ABSTRACT

To understand the basic mechanism of Message Passing Interface through MPI4Lecture Library

Cyril Naves & Ponathipan Jawahar

LAB SESSION 4

CONCURRENCY & PARALLELISM

Exercise 1: Discover MPI4Lectures

Create a new JAVA project and link it to the MPI4Lecture library. Get used to the syntax through the available example. The ring.

Code:

```
package org.m1int.mpi.ring;

import fr.unice.mpi4lectures.MPIMessage;
import fr.unice.mpi4lectures.MPIProcess;

public class Ring extends MPIProcess {
/**
 * This is entry point in the JVM.
 */
public static void main(String[] args)

{
	// This creates 6 processors of type
	// create 6 instances of your processor
	MPIProcess.createProcessors(Ring.class, 6);
}

/**
 * The implementation of this method is the code the processor will execute.
 */
@Override
public void main() {
	// get the rank of this processor
	int rank = Rank();
	System.out.println("Starting processor with rank " + rank);

	// this is the rendez-vous point for all processors
	Barrier(0, 1, 2, 3, 4, 5);

	// get the number of processor in the MPI system
	// remember you created 6 processors...
	int size = Size();

	// if this processor is the first one
	if (rank == 0) {
		// it has to create the message
		// and send it to the first processor
		Send("catch me!".getBytes(), 9, MPIMessage.TYPE.CHAR, 1, "");

		byte[] buffer = new byte[9];
		Recv(buffer, 9, MPIMessage.TYPE.CHAR, size - 1, "");// blocked for responce from other
		System.out.println("message is back home '" + new String(buffer) + "'");
	} else {
		// this is not the first processor, whatever its rank
		// it waits for the message coming from the previous processos (rank
		// - 1)
		byte[] buffer = new byte[9];
		Recv(buffer, 9, MPIMessage.TYPE.CHAR, rank - 1, "");
		System.out.println("processor " + rank + " has just received the message '" + new String(buffer) + "'");

		// alter the text by replac a character at a random index by
		// underscore
```

```
                    buffer[(int) (Math.random() * 9)] = '_';

                    // and send the message to the next processor (rank + 1)
                    System.out.println("processor " + rank + " forwards the message to " + ((rank + 1) % size));
                    Send(buffer, 9, MPIMessage.TYPE.CHAR, (rank + 1) % size, "");
          }
}
}
```

Output:

```
Creating 6 processes of type org.m1int.mpi.ring.Ring
Starting processor with rank 0
Starting processor with rank 1
Starting processor with rank 3
Starting processor with rank 4
Starting processor with rank 5
Starting processor with rank 2
processor 1 has just received the message 'catch me!'
processor 1 forwards the message to 2
processor 2 has just received the message 'catch_me!'
processor 2 forwards the message to 3
processor 3 has just received the message 'catch_me!'
processor 3 forwards the message to 4
processor 4 has just received the message 'catch__e!'
processor 4 forwards the message to 5
processor 5 has just received the message 'ca_ch__e!'
processor 5 forwards the message to 0
message is back home 'ca_ch__e!'
```

Summary of Ring Implementation through MPI:

1) Ring Implementation is a basic MPI model where each processor wait for the preceding processor to send message and then receive it. It is then forwarded to the subsequent processor .
2)Ring Class extends the abstract class MPIProcess and to implements the main() method.
3)main() method contains the attributes of the Rank of each process as well as the total size as well as Barriers(int...) which specifies the meeting point of the each processor
4) Then Send() and Recv() method is used to send and receive the message to the target processor based on the rank of each processor finally here the processor 5 send the message back to the origin of the message which is where the message was first communicated.
5) MPIMessage Enum is also used to specify the type of message being communicated.

Exercise 2: The bubble sorter is back

Re-implement the multithreading layer of the bubble sorter(TP1 exercise 4) in a MPI fashion.

