dest,
TAG_ROWS);
```

```
            MPI.COMM_WORLD.Send(A, offset * MATRIX_SIZE, rows *
MATRIX_SIZE, MPI.INT, dest, TAG_MATRIX_A);
            MPI.COMM_WORLD.Send(B, 0, MATRIX_SIZE * MATRIX_SIZE,
MPI.INT, dest, TAG_MATRIX_B);
            offset = offset + rows;
        }

        for (int source = 1; source <= numWorkers; source++) {
            int[] offsetBuffer = new int[1];
            MPI.COMM_WORLD.Recv(offsetBuffer, 0, 1, MPI.INT, source,
TAG_OFFSET);
            offset = offsetBuffer[0];
            int[] rowsBuffer = new int[1];
            MPI.COMM_WORLD.Recv(rowsBuffer, 0, 1, MPI.INT, source,
TAG_ROWS);
            rows = rowsBuffer[0];
            MPI.COMM_WORLD.Recv(C, offset, rows * MATRIX_SIZE, MPI.INT,
source, TAG_RESULT);
        }

        System.out.printf("Execution time for matrix %dx%d and %d workers:
%dms\n", MATRIX_SIZE, MATRIX_SIZE, numWorkers, System.currentTimeMillis()
- startTime);

        //printMatrix(C, "Result");
    } else {
        int[] offsetBuffer = new int[1];
        MPI.COMM_WORLD.Recv(offsetBuffer, 0, 1, MPI.INT, MASTER,
TAG_OFFSET);
        offset = offsetBuffer[0];
        int[] rowsBuffer = new int[1];
        MPI.COMM_WORLD.Recv(rowsBuffer, 0, 1, MPI.INT, MASTER,
TAG_ROWS);
        rows = rowsBuffer[0];

        MPI.COMM_WORLD.Recv(A, offset, rows * MATRIX_SIZE, MPI.INT,
MASTER, TAG_MATRIX_A);
        MPI.COMM_WORLD.Recv(B, 0, MATRIX_SIZE * MATRIX_SIZE, MPI.INT,
MASTER, TAG_MATRIX_B);

        // Perform matrix multiplication
        for (int k = 0; k < MATRIX_SIZE; k++) {
            for (int i = 0; i < rows; i++) {
                C[offset + MATRIX_SIZE * i + k] = 0;
                for (int j = 0; j < MATRIX_SIZE; j++) {
```

```
                    C[offset + MATRIX_SIZE * i + k] += A[offset + MATRIX_SIZE * i +
j] * B[MATRIX_SIZE * j + k];
                }
            }
        }

        MPI.COMM_WORLD.Send(new int[] { offset }, 0, 1, MPI.INT, MASTER,
TAG_OFFSET);
        MPI.COMM_WORLD.Send(new int[] { rows }, 0, 1, MPI.INT, MASTER,
TAG_ROWS);
        MPI.COMM_WORLD.Send(C, offset, rows * MATRIX_SIZE, MPI.INT,
MASTER, TAG_RESULT);
    }

    MPI.Finalize();
  }

  private static void printMatrix(int[] matrix, String name) {
    System.out.printf("Matrix %s:\n", name);
    for (int i = 0; i < MATRIX_SIZE; i++) {
      System.out.println();
      for (int j = 0; j < MATRIX_SIZE; j++)
        System.out.printf("%6d ", matrix[MATRIX_SIZE * i + j]);
    }
    System.out.println("\n" + "*".repeat(10));
  }
}
```

NonblockingMain.java

```
package org.example;

import mpi.*;

import java.util.Random;

public class NonblockingMain {
  private static final int MASTER = 0;
  private static final int TAG_OFFSET = 0;
  private static final int TAG_ROWS = 1;
  private static final int TAG_MATRIX_A = 2;
  private static final int TAG_MATRIX_B = 3;
  private static final int TAG_RESULT = 4;
  private static final int MATRIX_SIZE = 5;
```

```
public static void main(String[] args) {
    int offset;
    int rows;


    int[] A = new int[MATRIX_SIZE * MATRIX_SIZE];
    int[] B = new int[MATRIX_SIZE * MATRIX_SIZE];
    int[] C = new int[MATRIX_SIZE * MATRIX_SIZE];

