Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

**Комп’ютерний практикум № 6**

з дисципліни «Технології паралельних обчислень»

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

|  |  |
| --- | --- |
| **Виконав:**  студент групи ІТ-04  Чапча Святослав  Дата здачі \_\_\_\_\_\_\_\_  Захищено з балом \_\_\_\_\_\_\_ | **Перевірила:**  ас. кафедри ІПІ  Дифучина О.Ю. |

Київ 2023

**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.**

Лістинг програми:

**Block.java**

import mpi.\*;  
  
import static java.lang.System.*exit*;  
public class Block {  
 private static int *matrixSize* = 2000;  
 private static int *MASTER* = 0;  
 private static int *FROM\_MASTER* = 1;  
 private static int *FROM\_WORKER* = 1;  
  
 public static void main(String[] args) throws MPIException {  
 double[][] matrixA = Methods.*generateMatrix*(*matrixSize*);  
 double[][] matrixB = Methods.*generateMatrix*(*matrixSize*);  
 double[][] result = new double[*matrixSize*][*matrixSize*];  
 long startTime = System.*currentTimeMillis*();  
 long endTime = System.*currentTimeMillis*();  
  
 MPI.*Init*(args);  
 int size = MPI.*COMM\_WORLD*.Size();  
 int rank = MPI.*COMM\_WORLD*.Rank();  
  
 int countOfWorkers = size - 1;  
  
 if(size < 2 || *matrixSize* < countOfWorkers){  
 System.*out*.println("Num Of Workers Is Small");  
 MPI.*COMM\_WORLD*.Abort(1);  
 *exit*(1);  
 }  
  
 if(rank == *MASTER*){  
 System.*out*.println("Started with " + size + " tasks");  
 int work = *matrixSize* / countOfWorkers;  
 int extra = *matrixSize* % countOfWorkers;  
 for(int worker = 1; worker <= countOfWorkers; worker++){  
 int rowStart = (worker - 1) \* work;  
 int rowFinish = rowStart + work;  
 if(worker == countOfWorkers){  
 rowFinish+=extra;  
 }  
  
 double[][] blockA = Methods.*getBlock*(matrixA, rowStart, rowFinish, *matrixSize*);  
  
 MPI.*COMM\_WORLD*.Send(new int[]{rowStart}, 0, 1, MPI.*INT*, worker, *FROM\_MASTER*);  
 MPI.*COMM\_WORLD*.Send(new int[]{rowFinish}, 0, 1, MPI.*INT*, worker, *FROM\_MASTER*);  
 MPI.*COMM\_WORLD*.Send(blockA, 0, blockA.length , MPI.*OBJECT*, worker, *FROM\_MASTER*);  
 MPI.*COMM\_WORLD*.Send(matrixB, 0, matrixB.length , MPI.*OBJECT*, worker, *FROM\_MASTER*);  
 }  
  
 for(int worker = 1; worker <= countOfWorkers; worker++){  
 int[] rowStart = new int[1];  
 int[] rowFinish = new int[1];  
  
 MPI.*COMM\_WORLD*.Recv(rowStart, 0, 1, MPI.*INT*, worker, *FROM\_WORKER*);  
 MPI.*COMM\_WORLD*.Recv(rowFinish, 0, 1, MPI.*INT*, worker, *FROM\_WORKER*);  
  
 double[][] openedResult = new double[rowFinish[0] - rowStart[0] + 1][*matrixSize*];  
 MPI.*COMM\_WORLD*.Recv(openedResult, 0, openedResult.length, MPI.*OBJECT*, worker, *FROM\_WORKER*);  
  
 Methods.*addBlock*(openedResult, result, rowStart[0], rowFinish[0]);  
 }  
 endTime = System.*currentTimeMillis*();  
 System.*out*.println("Result: ");  
 //Methods.print(result);  
 System.*out*.println("Time: " + (endTime-startTime) + " ms");  
 } else {  
 int[] rowStart = new int[1];  
 int[] rowFinish = new int[1];  
  
