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 MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності»

Виконав(ла) *ІП-01 Корнієнко В.С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) *Стеценко І. В.* _____
(прізвище, ім'я, по батькові)

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.**

1. **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.

2. **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

```
public BlockingMPI(String[] args) {
    this.args = args;
}
```

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

```
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;
```

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

```
if (tasksCount < 2) {
    MPI.COMM_WORLD.Abort( errorcode: 1);
    exit( status: 1);
}
```

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()

```
if (taskID == MASTER_ID) {
    masterProcess(matrixA, matrixB, resultMatrix, workersCount);
    return new Result(resultMatrix, (System.currentTimeMillis() - startTime));
} else {
    workerProcess();
}
```

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

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

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().

```
private void workerProcess() {
    int[] startRowIndex = new int[1];
    int[] endRowIndex = new int[1];
    MPI.COMM_WORLD.Recv(startRowIndex, offset: 0, count: 1, MPI.INT, source: 0, TAG_MASTER);
    MPI.COMM_WORLD.Recv(endRowIndex, offset: 0, count: 1, MPI.INT, source: 0, TAG_MASTER);
```

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().

```
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, offset: 0, sizeSubMatrix1Buffer, MPI.INT, source: 0, TAG_MASTER);
MPI.COMM_WORLD.Recv(matrix2Buffer, offset: 0, sizeMatrix2Buffer, MPI.INT, source: 0, TAG_MASTER);
```

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

```
Matrix subMatrix1 = MatrixHelper.createMatrixFromBuffer(subMatrix1Buffer,
        rowsCount: endRowIndex[0] - startRowIndex[0] + 1, columnsCount);
Matrix matrix2 = MatrixHelper.createMatrixFromBuffer(matrix2Buffer, rowsCount, columnsCount);
Matrix resultMatrix = subMatrix1.multiply(matrix2);

int[] resultMatrixBuffer = resultMatrix.toIntBuffer();
```

В кінці цього методу Результати відправляються головному процесу за допомогою MPI.COMM_WORLD.Send()

```
MPI.COMM_WORLD.Send(startRowIndex, offset: 0, count: 1, MPI.INT, dest: 0, TAG_WORKER);
MPI.COMM_WORLD.Send(endRowIndex, offset: 0, count: 1, MPI.INT, dest: 0, TAG_WORKER);
MPI.COMM_WORLD.Send(resultMatrixBuffer, offset: 0, resultMatrixBuffer.length, MPI.INT, dest: 0, TAG_WORKER);
```

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

```
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);
```

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

```
    receiveResultsFromWorkers(resultMatrix, countWorkers);
}
```

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.

```
for (int i = 1; i <= countWorkers; i++) {
    int startRowIndex = (i - 1) * rowsForOneWorker;
    int endRowIndex = startRowIndex + rowsForOneWorker - 1;
    if (i == countWorkers) {
        endRowIndex += extraRows;
    }
```

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

```
Matrix subMatrix1 = matrix1.sliceMatrix(startRowIndex, endRowIndex, columnsCount);
int[] subMatrix1Buffer = subMatrix1.toIntBuffer();
int[] matrix2Buffer = matrix2.toIntBuffer();
```

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

```
            sendAssignmentToWorker(i, startRowIndex, endRowIndex, subMatrix1Buffer, matrix2Buffer);
        }
    }
```

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

```
private void sendAssignmentToWorker(int workerIndex, int startRowIndex, int endRowIndex,
                                    int[] subMatrix1Buffer, int[] matrix2Buffer) {
    MPI.COMM_WORLD.Send(new int[]{startRowIndex}, offset: 0, count: 1, MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Send(new int[]{endRowIndex}, offset: 0, count: 1, MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Send(subMatrix1Buffer, offset: 0, subMatrix1Buffer.length, MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Send(matrix2Buffer, offset: 0, matrix2Buffer.length, MPI.INT, workerIndex, TAG_MASTER);
}
```

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().

```
int[] startRowIndex = new int[1];
int[] endRowIndex = new int[1];
MPI.COMM_WORLD.Recv(startRowIndex, offset: 0, count: 1, MPI.INT, i, TAG_WORKER);
MPI.COMM_WORLD.Recv(endRowIndex, offset: 0, count: 1, MPI.INT, i, TAG_WORKER);
```

Далі отримуються дані результату з робітника за допомогою MPI.COMM_WORLD.Recv().

```
int countElemsResultBuffer = (endRowIndex[0] - startRowIndex[0] + 1) * columnsCount * Integer.BYTES;
int[] resultMatrixBuffer = new int[countElemsResultBuffer];
MPI.COMM_WORLD.Recv(resultMatrixBuffer, offset: 0,
        countElemsResultBuffer, MPI.INT, i, TAG_WORKER);
```