    MPI.Init(args);

    int me = MPI.COMM_WORLD.Rank();
    int size = MPI.COMM_WORLD.Size();

    if (size < 2) {
        MPI.COMM_WORLD.Abort(1);
        throw new RuntimeException("Need at least two MPI tasks. Quitting...\n");
    }

    int numWorkers = size - 1;
    if (me == MASTER) {
        System.out.printf("MPI has started with %d tasks.\n", size);

        Random random = new Random();
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                A[MATRIX_SIZE * i + j] = random.nextInt(10);
            }
        }
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                B[MATRIX_SIZE * i + j] = random.nextInt(10);
            }
        }
        //printMatrix(A, "A");
        //printMatrix(B, "B");

        long startTime = System.currentTimeMillis();

        int avgRow = MATRIX_SIZE / numWorkers;
        int extra = MATRIX_SIZE % numWorkers;
        offset = 0;

        for (int dest = 1; dest <= numWorkers; dest++) {
            rows = (dest <= extra) ? avgRow + 1 : avgRow;
```

```
            Request offsetRequest = MPI.COMM_WORLD.Isend(new int[] { offset *
MATRIX_SIZE}, 0, 1, MPI.INT, dest, TAG_OFFSET);
            Request rowsRequest = MPI.COMM_WORLD.Isend(new int[] { rows }, 0,
1, MPI.INT, dest, TAG_ROWS);
            Request matrixARequest = MPI.COMM_WORLD.Isend(A, offset *
MATRIX_SIZE, rows * MATRIX_SIZE, MPI.INT, dest, TAG_MATRIX_A);
            Request matrixBRequest = MPI.COMM_WORLD.Isend(B, 0,
MATRIX_SIZE * MATRIX_SIZE, MPI.INT, dest, TAG_MATRIX_B);
            Request.Waitall(new Request[]{offsetRequest, rowsRequest,
matrixARequest, matrixBRequest});
            offset = offset + rows;
         }

         for (int source = 1; source <= numWorkers; source++) {
            int[] offsetBuffer = new int[1];
            Request offsetRequest = MPI.COMM_WORLD.Irecv(offsetBuffer, 0, 1,
MPI.INT, source, TAG_OFFSET);
            int[] rowsBuffer = new int[1];
            Request rowsRequest = MPI.COMM_WORLD.Irecv(rowsBuffer, 0, 1,
MPI.INT, source, TAG_ROWS);
            Request.Waitall(new Request[]{offsetRequest, rowsRequest});
            offset = offsetBuffer[0];
            rows = rowsBuffer[0];
            MPI.COMM_WORLD.Recv(C, offset, rows * MATRIX_SIZE, MPI.INT,
source, TAG_RESULT);
         }

         System.out.printf("Execution time for matrix %dx%d and %d workers:
%dms\n", MATRIX_SIZE, MATRIX_SIZE, numWorkers, System.currentTimeMillis()
- startTime);

         //printMatrix(C, "Result");
      } else {
         int[] offsetBuffer = new int[1];
         Request recvOffsetRequest = MPI.COMM_WORLD.Irecv(offsetBuffer, 0, 1,
MPI.INT, MASTER, TAG_OFFSET);
         int[] rowsBuffer = new int[1];
         Request recvRowsRequest = MPI.COMM_WORLD.Irecv(rowsBuffer, 0, 1,
MPI.INT, MASTER, TAG_ROWS);
         Request.Waitall(new Request[]{recvOffsetRequest, recvRowsRequest});
         offset = offsetBuffer[0];
         rows = rowsBuffer[0];

         Request recvMatrixARequest = MPI.COMM_WORLD.Irecv(A, offset, rows *
MATRIX_SIZE, MPI.INT, MASTER, TAG_MATRIX_A);
```

```
            Request recvMatrixBRequest = MPI.COMM_WORLD.Irecv(B, 0,
MATRIX_SIZE * MATRIX_SIZE, MPI.INT, MASTER, TAG_MATRIX_B);
            Request.Waitall(new Request[]{recvMatrixARequest, recvMatrixBRequest});