Code:

```
package org.m1int.mpi.bubblesorter;

import fr.unice.mpi4lectures.MPIProcess;
import fr.unice.mpi4lectures.MPIMessage;

public class BubbleSort_MPI extends MPIProcess {

/*
 * In the bubble sort implementation, 4 objects of BubbleSorter and handing it
 * over to 4 different processors to do sorting.5 MPI processors are created
 * through main method which receive message, perform sorting and then send
 * messages to each of the other processor Finally processor 5 performs the
 * fusion of the sorted elements.
 *
 */
static BubbleSorter bubbleSorter1;
static BubbleSorter bubbleSorter2;
static BubbleSorter bubbleSorter3;
static BubbleSorter bubbleSorter4;
static byte counter = 0;

/**
 * The implementation of this method is the code the processor will execute.
 */
@Override
public void main() {
        // get the rank of this processor
        int rank = Rank();
        System.out.println("Starting processor with rank " + rank);

        // this is the rendez-vous point for all processors
        Barrier(0, 1, 2, 3, 4);

        // get the number of processor in the MPI system
        // remember you created 4 processors...
        int size = Size();

        // if this processor is the first one
        if (rank == 0) {
                bubbleSorter1.bubble_srt();
                counter++;
                Send(new byte[] { counter }, 1, MPIMessage.TYPE.CHAR, 4, "");
        } else if (rank == 1) {
                bubbleSorter2.bubble_srt();
                counter++;
                Send(new byte[] { counter }, 1, MPIMessage.TYPE.CHAR, 4, "");
        } else if (rank == 2) {
                bubbleSorter3.bubble_srt();
                counter++;
                Send(new byte[] { counter }, 1, MPIMessage.TYPE.CHAR, 4, "");
        } else if (rank == 3) {
                bubbleSorter4.bubble_srt();
                counter++;
```

```
				Send(new byte[] { counter }, 1, MPIMessage.TYPE.CHAR, 4, "");
		}

		else if (rank == 4) {

				byte[] counterArray = new byte[1];
				Recv(counterArray, 1, MPIMessage.TYPE.CHAR, 0, "");
				System.out.println("Processor " + counterArray[0] + " finished its task");
				Recv(counterArray, 1, MPIMessage.TYPE.CHAR, 1, "");
				System.out.println("Processor " + counterArray[0] + " finished its task");
				Recv(counterArray, 1, MPIMessage.TYPE.CHAR, 2, "");
				System.out.println("Processor " + counterArray[0] + " finished its task");
				Recv(counterArray, 1, MPIMessage.TYPE.CHAR, 3, "");
				System.out.println("Processor " + counterArray[0] + " finished its task");

				System.out.println("All worker processors finished tasks. Now lets do fusion");

				int[][] resultTab = new int[4][size];
				// get the sorted tabs from each sorter
				resultTab[0] = bubbleSorter1.getTab();
				resultTab[1] = bubbleSorter2.getTab();
				resultTab[2] = bubbleSorter3.getTab();
				resultTab[3] = bubbleSorter4.getTab();
				Fusion fusion = new Fusion(resultTab);
				// do the fusion

				// get the merged sorted array from fusion object
				int[] fusedResult = fusion.fusion();
				System.out.println("fusion is done");

				// print 10 elements of of resultab
				for (int j = 0; j < 10; j++)
						System.out.print("   " + fusedResult[j]);

		}
}

/**
 * Main method where 5 MPI Processors are created
 *
 * @param args
 */
public static void main(String[] args) {
		// Set up the table with random values
		int nb = 4; // number of tables
		int size = 10000;// Size of each table
		int[][] tab = new int[nb][size];
		for (int i = 0; i < nb; i++) {
				for (int j = 0; j < size; j++) {
						tab[i][j] = (int) (Math.random() * size);
				}
		}
		// Create the bubble sorters

		bubbleSorter1 = new BubbleSorter(tab[0]);
		bubbleSorter2 = new BubbleSorter(tab[1]);
		bubbleSorter3 = new BubbleSorter(tab[2]);
		bubbleSorter4 = new BubbleSorter(tab[3]);
```

```
        // create creating 5 processors
        MPIProcess.createProcessors(BubbleSort_MPI.class, 5);
}

}
```