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

```
        Matrix subMatrix = MatrixHelper.createMatrixFromBuffer(resultMatrixBuffer,
                rowsCount: endRowIndex[0] - startRowIndex[0] + 1, columnsCount);

        resultMatrix.updateMatrixSlice(subMatrix, startRowIndex[0], endRowIndex[0], columnsCount);
    }
}
```

3. **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

```
public NonBlockingMPI(String[] args) {
    this.args = args;
}
```

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.

```
@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;
```

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

```
if(countTasks < 2){
    MPI.COMM_WORLD.Abort( errorcode: 1);
    exit( status: 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.

```
        if(taskID == MASTER_ID){
            masterProcess(matrixA, matrixB, resultMatrix, countWorkers);

            return new Result(resultMatrix, (System.currentTimeMillis() - startTime));
        }
        else {
            workerProcess();
        }
        return null;
    }
    finally {
        MPI.Finalize();
    }
}
```

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.

```
private void workerProcess() {
    int[] startRowIndex = new int[1];
    int[] endRowIndex = new int[1];
    Request recStartIndex = MPI.COMM_WORLD.Irecv(startRowIndex, offset: 0, count: 1, MPI.INT, src: 0, TAG_MASTER);
    Request recEndIndex = MPI.COMM_WORLD.Irecv(endRowIndex, offset: 0, count: 1, MPI.INT, src: 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, offset: 0, sizeSubMatrix1Buffer,
        MPI.INT, src: 0, TAG_MASTER);
Request recMatrix2 = MPI.COMM_WORLD.Irecv(matrix2Buffer, offset: 0,sizeMatrix2Buffer, MPI.INT, src: 0, TAG_MASTER);
recSubMatrix1.Wait();
recMatrix2.Wait();
```

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

```
Matrix subMatrix1 = MatrixHelper.createMatrixFromBuffer(subMatrix1Buffer,
        rowsCount: endRowIndex[0] - startRowIndex[0] + 1, columnsCount);
Matrix matrix2 = MatrixHelper.createMatrixFromBuffer(matrix2Buffer, rowsCount, columnsCount);
Matrix resultMatrix = subMatrix1.multiply(matrix2);

int[] resultMatrixBuff = resultMatrix.toIntBuffer();
```

Результати (початковий і кінцевий індекси рядків та resultMatrix)

надсилаються майстер-процесу за допомогою MPI.COMM_WORLD.Isend().

```
    MPI.COMM_WORLD.Isend(startRowIndex, offset: 0,  count: 1, MPI.INT,  dest: 0, TAG_WORKER);
    MPI.COMM_WORLD.Isend(endRowIndex, offset: 0,  count: 1, MPI.INT,  dest: 0, TAG_WORKER);
    MPI.COMM_WORLD.Isend(resultMatrixBuff, offset: 0, resultMatrixBuff.length, MPI.INT,  dest: 0, TAG_WORKER);
}
```

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.

```
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);
}
```

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.

```
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);
    }
}
```

Метод **sendAssignmentToWorker** має наступний вигляд:

```java
private void sendAssignmentToWorker(int workerIndex, int startRowIndex, int endRowIndex,
                                    int[] subMatrix1Buff, int[] matrix2Buff) {
    MPI.COMM_WORLD.Isend(new int[]{startRowIndex}, offset: 0, count: 1, MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Isend(new int[]{endRowIndex}, offset: 0, count: 1, MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Isend(subMatrix1Buff, offset: 0, subMatrix1Buff.length , MPI.INT, workerIndex, TAG_MASTER);
    MPI.COMM_WORLD.Isend(matrix2Buff, offset: 0, matrix2Buff.length, MPI.INT, workerIndex, TAG_MASTER);
}
```

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.

```java
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, offset: 0, count: 1, MPI.INT, i, TAG_WORKER);
        Request recEndIndex =MPI.COMM_WORLD.Irecv(endRowIndex, offset: 0, count: 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, offset: 0, resultBufferElementsCount ,
                MPI.INT, i, TAG_WORKER);
        recRes.Wait();