            // Perform matrix multiplication
            for (int k = 0; k < MATRIX_SIZE; k++) {
                for (int i = 0; i < rows; i++) {
                    C[offset + MATRIX_SIZE * i + k] = 0;
                    for (int j = 0; j < MATRIX_SIZE; j++) {
                        C[offset + MATRIX_SIZE * i + k] += A[offset + MATRIX_SIZE * i +
j] * B[MATRIX_SIZE * j + k];
                    }
                }
            }

            Request sendOffsetRequest = MPI.COMM_WORLD.Isend(new int[]
{ offset }, 0, 1, MPI.INT, MASTER, TAG_OFFSET);
            Request sendRowsRequest = MPI.COMM_WORLD.Isend(new int[] { rows },
0, 1, MPI.INT, MASTER, TAG_ROWS);
            Request sendMatrixResultRequest = MPI.COMM_WORLD.Isend(C, offset,
rows * MATRIX_SIZE, MPI.INT, MASTER, TAG_RESULT);
            Request.Waitall(new Request[]{sendOffsetRequest, sendRowsRequest,
sendMatrixResultRequest});
        }

        MPI.Finalize();
    }

    private static void printMatrix(int[] matrix, String name) {
        System.out.printf("Matrix %s:\n", name);
        for (int i = 0; i < MATRIX_SIZE; i++) {
            System.out.println();
            for (int j = 0; j < MATRIX_SIZE; j++)
                System.out.printf("%6d ", matrix[MATRIX_SIZE * i + j]);
        }
        System.out.println("\n" + "*".repeat(10));
    }
}
```

Результат

```
MPJ Express (0.44) is started in the multicore configuration
MPI has started with 2 tasks.
Matrix A:
     1      7      2      8      4
     2      4      9      3      7
     8      8      1      1      7
     8      1      7      7      7
     8      7      1      3      4
**********
Matrix B:
     9      7      6      5      5
     9      4      1      4      1
     5      6      4      8      8
     5      8      6      9      7
     1      5      4      9      0
**********
Execution time for matrix 5x5 and 1 workers: 2ms
Matrix Result:
   126    131     85    157     84
   121    143     98    188    107
   161    137     94    152     63
   158    193    147    226    146
   159    134     93    139     76
**********
```

Виконання алгоритму паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями.

```
MPJ Express (0.44) is started in the multicore configuration
MPI has started with 2 tasks.
Matrix A:
     5      9      0      1      8
     0      6      4      9      3
     1      2      8      4      2
     8      3      7      1      3
     8      5      9      0      6
**********
Matrix B:
     4      9      6      7      9
     8      2      3      0      8
     8      4      2      7      5
     5      2      1      9      9
     8      5      9      0      9
**********
Execution time for matrix 5x5 and 1 workers: 2ms
Matrix Result:
   161    105    130     44    198
   149     61     62    109    176
   120     63     50     99    119
   141    123     99    114    167
   192    148    135    119    211
**********
```

Виконання алгоритму паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями.

Обидва алгоритми працюють коректно.

Тепер дослідимо ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняймо ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями.

Розмірність	1 поток		2 потоки		4 потоки		8 потоків	
	Б	Н	Б	Н	Б	Н	Б	Н
500x500	266	270	151	143	87	82	66	68
1000x1000	2324	2141	1123	1223	618	584	407	402
2000x2000	29453	28465	14690	14401	7568	7548	4515	4415
2500x2500	73069	70944	36987	36918	18811	18733	10670	10903

Бачимо, що прискорення для неблокуючих методів є, проте воно дуже незначне. Загалом, зі збільшенням кількості вузлів маєже не зростає прискорення.

Висновок: Під час виконання комп’ютерного практикуму я закріпив знання про MPI та його блокуючі та неблокуючі методи зі зв’язком «один до одного». Реалізував з допомогою MPJ Express множення матриць та дослідив алгоритм цього множення, збільшуючи кількість вузлів та розмір матриці.