**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**

Комп‘ютерного практикуму № 6 з дисципліни

«Технології паралельних та розподілених обчислень»

**«Розробка паралельного алгоритму множення матриць з використанням МРІ-методів обміну повідомленнями**

**«один-до-одного» та дослідження його ефективності»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Корнієнко В.С.*

**Перевірив(ла)**

(прізвище, ім'я, по батькові)

*Стеценко І. В.*

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.**
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.**
5. **Ознайомитись з методами блокуючого та неблокуючого обміну повідомленнями типу point-to-point**

В MPI (Message Passing Interface) існує два типи обміну повідомленнями типу point-to-point: блокуючий та неблокуючий.

Блокуючий обмін повідомленнями:

* Для блокуючого обміну повідомленнями використовуються методи MPI.Send() та MPI.Recv().
* Блокуючий обмін означає, що відправник та отримувач повідомлення будуть заблоковані, доки обмін не буде завершений.
* Під час блокуючого виклику, викликаючий процес (відправник або отримувач) зупиняє своє виконання та очікує, поки повідомлення буде передано або отримано.

Неблокуючий обмін повідомленнями:

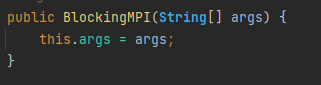
* Для неблокуючого обміну повідомленнями використовуються методи MPI.Isend() та MPI.Irecv().
* Неблокуючий обмін дозволяє продовжити виконання процесу негайно після виклику, навіть якщо обмін ще не завершений.
* Під час неблокуючого виклику, процес може продовжувати виконання і виконувати інші операції, не чекаючи завершення обміну повідомленнями.
* Щоб перевірити, чи завершений неблокуючий обмін, можна використовувати метод MPI.Test().

"Point-to-point" в даному контексті обміну вказує на передачу повідомлення між конкретним відправником та конкретним отримувачем. Це означає, що кожне повідомлення має одного відправника і одного отримувача.

1. **Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями (лістинг 1).**

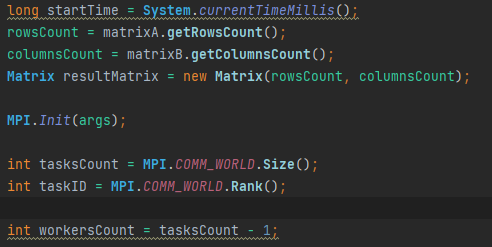
Для реалізації множення матриць з використанням блокуючого MPI був створений клас BlockingMPI.

Даний клас має конструктор, що приймає в себе вхідні аргументи програми, необхідні для ініціалізації MPI

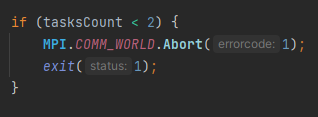


Для множення матриць клас має клас **multiply**(), що має наступну структуру:

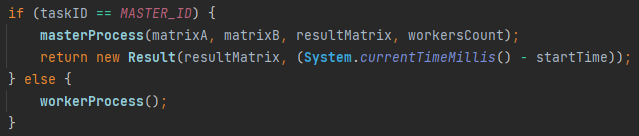
Спочатку ми ініціалізуємо всі змінні необхідні для подальшого множення матриць, ініціалізуємо MPI з аргументами командного рядка, збереженими в змінну args, дізнаємось загальну кількість задач у комунікаторі, ідентифікатор поточного процесу



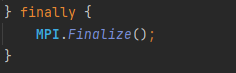
Далі ми перевіряємо чи достатня кількість процесів для роботи і якщо недостатня – то програма припиняє роботу.



Далі відбувається розподілення роботи між головним та робочими процесами, в залежності від ідентифікатора поточного процесу(Якщо поточний процес має ідентифікатор MASTER\_ID, то виконується метод masterProcess(), який керує роботою головного процесу, в іншому випадку – workerProcess()



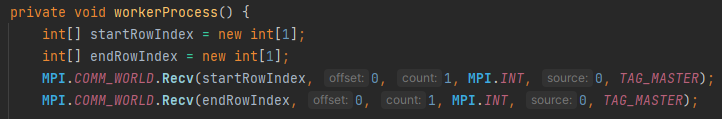
Далі відбувається завершення програми шляхом звільнення всіх ресурсів пов’язаних з комунікатором



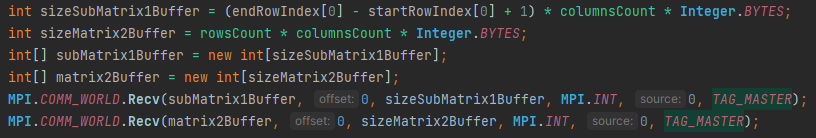
Блок try-finally в цьому коді використовується для забезпечення правильного вивільнення ресурсів MPI після завершення виконання програми або в разі виникнення виключення.