 MPI.*COMM\_WORLD*.Recv(rowStart, 0, 1, MPI.*INT*, *MASTER*, *FROM\_MASTER*);  
 MPI.*COMM\_WORLD*.Recv(rowFinish, 0, 1, MPI.*INT*, *MASTER*, *FROM\_MASTER*);  
  
 double[][] openedBlockA = new double[rowFinish[0] - rowStart[0] + 1][*matrixSize*];  
 double[][] openedMatrixB = new double[*matrixSize*][*matrixSize*];  
  
 MPI.*COMM\_WORLD*.Recv(openedBlockA, 0, openedBlockA.length, MPI.*OBJECT*, *MASTER*, *FROM\_MASTER*);  
 MPI.*COMM\_WORLD*.Recv(openedMatrixB, 0, openedMatrixB.length, MPI.*OBJECT*, *MASTER*, *FROM\_MASTER*);  
 System.*out*.println("Row start: " + rowStart[0] + " Row finish: " + rowFinish[0] + " From task " + rank);  
  
 double[][] subResult = Methods.*multiply*(openedBlockA, openedMatrixB);  
  
 MPI.*COMM\_WORLD*.Send(rowStart, 0, 1, MPI.*INT*, *MASTER*, *FROM\_WORKER*);  
 MPI.*COMM\_WORLD*.Send(rowFinish, 0, 1, MPI.*INT*, *MASTER*, *FROM\_WORKER*);  
 MPI.*COMM\_WORLD*.Send(subResult, 0, subResult.length, MPI.*OBJECT*, *MASTER*, *FROM\_WORKER*);  
 }  
 MPI.*Finalize*();  
 }  
}

**Methods.java**

public class Methods {  
 public static double[][] generateMatrix(int size){  
 double[][] matrix = new double[size][size];  
 for (int i = 0; i < size; i++) {  
 double[] row = new double[size];  
 for (int j = 0; j < size; j++) {  
 row[j] = 1;  
 }  
 matrix[i] = row;  
 }  
 return matrix;  
 }  
  
 public static double[][] multiply(double[][] matrix1, double[][] matrix2){  
 double[][] result = new double[matrix1.length][matrix2[0].length];  
 for (int i = 0; i < matrix1.length; i++) {  
 for (int j = 0; j < matrix1[0].length; j++) {  
 double sum = 0;  
 for (int k = 0; k < matrix1[0].length; k++) {  
 sum += matrix1[i][k] \* matrix2[k][j];  
 }  
 result[i][j] = sum;  
 }  
 }  
 return result;  
 }  
  
 public static double[][] getBlock(double[][] matrix, int start, int finish, int cols){  
 double[][] result = new double[finish - start + 1][cols];  
 int resultIndex = 0;  
 for(int i = start; i < finish; i++){  
 double[] temp = new double[cols];  
 for(int j = 0; j < cols; j++){  
 temp[j] = matrix[i][j];  
 }  
 result[resultIndex] = temp;  
 resultIndex++;  
 }  
 return result;  
 }  
  
 public static double[][] addBlock(double[][] part, double[][] matrix, int start, int finish){  
 double[][] result = matrix;  
 int partRowIndex= 0;  
 for(int i = start; i < finish; i++){  
 for(int j = 0; j < matrix[0].length; j++){  
 result[i][j] = part[partRowIndex][j];  
 }  
 partRowIndex++;  
 }  
 return result;  
 }  
  
 public static void print(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();  
 }  
 }  
}

Результат виконання програми:

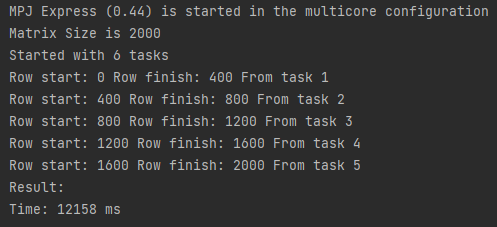


Рис. 1. Результат виконання Block.java

3.    Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями. **30 балів.**

Лістинг програми:

**Nonblock.java**

import mpi.\*;  
  
import static java.lang.System.*exit*;  
  
public class Nonblock {  
 private static int *matrixSize* = 2000;  
 private static int *MASTER* = 0;  
 private static int *FROM\_MASTER* = 1;  
 private static int *FROM\_WORKER* = 1;  
  
 public static void main(String[] args) throws MPIException {  
 double[][] matrixA = Methods.*generateMatrix*(*matrixSize*);  
 double[][] matrixB = Methods.*generateMatrix*(*matrixSize*);  
 double[][] result = new double[*matrixSize*][*matrixSize*];  
 long startTime = System.*currentTimeMillis*();  
 long endTime = System.*currentTimeMillis*();  
  
 MPI.*Init*(args);  
 int size = MPI.*COMM\_WORLD*.Size();  
 int rank = MPI.*COMM\_WORLD*.Rank();  
  
 int countOfWorkers = size - 1;  
  
 if(size < 2 || *matrixSize* < countOfWorkers){  
 System.*out*.println("Num Of Workers Is Small");  
 MPI.*COMM\_WORLD*.Abort(1);  
 *exit*(1);  
 }  
  
 if(rank == *MASTER*){  
 System.*out*.println("Matrix Size is " + *matrixSize*);  
 System.*out*.println("Started with " + size + " tasks");  
 int work = *matrixSize* / countOfWorkers;  
 int extra = *matrixSize* % countOfWorkers;  
 for(int worker = 1; worker <= countOfWorkers; worker++){  
 int rowStart = (worker - 1) \* work;  
 int rowFinish = rowStart + work;  
 if(worker == countOfWorkers){  
 rowFinish += extra;  
 }  
 Request send1 = MPI.*COMM\_WORLD*.Isend(new int[]{rowStart}, 0, 1, MPI.*INT*, worker, *FROM\_MASTER*);  
 Request send2 = MPI.*COMM\_WORLD*.Isend(new int[]{rowFinish}, 0, 1, MPI.*INT*, worker, *FROM\_MASTER*);  
  
 double[][] blockA = Methods.*getBlock*(matrixA, rowStart, rowFinish, *matrixSize*);  
  
 Request send3 = MPI.*COMM\_WORLD*.Isend(blockA, 0, blockA.length, MPI.*OBJECT*, worker, *FROM\_MASTER*);  
 Request send4 = MPI.*COMM\_WORLD*.Isend(matrixB, 0, matrixB.length, MPI.*OBJECT*, worker, *FROM\_MASTER*);  
  
 send1.Wait();  
 send2.Wait();  
 send3.Wait();  
 send4.Wait();  
 }  
  
 for(int worker = 1; worker <= countOfWorkers; worker++){  
 int[] rowStart = new int[1];  
 int[] rowFinish = new int[1];  
  
 Request rec1 = MPI.*COMM\_WORLD*.Irecv(rowStart, 0, 1, MPI.*INT*, worker, *FROM\_WORKER*);  
 Request rec2 = MPI.*COMM\_WORLD*.Irecv(rowFinish, 0, 1, MPI.*INT*, worker, *FROM\_WORKER*);  
  
 rec1.Wait();  
 rec2.Wait();  
  
 double[][] openedResult = new double[rowFinish[0] - rowStart[0] + 1][*matrixSize*];  
  
 Request rec3 = MPI.*COMM\_WORLD*.Irecv(openedResult, 0, openedResult.length, MPI.*OBJECT*, worker, *FROM\_WORKER*);  
 rec3.Wait();  
  