Output:

```
Creating 5 processes of type org.m1int.mpi.bubblesorter.BubbleSort_MPI
Starting processor with rank 0
Starting processor with rank 1
Starting processor with rank 2
Starting processor with rank 3
Starting processor with rank 4
Processor 2 finished its task
Processor 1 finished its task
Processor 4 finished its task
Processor 3 finished its task
All worker processors finished tasks. Now lets do fusion
fusion is done
   0   0   0   1   1   1   1   1   1   1
```

Summary of Bubble Sorting Implementation through MPI:
1)BubbleSort_MPI Class extends the abstract class MPIProcess and overrides the main method.
In this 5 processors are created , in which 4 will handle the sorting of splitted array and the 5th processor will implement the fusion
2)It utilizes the old Bubble Sort implementation as well as fusion implementation
3)In the main() method each of the array contain the 40000 elements is split into 4 separate array and passed to the separate processor based on the rank0,rank1, rank 2, rank3.
4)Then after finishing the task each of them send the final completion message to the 5th processor through send() method which performs the final fusion of the arrays.
5) As a sample only 10 elements are printed out of the 40000 sorted elements

Exercise 3: Matrix multiplication

Write an MPI program that takes an input two compatible matrices (MxN) and (NxP) and the number of available nodes for computation. The program computes the matrix multiplication while efficiently using all the available nodes.
1/ Determine the rule to be followed to distribute efficiently the computations over the nodes.
2/Write the code.

Code:

```
package org.m1int.mpi.matrixmultiplication;

import fr.unice.mpi4lectures.MPIMessage;
import fr.unice.mpi4lectures.MPIProcess;
import java.util.*;

/**
 *
 * MatrixMultilplication Class consists of processes equal to the no of rows of
 * final matrix computed. Finally the resultant matrix is displayed after each
 * of the sub processes are completed In this case we handle [3 *4 ] [4*3]
 * matrix which results in [3*3] matrix where by the the process for row 1, 2,3
 * are computed and finally displayed in the result with the sending messages
 * all received from the previous processes.
 *
 *
 */
public class MatrixMutiplication extends MPIProcess {

static int M = 3, N = 4, P = 3;
static int[][] matrix1, matrix2, finalMatrix;
static int processors;

public static void main(String[] args) {

		matrix1 = new int[M][N];
		matrix2 = new int[N][P];
		Scanner input = new Scanner(System.in);
		finalMatrix = new int[M][P];
		System.out.println("Input of Matrix 1 with:" + M + " rows and " + N + " columns");
		for (int matrix1Row = 0; matrix1Row < M; matrix1Row++) {
			for (int matrix1Col = 0; matrix1Col < N; matrix1Col++) {
				System.out.println("MATRIX 1:: Enter the values for ROW " + matrix1Row + " COLUMN " +
matrix1Col);
				matrix1[matrix1Row][matrix1Col] = Integer.parseInt(input.nextLine());
			}
		}

		System.out.println("Input of MATRIX 2 with" + N + " rows " + P + " columns");
		for (int matrix2Row = 0; matrix2Row < N; matrix2Row++) {
			for (int matrix2Col = 0; matrix2Col < P; matrix2Col++) {
				System.out.println("MATRIX 2 :: Enter the values for ROW " + matrix2Row + " COLUMN " +
matrix2Col);
				matrix2[matrix2Row][matrix2Col] = Integer.parseInt(input.nextLine());
			}
		}

		processors = M + 1;
		MPIProcess.createProcessors(MatrixMutiplication.class, processors);
}

/**
 *
 * @param rowOfFirstMatrix
 * @param rowNumOfResultantMatrix
 */
static void multiply(int[] rowOfFirstMatrix, int rowNumOfResultantMatrix) {

		int sum = 0;
		int columnCount = 0;

		for (int col = 0; col < 3; col++) {
			for (int row = 0; row < matrix2.length; row++)// row<2
			{
				sum = sum + matrix1[rowNumOfResultantMatrix][columnCount] * matrix2[row][col];
				columnCount++;
			}
			finalMatrix[rowNumOfResultantMatrix][col] = sum;
			columnCount = 0;
			sum = 0;
		}
}

/**
 * The implementation of this method is the code the processor will execute.
 */
@Override
public void main() {
	// get the rank of this processor
	int rank = Rank();
	System.out.println("Starting processor with rank " + rank);

	// Creation of barrier Renedez-vous points
	int[] barriers = new int[processors];
	for (int i = 0; i < processors; i++) {
		barriers[i] = i;
	}
	Barrier(barriers);

	// if this processor is the first one
	if (rank == 0) // processor 0 is master processor
	{
		System.out.println("processor " + rank);

		// rowCount is also my processor count
		multiply(matrix1[0], 0);
		Send(new byte[] { (byte) (1) }, 1, MPIMessage.TYPE.CHAR, 3, "");
	}
	if (rank == 1) {
		System.out.println("processor " + rank);
		multiply(matrix1[1], 1);
		Send(new byte[] { (byte) (2) }, 1, MPIMessage.TYPE.CHAR, 3, "");
	}
	if (rank == 2) {
		System.out.println("processor " + rank);
		multiply(matrix1[2], 2);
		Send(new byte[] { (byte) (2) }, 1, MPIMessage.TYPE.CHAR, 3, "");
	}
	if (rank == 3) {
		byte[] buffer = new byte[1];
		Recv(buffer, 1, MPIMessage.TYPE.CHAR, 0, "");
		System.out.println("processor " + buffer[0] + " finished its task");
		Recv(buffer, 1, MPIMessage.TYPE.CHAR, 1, "");
		System.out.println("processor " + buffer[0] + " finished its task");
		Recv(buffer, 1, MPIMessage.TYPE.CHAR, 2, "");
		System.out.println("processor " + buffer[0] + " finished its task");
		System.out.println("Message received");
		System.out.println("Printing resultant matrix ");
		for (int row = 0; row < finalMatrix.length; row++) {
			for (int col = 0; col < finalMatrix[row].length; col++)

			{
				System.out.print(finalMatrix[row][col] + "\t");
			}
			System.out.println();
		}
	}

}
}
```