Тепер розглянемо безпосередньо **workerProcess**():

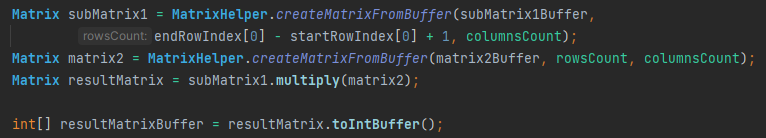
Метод починається з отриманням індексів початкового та кінцевого рядка від головного процесу за допомогою MPI.COMM\_WORLD.Recv().



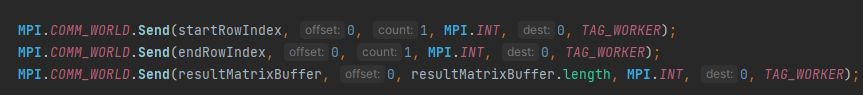
Далі обчислюється розмір буферів для зберігання підматриці та матриці 2, створюються буфери для отримання цих матриць та безпосередньо отримання даних підматриці та матриці 2 з головного процесу за допомогою MPI.COMM\_WORLD.Recv().



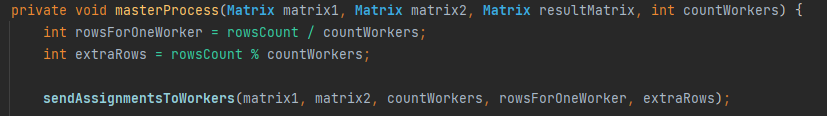
З буферів створюються об'єкти subMatrix1 та matrix2, виконується процес множення матриць та отримується результат resultMatrix, який в свою чергу конвертується назад в буфер.



В кінці цього методу Результати відправляються головному процесу за допомогою MPI.COMM\_WORLD.Send()



В **masterProcess**() спочатку обчислюється кількість рядків для одного робітника та додаткові рядки, викликається метод sendAssignmentsToWorkers() для розподілу завдань між робітниками

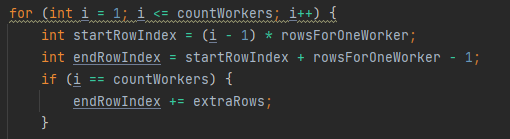


В кінці методу викликається метод receiveResultsFromWorkers() для отримання результатів від всіх робітників.

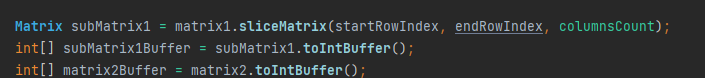


Метод **sendAssignmentsToWorkers**() має такі складові:

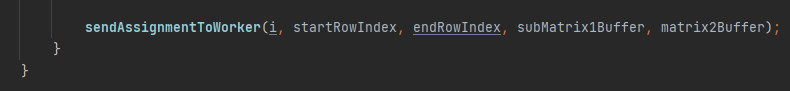
Для кожного робітника обчислюються індекси початкового та кінцевого рядка, якщо поточний робітник - останній, то до кінцевого рядка додаються додаткові рядки.



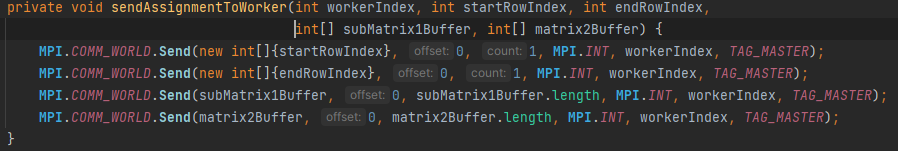
Створюється підматриця subMatrix1, яка разом з matrix2 в конвертується в буфери



Далі викликається метод sendAssignmentToWorker() для відправлення завдання робітнику.(такі операції виконуються для кожного робітника)

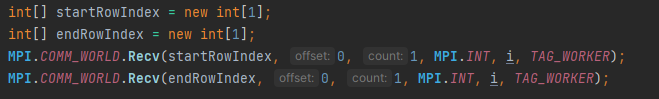


Метод **sendAssignmentToWorker** має одну мету: відправлення необхідних данних для робітника, використовуючи метод MPI.COMM\_WORLD.Send()

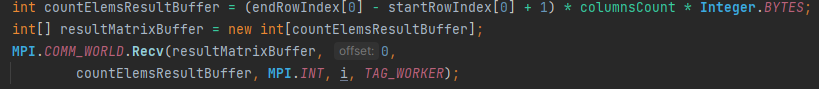


Останнім методом класу є **receiveResultsFromWorkers**. В ньому циклічно для кожного робітника виконуються наступні дії:

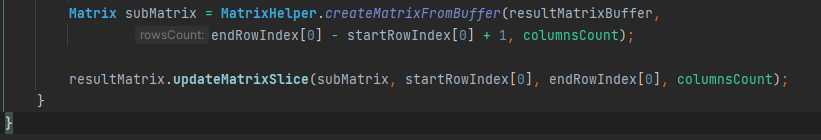
Спочатку отримуються індекси початкового та кінцевого рядка відповідного робітника за допомогою MPI.COMM\_WORLD.Recv().