 Methods.*addBlock*(openedResult, result, rowStart[0], rowFinish[0]);  
 }  
 endTime = System.*currentTimeMillis*();  
 System.*out*.println("Result :");  
 //Methods.print(result);  
 System.*out*.println("Time: " +(endTime-startTime) + " ms");  
 } else {  
 int[] rowStart = new int[1];  
 int[] rowFinish = new int[1];  
 Request rec1 = MPI.*COMM\_WORLD*.Irecv(rowStart, 0, 1, MPI.*INT*, *MASTER*, *FROM\_MASTER*);  
 Request rec2 = MPI.*COMM\_WORLD*.Irecv(rowFinish, 0, 1, MPI.*INT*, *MASTER*, *FROM\_MASTER*);  
  
 rec1.Wait();  
 rec2.Wait();  
  
 double[][] openedBlockA = new double[rowFinish[0] - rowStart[0] + 1][*matrixSize*];  
 double[][] openedMatrixB = new double[*matrixSize*][*matrixSize*];  
  
 Request rec3 = MPI.*COMM\_WORLD*.Irecv(openedBlockA, 0, openedBlockA.length, MPI.*OBJECT*, *MASTER*, *FROM\_MASTER*);  
 Request rec4 = MPI.*COMM\_WORLD*.Irecv(openedMatrixB, 0, openedMatrixB.length, MPI.*OBJECT*, *MASTER*, *FROM\_MASTER*);  
  
 rec3.Wait();  
 rec4.Wait();  
 System.*out*.println("Row start: " + rowStart[0] + " Row finish: " + rowFinish[0] + " From task " + rank);  
  
 double[][] subResult = Methods.*multiply*(openedBlockA, openedMatrixB);  
  
 MPI.*COMM\_WORLD*.Send(rowStart, 0, 1, MPI.*INT*, *MASTER*, *FROM\_WORKER*);  
 MPI.*COMM\_WORLD*.Send(rowFinish, 0, 1, MPI.*INT*, *MASTER*, *FROM\_WORKER*);  
 MPI.*COMM\_WORLD*.Send(subResult, 0, subResult.length, MPI.*OBJECT*, *MASTER*, *FROM\_WORKER*);  
 }  
 MPI.*Finalize*();  
 }  
}

Результат виконання програми:

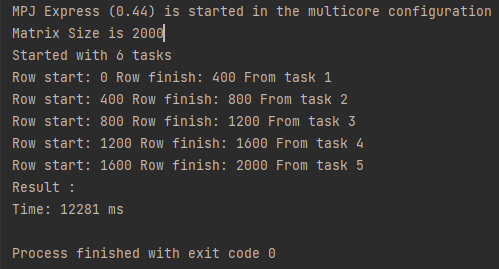


Рис. 2. Результат виконання Nonblock.java

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.**

Час виконання блокуючих методів

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naive | 4 | | | 6 | | | 8 | | |
| time | diff | eff | time | diff | eff | time | diff | eff |
| 500 | 0.151 | 0.281 | 0.537 | 0.134 | 0.313 | 0.48 | 0.08 | 0.308 | 0.49 | 0.061 |
| 1000 | 1.317 | 1.218 | 1.081 | 0.271 | 1.072 | 1.228 | 0.205 | 1.148 | 1.147 | 0.143 |
| 2000 | 41.09 | 18.38 | 2.236 | 0.559 | 12.01 | 3.421 | 0.57 | 11.06 | 3.715 | 0.464 |
| 3000 | 223.373 | 80.082 | 2.789 | 0.697 | 55.958 | 3.992 | 0.665 | 43.962 | 5.081 | 0.635 |

Графік прискорення блокуючих методів:

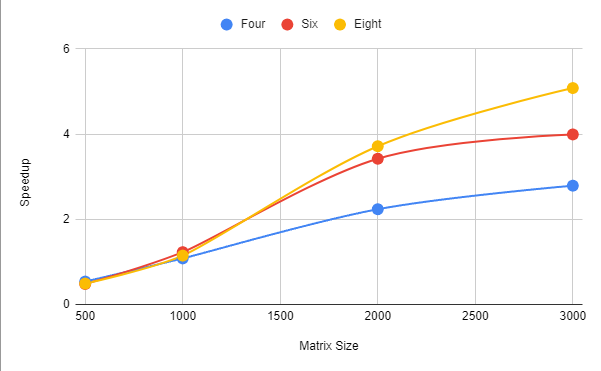


Рис. 3. Графік прискорення блокуючих методів

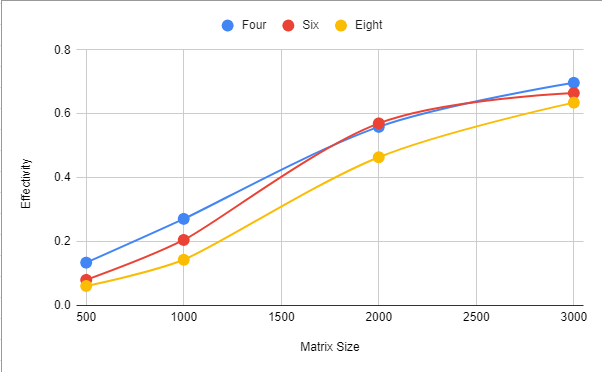
Графік ефективності блокуючих методів:  


Рис. 4. Графік ефективності блокуючих методів

Час виконання неблокуючих методів

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naive | 4 | | | 6 | | | 8 | | |
| time | diff | eff | time | diff | eff | time | diff | eff |
| 500 | 0.151 | 0.437 | 0.346 | 0.086 | 0.298 | 0.51 | 0.085 | 0.311 | 0.49 | 0.061 |
| 1000 | 1.317 | 1.116 | 1.18 | 0.295 | 1.198 | 1.099 | 0.183 | 1.177 | 1.119 | 0.14 |
| 2000 | 41.09 | 18.817 | 2.184 | 0.546 | 11.805 | 3.481 | 0.58 | 10.108 | 4.065 | 0.508 |
| 3000 | 223.373 | 87.018 | 2.567 | 0.642 | 55.559 | 4.02 | 0.67 | 45.528 | 4.906 | 0.613 |

Графік прискорення неблокуючих методів:

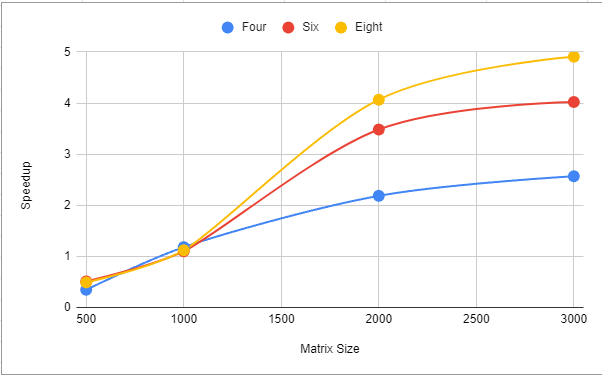


Рис. 5. Графік прискорення неблокуючих методів

Графік ефективності неблокуючих методів:

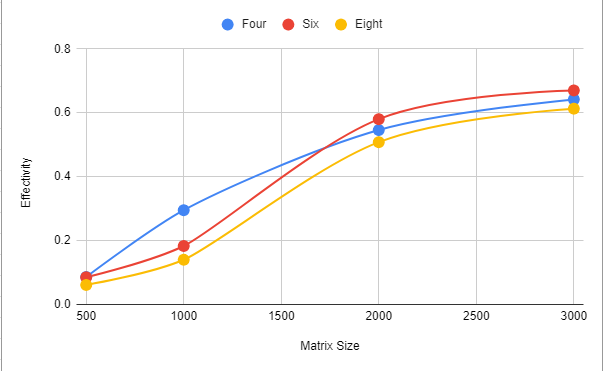


Рис. 6. Графік ефективності неблокуючих методів

Програмний код до всіх завдань наведений в репозиторії:

<https://github.com/whitetark/multithreading/tree/main/lab06>

**Висновок**

У ході виконання моєї лабораторної роботи я познайомився з методами блокуючого та неблокуючого обміну повідомленнями типу point-to-point та дослідив їх ефективність при збільшенні розміру матриць та при збільшенні кількості вузлів.