Output:

```
Input of Matrix 1 with:3 rows and 4 columns
MATRIX 1:: Enter the values for ROW 0 COLUMN 0
3
MATRIX 1:: Enter the values for ROW 0 COLUMN 1
4
MATRIX 1:: Enter the values for ROW 0 COLUMN 2
5
MATRIX 1:: Enter the values for ROW 0 COLUMN 3
5
MATRIX 1:: Enter the values for ROW 1 COLUMN 0
6
MATRIX 1:: Enter the values for ROW 1 COLUMN 1
3
MATRIX 1:: Enter the values for ROW 1 COLUMN 2
4
MATRIX 1:: Enter the values for ROW 1 COLUMN 3
5
MATRIX 1:: Enter the values for ROW 2 COLUMN 0
6
MATRIX 1:: Enter the values for ROW 2 COLUMN 1
7
MATRIX 1:: Enter the values for ROW 2 COLUMN 2
7
MATRIX 1:: Enter the values for ROW 2 COLUMN 3
3
Input of MATRIX 2 with4 rows 3 columns
MATRIX 2 :: Enter the values for ROW 0 COLUMN 0
2
MATRIX 2 :: Enter the values for ROW 0 COLUMN 1
4
MATRIX 2 :: Enter the values for ROW 0 COLUMN 2
5
MATRIX 2 :: Enter the values for ROW 1 COLUMN 0
6
MATRIX 2 :: Enter the values for ROW 1 COLUMN 1
6
MATRIX 2 :: Enter the values for ROW 1 COLUMN 2
7
MATRIX 2 :: Enter the values for ROW 2 COLUMN 0
7
MATRIX 2 :: Enter the values for ROW 2 COLUMN 1
4
MATRIX 2 :: Enter the values for ROW 2 COLUMN 2
4
MATRIX 2 :: Enter the values for ROW 3 COLUMN 0
3
MATRIX 2 :: Enter the values for ROW 3 COLUMN 1
2
MATRIX 2 :: Enter the values for ROW 3 COLUMN 2
8
Creating 4 processes of type org.m1int.mpi.matrixmultiplication.MatrixMutiplication
Starting processor with rank 0
Starting processor with rank 2
Starting processor with rank 1
Starting processor with rank 3
processor 0
processor 2
processor 1
processor 1 finished its task
processor 2 finished its task
processor 2 finished its task
Message received
Printing resultant matrix
80     66     103
73     68     107
112    100    131
```