Далі отримуються дані результату з робітника за допомогою MPI.COMM\_WORLD.Recv().



Після цього створюється підматриця subMatrix з отриманих даних буфера, оновлюється результуюча матриця шляхом оновлення відповідного сегмента матриці.

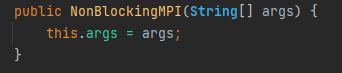


1. **Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями.**

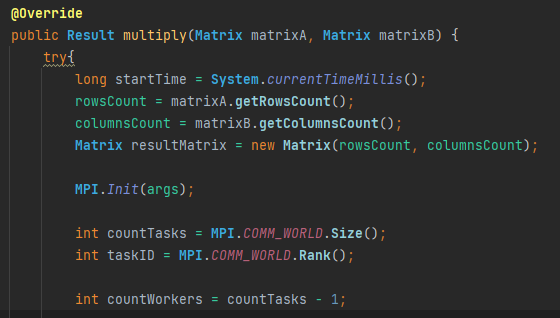
Для реалізації цього завдання був створений клас NonBlockingMPI.

*(Оскільки даний пункт має схожу специфіку з завданням 2, буду описувати не так детально як в завданні 2)*

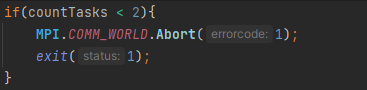
Клас містить декілька константних полів, які використовуються як теги для розпізнавання повідомлень між майстер-процесом і робітниками. args - це масив аргументів командного рядка. Клас також має конструктор аналогічний до конструктора BlockingMPI



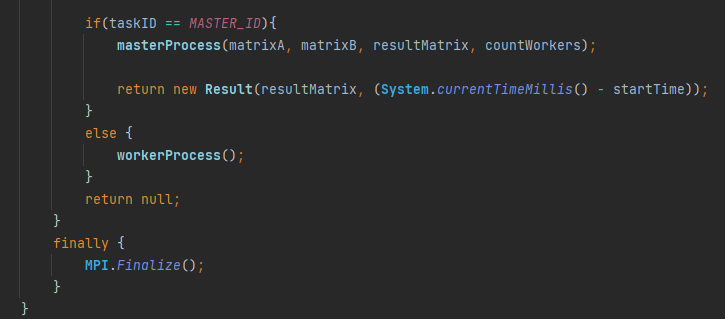
Метод **multiply** виконує множення матриць matrixA і matrixB. Він ініціалізує змінні countRows і countColumn з відповідними розмірами матриць. Далі він ініціалізує MPI і отримує кількість задач і ідентифікатор поточного процесу.



Якщо кількість задач менше 2, то майстер-процес викликає MPI.COMM\_WORLD.Abort(1) та виходимо з програми за допомогою exit(1).

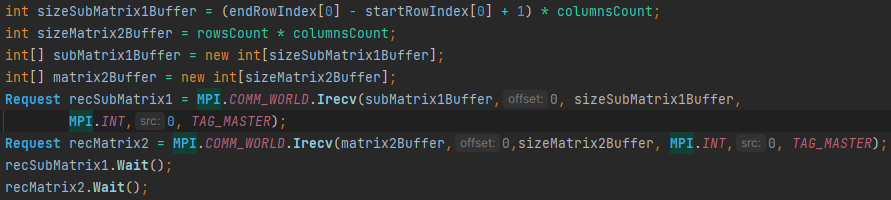


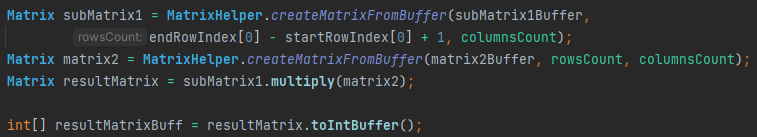
Якщо поточний процес є майстер-процесом, то викликається метод masterProcess, в іншому випадку викликається метод workerProcess. Після завершення роботи метод повертає null. У блоку finally викликається MPI.Finalize() для завершення роботи з MPI.

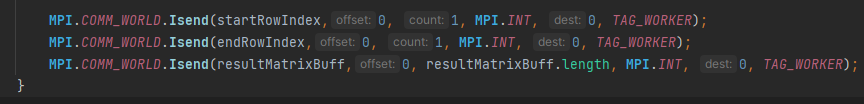


Метод **workerProcess** виконує обчислення в робітничому процесі. Він очікує прийом початкового та кінцевого індексів рядків матриці від майстер-процесу, а також отримує підматрицю subMatrix1 та matrix2.

