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 № 7 з дисципліни
«Технології паралельних та розподілених обчислень»

«Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багатодо-багатьох») та дослідження його ефективності»

Виконав(ла) ІП-01 Галько М.В. _____
(шифр, прізвище, ім'я,

Перевірив(ла) Степенко І. В. _____
(прізвище, ім'я, по

Київ 2022

1. **Завдання:**

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. **40 балів.**
3. 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 методів обміну повідомленнями «один-доодного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох». **60 балів.**

2. **Хід роботи**

Для даної лабораторної роботи був створений клас CollectiveMPI, який використовує колективну комунікацію MPI для виконання операцій над матрицями у розподіленому середовищі.

У методі main виконується наступна послідовність дій:

1. Ініціалізація MPI за допомогою MPI.Init(args).
2. Отримання загальної кількості процесів та ідентифікатора поточного процесу за допомогою MPI.COMM_WORLD.Size() та MPI.COMM_WORLD.Rank() відповідно.
3. Обчислення кількості рядків для кожного процесу та кількості додаткових рядків для останнього процесу.
4. Створення масиву bytes, який містить розмір буфера для кожного процесу. За допомогою цього масиву визначається кількість даних, які будуть розсилатися кожному процесу.

5. Створення масиву offsets, який містить зміщення для кожного процесу. Це необхідно для коректного розподілу даних між процесами.

6. Конвертація матриці matrixA у байтовий масив та матриці matrixB у байтовий масив.

7. Визначення розміру даних для поточного процесу та створення буфера subMatrixBytes для отримання підматриці matrixA для поточного процесу.

8. Створення буфера для отримання результату множення матриць matrixA та matrixB.

9. Розсилка частини матриці matrixA (підматриці) з головного процесу до інших процесів за допомогою MPI.COMM_WORLD.Scatterv().

10. Розсилка матриці matrixB з головного процесу до всіх інших процесів за допомогою MPI.COMM_WORLD.Bcast().

11. Виконання множення підматриці на матрицю B , повернення результату у вигляді байтового масиву.

12. Збір результатів з усіх процесів на головному процесі за допомогою MPI.COMM_WORLD.Gatherv().

13. Якщо поточний процес є головним, то розбирається отриманий байтовий масив та створюється результуюча матриця з його даних.

14. Виведення інформації про виконання обчислень, такі як час виконання, розмір матриці та кількість процесів.

На завершення викликається MPI.Finalize(), що завершує роботу з MPI.

Приклад роботи програми для колективного MPI^

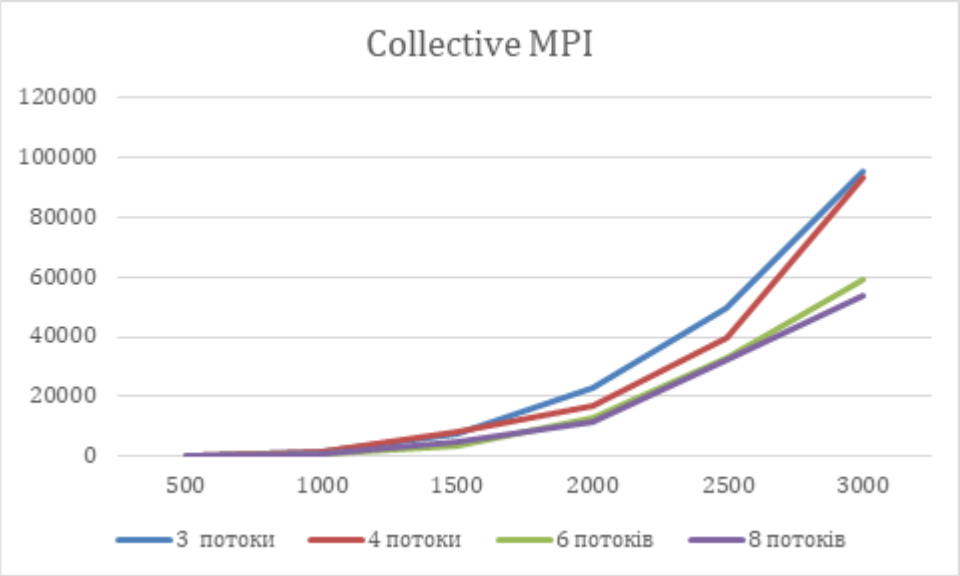
```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
MPJ Express (0.44) is started in the multicore configuration
CollectiveMPI
Total time: 683 ms
Matrix size: 1000x1000
Tasks count: 3

Process finished with exit code 0
```