Summary of Matrix Multiplication Implementation through MPI:

1) MatrixMultiplication Class extends the MPI Process which will implement in our case the multiplication of two matrices sequentially row by row.

2)MatrixMultiplication Class consists of processes equal to the no of rows of final matrix computed.

3) The resultant matrix is displayed after each of the sub processes are completed .In this case we handle [3 *4] [4*3] matrix which results in [3*3] matrix where by the process for each row 1, 2,3 are computed and finally displayed in the result with the sending messages.

4)Each processor individually performs the matrix multiplication of each row which ranges from rank 0,1,2 and send the messages to the final processor of rank 3

5)Processor of Rank 3 finally receives message from rank 0,1,2 processors and the display the final matrix which contains the computed results.

Exercise 4: Broadcasting

MPI4lectures is a basic implementation of MPI. For example, the broadcasting primitive is not implemented. Let's extends the MPI interface and the MPIImplementation class to provide the broadcasting feature.
1/ What are the possible broadcasting scheme? What is the impact on the API? 2/ Try it on an example.

Code:

```
package org.m1int.mpi.broadcasting;

import fr.unice.mpi4lectures.MPIMessage;
import fr.unice.mpi4lectures.MPIProcess;
import java.util.*;

/**
 *
 * In Broadcasting Implementation a simple producer consumer scenario is
 * replicated with one of the processes sending the message and the other one
 * receiving it from the corresponding sender.
 *
 */
public class Broadcasting extends MPIProcess {

static String message;
public static void main(String[] args) {

        System.out.println("Enter message");
        message = new Scanner(System.in).nextLine();
        MPIProcess.createProcessors(Broadcasting.class, 4);
}

/**
 *
 */
@Override
public void main() {

        int rank = Rank();
        int size = Size();
        Barrier(0, 1, 2, 3);
        if (rank == 0) {
                for (int processor = 1; processor < size; processor++) {
                        byte[] bytesBroadCast = message.getBytes();
                        Send(bytesBroadCast, bytesBroadCast.length, MPIMessage.TYPE.CHAR, processor, "");
                        System.out.println("broadcast message sent to processor " + processor);
                }
        }

        else {
                byte[] receivedMessage = message.getBytes();
                Recv(receivedMessage, receivedMessage.length, MPIMessage.TYPE.CHAR, 0, "");
                System.out.println("Processor " + rank + " received " + new String(receivedMessage));
        }

}

}
```

```
Output:
Enter message
This is MPI Implmenetation
Creating 4 processes of type org.m1int.mpi.broadcasting.Broadcasting
broadcast message sent to processor 1
broadcast message sent to processor 2
Processor 2 received This is MPI Implmenetation
Processor 1 received This is MPI Implmenetation
```

Summary of Broadcasting Implementation through MPI:

1) Broadcasting Implementation followed here is the basic producer consumer strategy where 4 processors are created.
2) Each of the processor with rank 0 and rank 1 acts as the transmitter and receiver correspondingly.
3)When rank 0 sends the message to rank 1 processor and then rank 1 receiver accepts the message and then displays it.
4) In this all the receivers ranged from rank 1, rank 2, rank3 get the message from rank 0.
5) API performs well for the type of single source and multiple receptors since each of them are subscribed to the rank 0 processor.