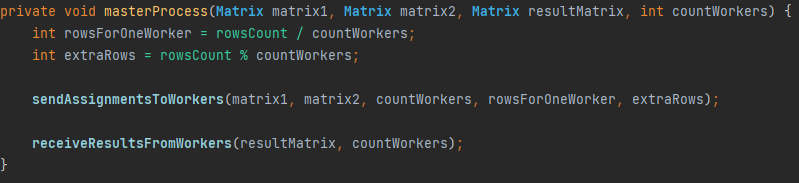


Далі відбувається множення subMatrix1 на matrix2, в результаті якого отримуємо resultMatrix.

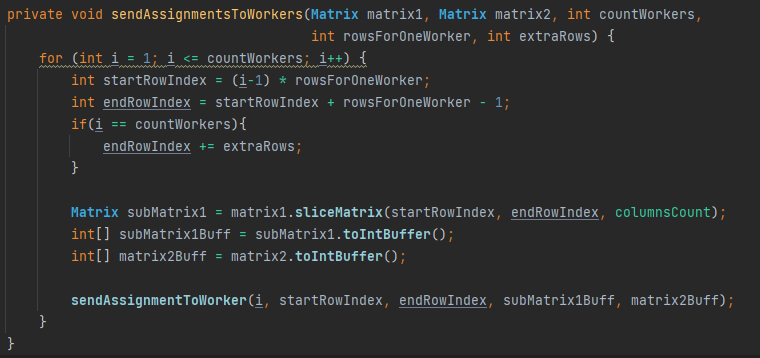
 Результати (початковий і кінцевий індекси рядків та resultMatrix) надсилаються майстер-процесу за допомогою MPI.COMM\_WORLD.Isend().



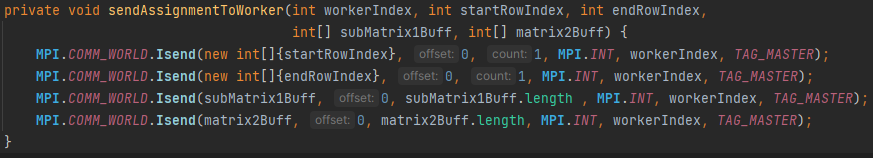
Метод **masterProcess** виконує обчислення в майстер-процесі. Він розподіляє завдання між робітничими процесами, відправляючи їм відповідні підматриці для обчислення. Після цього отримує результати від робітничих процесів і оновлює resultMatrix.



Метод **sendAssignmentsToWorkers** відправляє підматриці subMatrix1 та matrix2 робітничим процесам. Для кожного робітничого процесу обчислюються початковий та кінцевий індекси рядків матриці, а також створюються буфери subMatrix1Buff та matrix2Buff для передачі даних.



Метод **sendAssignmentToWorker** має наступний вигляд:



Метод **receiveResultsFromWorkers** отримує результати обчислень від робітничих процесів і оновлює відповідні частини resultMatrix. Він отримує початковий і кінцевий індекси рядків та буфер resultMatrixBuff, створений робітничим процесом. Після цього використовуючи MatrixHelper.createMatrixFromBuffer, створюється підматриця subMatrix, яка потім оновлює відповідну частину resultMatrix.



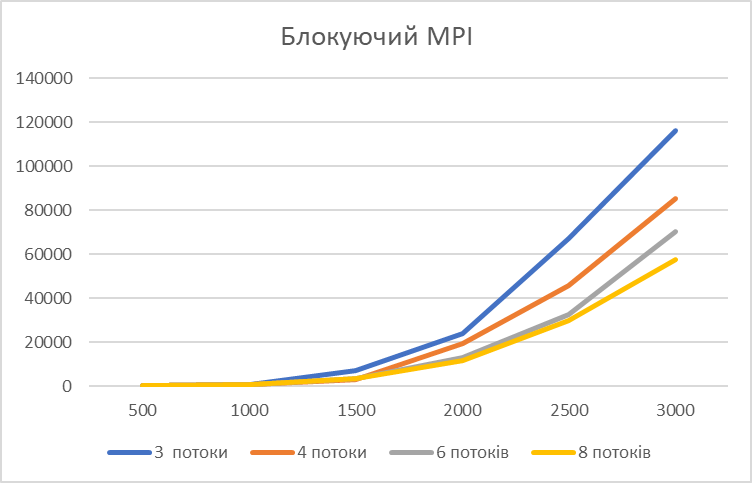
1. **Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями.**

Для дослідження ефективності блокуючого та неблокуючого методів обміну повідомленнями проведемо ряд тестів, варіюючи при цьому кількість потоків та розмірності матриць

В результаті досліджень були отримані наступні результати:

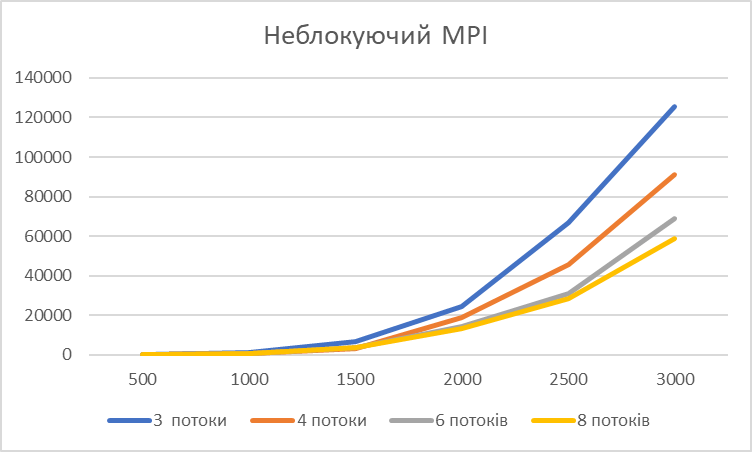
**Для блокуючого MPI:**





**Для неблокуючого MPI:**



****

Порівнюючи результати виконання множення матриць за допомогою блокуючого MPI і за допомогою неблокуючого MPI, можна зробити наступні спостереження:

* Загалом, результати множення матриць за допомогою неблокуючого MPI показують менші значення часу виконання порівняно з блокуючим MPI для більшості комбінацій розмірності матриць та кількості потоків. Це свідчить про те, що неблокуючий MPI може бути ефективнішим при виконанні обчислювально інтенсивних завдань, таких як множення матриць.
* Зафіксована кількість потоків показує, що неблокуючий MPI зазвичай дає менші значення часу виконання навіть при збільшенні розмірності матриці. Це пов'язано з ефективнішим управлінням комунікаціями та використанням ресурсів при виконанні операцій над матрицями.
* При збільшенні кількості потоків спостерігається загальне зменшення часу виконання для обох варіантів MPI. Однак, неблокуючий MPI все ще зазвичай дає менші значення часу виконання навіть при більшій кількості потоків.
* Знову ж таки, великі розмірності матриць (2000, 2500, 3000) та більша кількість потоків (6, 8) показують найбільше зменшення часу виконання для неблокуючого MPI. Це підкреслює його ефективність при обробці великих обсягів даних та використання більшої обчислювальної потужності.

Враховуючи ці спостереження, можна зробити висновок, що неблокуючий MPI є більш ефективним і швидким для виконання множення матриць у порівнянні з блокуючим MPI, особливо при великих розмірностях матриць та більшій кількості потоків.

**Лістинг коду:**

**BlockingMPI.java**

import **mpi.**\*;  
  
import static **java.lang.System**.*exit*;  
  
public class **BlockingMPI** implements IMatrixMultiplicationAlgorithm {  
 private static final int *TAG\_MASTER* = 1;  
 private static final int *TAG\_WORKER* = 2;  
 private static final int *MASTER\_ID* = 0;  
 private final **String**[] args;  
 private int columnsCount;  
 private int rowsCount;  
 public BlockingMPI(**String**[] args) {  
 this.args = args;  
 }  
  
 **@Override** public **Result** multiply(**Matrix** matrixA, **Matrix** matrixB) {  
 try {  
 long startTime = **System**.*currentTimeMillis*();  
 rowsCount = matrixA.**getRowsCount**();  
 columnsCount = matrixB.**getColumnsCount**();  
 **Matrix** resultMatrix = new **Matrix**(rowsCount, columnsCount);  
  
 **MPI**.*Init*(args);  
  
 int tasksCount = **MPI**.*COMM\_WORLD*.**Size**();  
 int taskID = **MPI**.*COMM\_WORLD*.**Rank**();  
  
 int workersCount = tasksCount - 1;  
  
 if (tasksCount < 2) {  
 **MPI**.*COMM\_WORLD*.**Abort**(1);  
 *exit*(1);  
 }  
  
 if (taskID == *MASTER\_ID*) {  
 **masterProcess**(matrixA, matrixB, resultMatrix, workersCount);  
 return new **Result**(resultMatrix, (**System**.*currentTimeMillis*() - startTime));  
 } else {  
 **workerProcess**();  
 }  
 return null;  
 } finally {  
 **MPI**.*Finalize*();  
 }  
 }  
  
  
 private void workerProcess() {  
 int[] startRowIndex = new int[1];  
 int[] endRowIndex = new int[1];  
 **MPI**.*COMM\_WORLD*.**Recv**(startRowIndex, 0, 1, **MPI**.*INT*, 0, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Recv**(endRowIndex, 0, 1, **MPI**.*INT*, 0, *TAG\_MASTER*);  
  
 int sizeSubMatrix1Buffer = (endRowIndex[0] - startRowIndex[0] + 1) \* columnsCount \* **Integer**.*BYTES*;  
 int sizeMatrix2Buffer = rowsCount \* columnsCount \* **Integer**.*BYTES*;  
 int[] subMatrix1Buffer = new int[sizeSubMatrix1Buffer];  
 int[] matrix2Buffer = new int[sizeMatrix2Buffer];  
 **MPI**.*COMM\_WORLD*.**Recv**(subMatrix1Buffer, 0, sizeSubMatrix1Buffer, **MPI**.*INT*, 0, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Recv**(matrix2Buffer, 0, sizeMatrix2Buffer, **MPI**.*INT*, 0, *TAG\_MASTER*);  
  