В рамках лабораторної роботи також було проведене дослідження щодо виміру ефективності колективного MPI, порівняння його з методами обміну повідомленнями типу «один-до-одного» і тд.

Після проведення ряду тестів були отримані наступні результати:

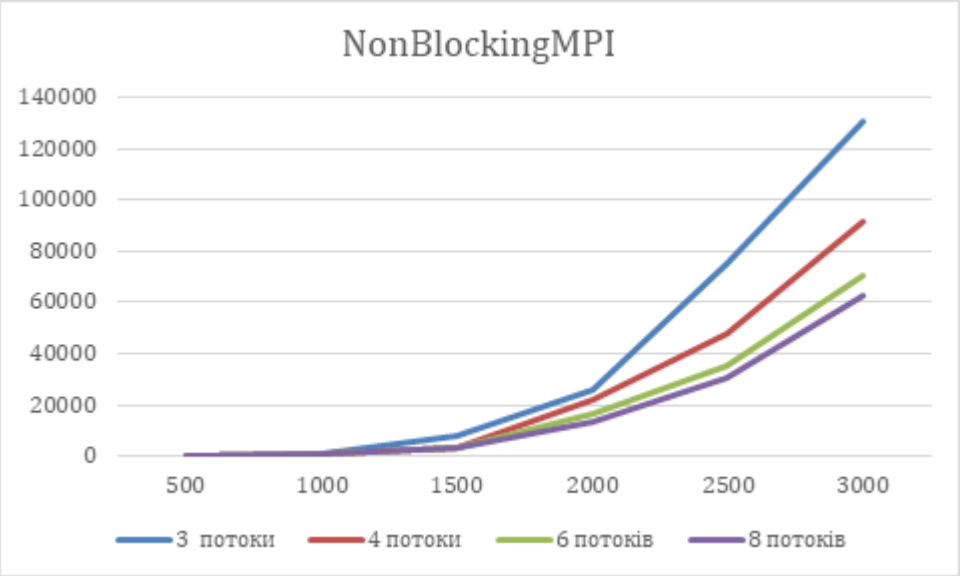
Розмір матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	363	410	327	282
1000	1360	1363	705	662
1500	7704	7659	4098	4229
2000	23923	17857	14321	11078
2500	53134	46945	30154	31017
3000	94800	92829	59971	63675

Графічне представлення:



При порівнянні результатів з неблокуючим обміном MPI, який був найшвидшим серед «один-до-одного» можемо побачити прискорення на рівні 1.1 – 1.3 рази

Нижче наведені результати тестування для неблокуючого MPI



3. Висновок

Порівнюючи роботу колективного MPI та MPI з один-до-одного обміном повідомленнями для множення матриць, можна зробити деякі спостереження.

Колективний MPI використовує оптимізовані колективні операції для обміну даними між вузлами. Це дозволяє зменшити кількість потрібних повідомлень та зменшити накладні витрати на комунікацію між вузлами. Крім того, колективні операції можуть бути ефективно оптимізовані на рівні бібліотеки MPI, що призводить до покращення продуктивності.

З іншого боку, MPI з один-до-одного обміном повідомленнями вимагає безпосереднього взаємодії між кожною парою вузлів. Це може призвести до збільшення кількості потрібних повідомлень та зростання накладних витрат на комунікацію між вузлами, особливо при збільшенні кількості вузлів або розміру матриць.

Отже, колективний MPI виявляється більш ефективним в порівнянні з MPI з один-до-одного обміном повідомленнями для множення матриць. Використання колективних операцій дозволяє знизити накладні витрати на комунікацію та досягти кращої продуктивності, зокрема при обробці великих матриць та на системах з багатьма вузлами.