        Matrix subMatrix = MatrixHelper.createMatrixFromBuffer(resultMatrixBuff,
                rowsCount: endRowIndex[0] - startRowIndex[0] + 1, columnsCount);
        resultMatrix.updateMatrixSlice(subMatrix, startRowIndex[0], endRowIndex[0], columnsCount);
    }
}
```

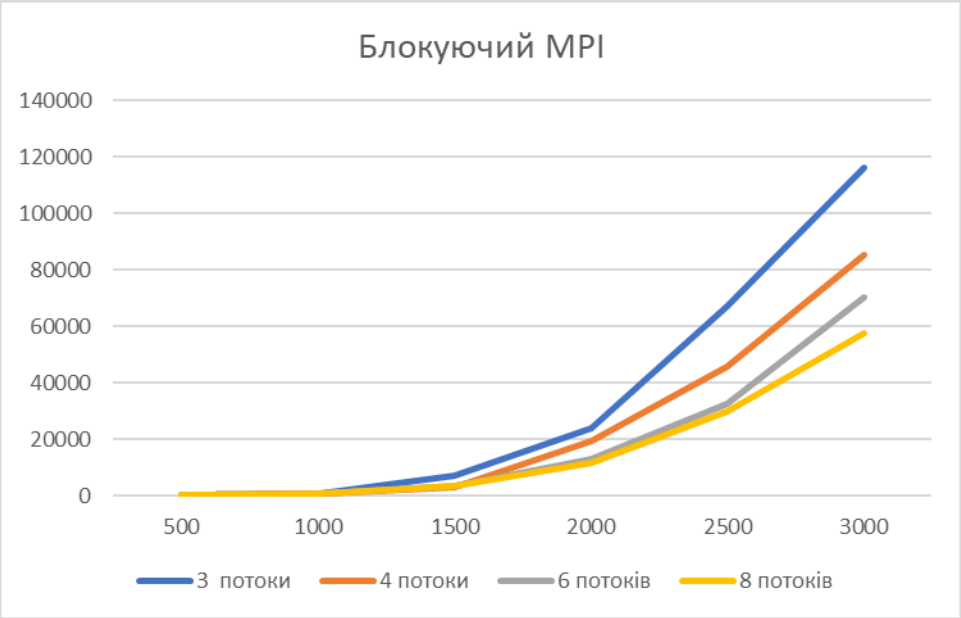
4. **Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями.**

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:

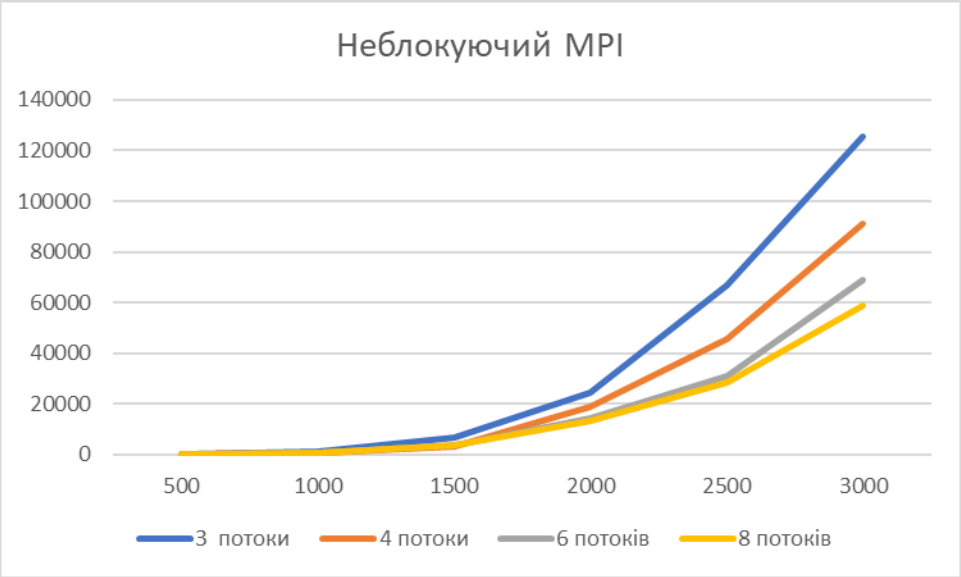
Для блокуючого MPI:

Розмірність матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	246	241	283	279
1000	978	690	626	646
1500	6941	2980	3683	3632
2000	24136	19524	13179	11593
2500	67245	45834	32705	29895
3000	116247	85168	70402	57420



Для неблокуючого MPI:

Розмірність матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	252	263	289	293
1000	882	620	693	616
1500	6756	3118	3527	3532
2000	24578	19041	14434	13014
2500	66683	45369	31083	28360
3000	125319	91133	69070	58871



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().equa
ls(matrix1.multiply(matrix2)) ?
//                    "Result is Correct" : "Result is
Incorrect");
            System.out.println("Total time: " +
nonBlockingMPIResult.getTotalTime());
```

```
            }


        }
}
```