 **Matrix** subMatrix1 = **MatrixHelper**.*createMatrixFromBuffer*(subMatrix1Buffer,  
 endRowIndex[0] - startRowIndex[0] + 1, columnsCount);  
 **Matrix** matrix2 = **MatrixHelper**.*createMatrixFromBuffer*(matrix2Buffer, rowsCount, columnsCount);  
 **Matrix** resultMatrix = subMatrix1.**multiply**(matrix2);  
  
 int[] resultMatrixBuffer = resultMatrix.**toIntBuffer**();  
  
 **MPI**.*COMM\_WORLD*.**Send**(startRowIndex, 0, 1, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 **MPI**.*COMM\_WORLD*.**Send**(endRowIndex, 0, 1, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 **MPI**.*COMM\_WORLD*.**Send**(resultMatrixBuffer, 0, resultMatrixBuffer.length, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 }  
  
 private void masterProcess(**Matrix** matrix1, **Matrix** matrix2, **Matrix** resultMatrix, int countWorkers) {  
 int rowsForOneWorker = rowsCount / countWorkers;  
 int extraRows = rowsCount % countWorkers;  
  
 **sendAssignmentsToWorkers**(matrix1, matrix2, countWorkers, rowsForOneWorker, extraRows);  
  
 **receiveResultsFromWorkers**(resultMatrix, countWorkers);  
 }  
  
 private void receiveResultsFromWorkers(**Matrix** resultMatrix, int countWorkers) {  
 for (int i = 1; i <= countWorkers; i++) {  
 int[] startRowIndex = new int[1];  
 int[] endRowIndex = new int[1];  
 **MPI**.*COMM\_WORLD*.**Recv**(startRowIndex, 0, 1, **MPI**.*INT*, i, *TAG\_WORKER*);  
 **MPI**.*COMM\_WORLD*.**Recv**(endRowIndex, 0, 1, **MPI**.*INT*, i, *TAG\_WORKER*);  
  
 int countElemsResultBuffer = (endRowIndex[0] - startRowIndex[0] + 1) \* columnsCount \* **Integer**.*BYTES*;  
 int[] resultMatrixBuffer = new int[countElemsResultBuffer];  
 **MPI**.*COMM\_WORLD*.**Recv**(resultMatrixBuffer, 0,  
 countElemsResultBuffer, **MPI**.*INT*, i, *TAG\_WORKER*);  
 **Matrix** subMatrix = **MatrixHelper**.*createMatrixFromBuffer*(resultMatrixBuffer,  
 endRowIndex[0] - startRowIndex[0] + 1, columnsCount);  
  
 resultMatrix.**updateMatrixSlice**(subMatrix, startRowIndex[0], endRowIndex[0], columnsCount);  
 }  
 }  
  
 private void sendAssignmentsToWorkers(**Matrix** matrix1, **Matrix** matrix2, int countWorkers, int rowsForOneWorker, int extraRows) {  
 for (int i = 1; i <= countWorkers; i++) {  
 int startRowIndex = (i - 1) \* rowsForOneWorker;  
 int endRowIndex = startRowIndex + rowsForOneWorker - 1;  
 if (i == countWorkers) {  
 endRowIndex += extraRows;  
 }  
  
 **Matrix** subMatrix1 = matrix1.**sliceMatrix**(startRowIndex, endRowIndex, columnsCount);  
 int[] subMatrix1Buffer = subMatrix1.**toIntBuffer**();  
 int[] matrix2Buffer = matrix2.**toIntBuffer**();  
  
 **sendAssignmentToWorker**(i, startRowIndex, endRowIndex, subMatrix1Buffer, matrix2Buffer);  
 }  
 }  
  
 private void sendAssignmentToWorker(int workerIndex, int startRowIndex, int endRowIndex,  
 int[] subMatrix1Buffer, int[] matrix2Buffer) {  
 **MPI**.*COMM\_WORLD*.**Send**(new int[]{startRowIndex}, 0, 1, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Send**(new int[]{endRowIndex}, 0, 1, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Send**(subMatrix1Buffer, 0, subMatrix1Buffer.length, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Send**(matrix2Buffer, 0, matrix2Buffer.length, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 }  
}