4. Код

```
import mpi.MPI;

public class CollectiveMPI {
    private static final int MASTER_ID = 0;
    private static final int INT_32_BYTE_SIZE = 4;
    public static void main(String[] args) {
        int matrixSize = 1000;
        Matrix matrixA = new Matrix(matrixSize, matrixSize, true);
        Matrix matrixB = new Matrix(matrixSize, matrixSize, true);
        try{
            long startTime = System.currentTimeMillis();

            MPI.Init(args);

            int tasksCount = MPI.COMM_WORLD.Size();
            int taskID = MPI.COMM_WORLD.Rank();

            var rowsForOneWorker = matrixA.getRowsNumber() / tasksCount;
            var extraRows = matrixA.getRowsNumber() % tasksCount;

            int[] bytes = new int[tasksCount];
            for (var i = 0; i < tasksCount; i++) {
                if (i != tasksCount - 1) {
                    bytes[i] = rowsForOneWorker *
matrixB.getColumnsNumber() * INT_32_BYTE_SIZE;
                } else {
                    bytes[i] = (rowsForOneWorker + extraRows) *
matrixB.getColumnsNumber() * INT_32_BYTE_SIZE;
                }
            }

            int[] offsets = new int[tasksCount];
            for (var i = 0; i < offsets.length; i++) {
                if (i == 0) continue;

                offsets[i] = bytes[i - 1] + offsets[i - 1];
            }

            byte[] matrixAByteBuffer = matrixA.toByteBuffer();
            byte[] matrixBByteBuffer = matrixB.toByteBuffer();

            int taskBytes = bytes[taskID];
            byte[] subMatrixBytes = new byte[taskBytes];
            byte[] resBytes = new byte[matrixA.getRowsNumber() *
matrixB.getColumnsNumber() * INT_32_BYTE_SIZE];

            MPI.COMM_WORLD.Scatterv(matrixAByteBuffer, 0, bytes, offsets,
MPI.BYTE,
                    subMatrixBytes, 0, taskBytes, MPI.BYTE, 0);

            MPI.COMM_WORLD.Bcast(matrixBByteBuffer, 0,
matrixBByteBuffer.length, MPI.BYTE, 0);

            byte[] multiplicationResultBuffer =
performMatrixMultiplication(matrixA.getRowsNumber(),
                    matrixB.getColumnsNumber(), matrixBByteBuffer,
taskBytes, subMatrixBytes)
```

```
                    .toByteBuffer();

            MPI.COMM_WORLD.Gatherv(multiplicationResultBuffer, 0,
multiplicationResultBuffer.length,
                    MPI.BYTE, resBytes, 0, bytes, offsets, MPI.BYTE, 0);

            if (taskID == MASTER_ID) {
                Matrix resultMatrix =
Matrix.createMatrixFromBuffer(resBytes, matrixA.getRowsNumber(),
                        matrixB.getColumnsNumber());

                System.out.println("CollectiveMPI");
                System.out.println("Total time: " +
(System.currentTimeMillis() - startTime) + " ms");
                System.out.println("Matrix size: " +
resultMatrix.getRowsNumber() + "x" + resultMatrix.getColumnsNumber());
                System.out.println("Tasks count: " + tasksCount);
            }
        }finally {
            MPI.Finalize();
        }
    }

    private static Matrix performMatrixMultiplication(int matrix1RowsCount,
int matrix2ColumnsCount, byte[] secondMatrixBuffer, int taskBytes, byte[]
subMatrixBytes) {
        Matrix subMatrix = Matrix.createMatrixFromBuffer(subMatrixBytes,
taskBytes / (INT_32_BYTE_SIZE * matrix2ColumnsCount), matrix1RowsCount);
        Matrix secondMatrix =
Matrix.createMatrixFromBuffer(secondMatrixBuffer, matrix2ColumnsCount,
matrix1RowsCount);

        return subMatrix.multiply(secondMatrix);
    }
}
```