**NonBlockingMPI.java**

import **mpi.MPI**;  
import **mpi.Request**;  
  
import static **java.lang.System**.*exit*;  
  
public class **NonBlockingMPI** implements IMatrixMultiplicationAlgorithm {  
  
 private static final int *TAG\_MASTER* = 1;  
 private static final int *TAG\_WORKER* = 2;  
 private static final int *MASTER\_ID* = 0;  
  
 private final **String**[] args;  
 private int columnsCount;  
 private int rowsCount;  
  
 public NonBlockingMPI(**String**[] args) {  
 this.args = args;  
 }  
  
 **@Override** public **Result** multiply(**Matrix** matrixA, **Matrix** matrixB) {  
 try{  
 long startTime = **System**.*currentTimeMillis*();  
 rowsCount = matrixA.**getRowsCount**();  
 columnsCount = matrixB.**getColumnsCount**();  
 **Matrix** resultMatrix = new **Matrix**(rowsCount, columnsCount);  
  
 **MPI**.*Init*(args);  
  
 int countTasks = **MPI**.*COMM\_WORLD*.**Size**();  
 int taskID = **MPI**.*COMM\_WORLD*.**Rank**();  
  
 int countWorkers = countTasks - 1;  
  
 if(countTasks < 2){  
 **MPI**.*COMM\_WORLD*.**Abort**(1);  
 *exit*(1);  
 }  
  
 if(taskID == *MASTER\_ID*){  
 **masterProcess**(matrixA, matrixB, resultMatrix, countWorkers);  
  
 return new **Result**(resultMatrix, (**System**.*currentTimeMillis*() - startTime));  
 }  
 else {  
 **workerProcess**();  
 }  
 return null;  
 }  
 finally {  
 **MPI**.*Finalize*();  
 }  
 }  
  
 private void workerProcess() {  
 int[] startRowIndex = new int[1];  
 int[] endRowIndex = new int[1];  
 **Request** recStartIndex = **MPI**.*COMM\_WORLD*.**Irecv**(startRowIndex,0,1, **MPI**.*INT*, 0, *TAG\_MASTER*);  
 **Request** recEndIndex = **MPI**.*COMM\_WORLD*.**Irecv**(endRowIndex,0,1, **MPI**.*INT*, 0, *TAG\_MASTER*);  
 recStartIndex.**Wait**();  
 recEndIndex.**Wait**();  
  
 int sizeSubMatrix1Buffer = (endRowIndex[0] - startRowIndex[0] + 1) \* columnsCount;  
 int sizeMatrix2Buffer = rowsCount \* columnsCount;  
 int[] subMatrix1Buffer = new int[sizeSubMatrix1Buffer];  
 int[] matrix2Buffer = new int[sizeMatrix2Buffer];  
 **Request** recSubMatrix1 = **MPI**.*COMM\_WORLD*.**Irecv**(subMatrix1Buffer,0, sizeSubMatrix1Buffer,  
 **MPI**.*INT*,0, *TAG\_MASTER*);  
 **Request** recMatrix2 = **MPI**.*COMM\_WORLD*.**Irecv**(matrix2Buffer,0,sizeMatrix2Buffer, **MPI**.*INT*,0, *TAG\_MASTER*);  
 recSubMatrix1.**Wait**();  
 recMatrix2.**Wait**();  
  
 **Matrix** subMatrix1 = **MatrixHelper**.*createMatrixFromBuffer*(subMatrix1Buffer,  
 endRowIndex[0] - startRowIndex[0] + 1, columnsCount);  
 **Matrix** matrix2 = **MatrixHelper**.*createMatrixFromBuffer*(matrix2Buffer, rowsCount, columnsCount);  
 **Matrix** resultMatrix = subMatrix1.**multiply**(matrix2);  
  
 int[] resultMatrixBuff = resultMatrix.**toIntBuffer**();  
  
 **MPI**.*COMM\_WORLD*.**Isend**(startRowIndex,0, 1, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 **MPI**.*COMM\_WORLD*.**Isend**(endRowIndex,0, 1, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 **MPI**.*COMM\_WORLD*.**Isend**(resultMatrixBuff,0, resultMatrixBuff.length, **MPI**.*INT*, 0, *TAG\_WORKER*);  
 }  
  
 private void masterProcess(**Matrix** matrix1, **Matrix** matrix2, **Matrix** resultMatrix, int countWorkers) {  
 int rowsForOneWorker = rowsCount / countWorkers;  
 int extraRows = rowsCount % countWorkers;  
  
 **sendAssignmentsToWorkers**(matrix1, matrix2, countWorkers, rowsForOneWorker, extraRows);  
  
 **receiveResultsFromWorkers**(resultMatrix, countWorkers);  
 }  
  
 private void sendAssignmentsToWorkers(**Matrix** matrix1, **Matrix** matrix2, int countWorkers,  
 int rowsForOneWorker, int extraRows) {  
 for (int i = 1; i <= countWorkers; i++) {  
 int startRowIndex = (i-1) \* rowsForOneWorker;  
 int endRowIndex = startRowIndex + rowsForOneWorker - 1;  
 if(i == countWorkers){  
 endRowIndex += extraRows;  
 }  
  
 **Matrix** subMatrix1 = matrix1.**sliceMatrix**(startRowIndex, endRowIndex, columnsCount);  
 int[] subMatrix1Buff = subMatrix1.**toIntBuffer**();  
 int[] matrix2Buff = matrix2.**toIntBuffer**();  
  
 **sendAssignmentToWorker**(i, startRowIndex, endRowIndex, subMatrix1Buff, matrix2Buff);  
 }  
 }  
  
 private void receiveResultsFromWorkers(**Matrix** resultMatrix, int countWorkers) {  
 for (int i = 1; i <= countWorkers; i++) {  
 int[] startRowIndex = new int[1];  
 int[] endRowIndex = new int[1];  
  
 **Request** recStartIndex = **MPI**.*COMM\_WORLD*.**Irecv**(startRowIndex,0,1, **MPI**.*INT*, i, *TAG\_WORKER*);  
 **Request** recEndIndex =**MPI**.*COMM\_WORLD*.**Irecv**(endRowIndex,0,1, **MPI**.*INT*, i, *TAG\_WORKER*);  
 recStartIndex.**Wait**();  
 recEndIndex.**Wait**();  
  
 int resultBufferElementsCount = (endRowIndex[0] - startRowIndex[0] + 1) \* columnsCount;  
 int[] resultMatrixBuff = new int[resultBufferElementsCount];  
  
 **Request** recRes = **MPI**.*COMM\_WORLD*.**Irecv**(resultMatrixBuff,0, resultBufferElementsCount ,  
 **MPI**.*INT*, i, *TAG\_WORKER*);  
 recRes.**Wait**();  
  
 **Matrix** subMatrix = **MatrixHelper**.*createMatrixFromBuffer*(resultMatrixBuff,  
 endRowIndex[0] - startRowIndex[0] + 1, columnsCount);  
 resultMatrix.**updateMatrixSlice**(subMatrix, startRowIndex[0], endRowIndex[0], columnsCount);  
 }  
 }  
  
 private void sendAssignmentToWorker(int workerIndex, int startRowIndex, int endRowIndex,  
 int[] subMatrix1Buff, int[] matrix2Buff) {  
 **MPI**.*COMM\_WORLD*.**Isend**(new int[]{startRowIndex}, 0, 1, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Isend**(new int[]{endRowIndex}, 0, 1, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Isend**(subMatrix1Buff, 0, subMatrix1Buff.length , **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 **MPI**.*COMM\_WORLD*.**Isend**(matrix2Buff, 0, matrix2Buff.length, **MPI**.*INT*, workerIndex, *TAG\_MASTER*);  
 }  
}

**MPIMatrixMultiplication.java**

import **mpi.MPI**;  
  
public class **MPIMatrixMultiplication** {  
 public static void main(**String**[] args) {  
 int size = 3000;  
 boolean blocking = false;  
 **Matrix** matrix1 = **MatrixHelper**.*generateRandomMatrix*(size);  
 **Matrix** matrix2 = **MatrixHelper**.*generateRandomMatrix*(size);  
  
 if (blocking){  
 **BlockingMPI** blockingMPI = new **BlockingMPI**(args);  
 **Result** blockingMPIResult = blockingMPI.**multiply**(matrix1, matrix2);  
 if (blockingMPIResult == null) {  
 return;  
 }  
 **System**.*out*.**println**(**"Blocking MPI: "**);  
 **System**.*out*.**println**(**"Matrix size: "** + size);  
 **System**.*out*.**println**(**"Processors count: "** + **MPI**.*COMM\_WORLD*.**Size**());  
// System.out.println(blockingMPIResult.getResultMatrix().equals(matrix1.multiply(matrix2)) ?  
// "Result is Correct" : "Result is Incorrect");  
 **System**.*out*.**println**(**"Total time: "** + blockingMPIResult.**getTotalTime**());  
 } else {  
 **NonBlockingMPI** nonBlockingMPI = new **NonBlockingMPI**(args);  
 **Result** nonBlockingMPIResult = nonBlockingMPI.**multiply**(matrix1, matrix2);  
 if (nonBlockingMPIResult == null) {  
 return;  
 }  
  
 **System**.*out*.**println**(**"Non-Blocking MPI: "**);  
 **System**.*out*.**println**(**"Matrix size: "** + size);  
 **System**.*out*.**println**(**"Processors count: "** + **MPI**.*COMM\_WORLD*.**Size**());  
// System.out.println(nonBlockingMPIResult.getResultMatrix().equals(matrix1.multiply(matrix2)) ?  
// "Result is Correct" : "Result is Incorrect");  
 **System**.*out*.**println**(**"Total time: "** + nonBlockingMPIResult.**getTotalTime**());  
 }  
  
 }  
}