



Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики і програмної інженерії

Лабораторна робота №7
з дисципліни
Технології паралельних обчислень

Виконав:

студент групи ІІ-13:
Бабашев О. Д.

Перевірив:

ст.викл.
Дифучин А. Ю.

Київ 2024

Завдання до комп'ютерного практикуму 7

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

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох».
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями.
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-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох».

Хід роботи

Лістинг коду:

CollectiveMultiplier.java

```
package lab7;

//import java.util.Random;
import mpi.MPI;

public class CollectiveMultiplier {
    static final int SIZE = 1500;
    static final int MASTER = 0;
    public static void main(String[] args) {

        int[][] A = new int[SIZE][SIZE];
        int[][] B = new int[SIZE][SIZE];
        int[][] C = new int[SIZE][SIZE];

        MPI.Init(args);
```

```

int taskId = MPI.COMM_WORLD.Rank();
int tasksNumber = MPI.COMM_WORLD.Size();
int slice = SIZE / tasksNumber;

if (SIZE % tasksNumber != 0) {
    System.out.println("Should be no residue! SIZE % tasksNumber != 0");
    MPI.Finalize();
    System.exit(1);
}

if (tasksNumber < 2) {
    System.out.println("At least 2 tasks!");
    MPI.Finalize();
    System.exit(1);
}

long start = 0;

if (taskId == MASTER) {

    //Random random = new Random();
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            A[i][j] = 10;
            // A[i][j] = random.nextInt(11);
        }
    }

    for (int i = 0; i < SIZE; i++) {

```

```

        for (int j = 0; j < SIZE; j++) {
            B[i][j] = 10;
            // B[i][j] = random.nextInt(11);
        }
    }

    start = System.currentTimeMillis();

}

int[][] aBuffer = new int[slice][SIZE];

MPI.COMM_WORLD.Scatter(A, taskId * slice, slice, MPI.OBJECT,
aBuffer, 0, slice, MPI.OBJECT, MASTER);

MPI.COMM_WORLD.Bcast(B, 0, SIZE, MPI.OBJECT, MASTER);

int[][] cBuffer = new int[slice][SIZE];

for (int i = 0; i < slice; i++) {
    for (int j = 0; j < SIZE; j++) {
        for (int k = 0; k < SIZE; k++) {
            cBuffer[i][j] += aBuffer[i][k] * B[k][j];
        }
    }
}

MPI.COMM_WORLD.Gather(cBuffer, 0, slice, MPI.OBJECT, C, taskId *
slice, slice, MPI.OBJECT, MASTER);

if (taskId == MASTER) {
    long end = System.currentTimeMillis();

    System.out.println("\nCollective gather time: " + (end - start));
}

```

```

        int[][] SequentialMatrix = new int[SIZE][SIZE];
        SequentialMultiplier.multiply(A, B, SequentialMatrix);

        MatrixHelper.compareMatrices(C, SequentialMatrix);
    }

    MPI.Finalize();
}
}

```

CollectiveMultiplierAll.java

```

package lab7;

//import java.util.Random;
import mpi.MPI;

public class CollectiveMultiplierAll {
    static final int SIZE = 1500;
    static final int MASTER = 0;

    public static void main(String[] args) {

        int[][] A = new int[SIZE][SIZE];
        int[][] B = new int[SIZE][SIZE];
        int[][] C = new int[SIZE][SIZE];

        MPI.Init(args);

        int taskId = MPI.COMM_WORLD.Rank();
        int tasksNumber = MPI.COMM_WORLD.Size();
        int slice = SIZE / tasksNumber;
    }
}

```

```
if (SIZE % tasksNumber != 0) {  
    System.out.println("Should be no residue! SIZE % tasksNumber != 0");  
    MPI.Finalize();  
    System.exit(1);  
}
```

```
if (tasksNumber < 2) {  
    System.out.println("At least 2 tasks!");  
    MPI.Finalize();  
    System.exit(1);  
}
```

```
long start = 0;
```

```
if (taskId == MASTER) {  
  
    //Random random = new Random();  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            A[i][j] = 10;  
            // A[i][j] = random.nextInt(11);  
        }  
    }  
  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            B[i][j] = 10;  
            // B[i][j] = random.nextInt(11);  
        }  
    }  
}
```

```
start = System.currentTimeMillis();  
}
```

```
int[][] aBuffer = new int[slice][SIZE];  
MPI.COMM_WORLD.Scatter(A, taskId * slice, slice, MPI.OBJECT,  
aBuffer, 0, slice, MPI.OBJECT, MASTER);  
MPI.COMM_WORLD.Bcast(B, 0, SIZE, MPI.OBJECT, MASTER);
```

```
int[][] cBuffer = new int[slice][SIZE];
```

```
for (int i = 0; i < slice; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int k = 0; k < SIZE; k++) {  
            cBuffer[i][j] += aBuffer[i][k] * B[k][j];  
        }  
    }  
}
```

```
MPI.COMM_WORLD.Allgather(cBuffer, 0, slice, MPI.OBJECT, C, taskId *  
slice, slice, MPI.OBJECT);
```

```
if (taskId == MASTER) {  
    long end = System.currentTimeMillis();
```

```
    System.out.println("\nCollective all gather time: " + (end - start));
```

```
int[][] SequentialMatrix = new int[SIZE][SIZE];
```

```
SequentialMultiplier.multiply(A, B, SequentialMatrix);
```

```
        MatrixHelper.compareMatrices(C, SequentialMatrix);  
    }  
}
```

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

MatrixHelper.java

```
package lab7;
```

```
public class MatrixHelper {
```

```
    public static void printMatrix(int[][] matrix) {  
        for (int i = 0; i < matrix.length; i++) {  
            System.out.println();  
            for (int j = 0; j < matrix[0].length; j++) {  
                System.out.print(matrix[i][j] + " ");  
            }  
        }  
    }  
}
```

```
    public static void compareMatrices(int[][] a, int[][] b) {  
        for (int i = 0; i < a.length; i++) {  
            for (int j = 0; j < a[0].length; j++) {  
                if (a[i][j] != b[i][j]) {  
                    System.out.println("\nNOT Correct");  
                    return;  
                }  
            }  
        }  
    }  
}
```



```

        }
    }
    System.out.println("\nCorrect");
}

}

```

SequentialMultiplier.java

```

package lab7;

//import java.util.Random;

public class SequentialMultiplier {
    static final int A_ROWS = 1500;
    static final int A_COLS = 1500;
    static final int B_COLS = 1500;

    public static void multiply(int[][] a, int[][] b, int[][] c) {
        for (int i = 0; i < A_ROWS; i++) {
            for (int j = 0; j < B_COLS; j++) {
                for (int k = 0; k < A_COLS; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
    }

    public static void main(String[] args) {
        int[][] A = new int[A_ROWS][A_COLS];
        int[][] B = new int[A_COLS][B_COLS];
    }
}

```

```

int[][] C = new int[A_ROWS][B_COLS];

//Random random = new Random();
for (int i = 0; i < A_ROWS; i++) {
    for (int j = 0; j < A_COLS; j++) {
        A[i][j] = 10;
        //A[i][j] = random.nextInt(11);
    }
}

for (int i = 0; i < A_COLS; i++) {
    for (int j = 0; j < B_COLS; j++) {
        B[i][j] = 10;
        //B[i][j] = random.nextInt(11);
    }
}

long start = System.currentTimeMillis();
multiply(A, B, C);
long end = System.currentTimeMillis();
System.out.println("\nMatrix A:\n");
MatrixHelper.printMatrix(A);
System.out.println("\nMatrix B:\n");
MatrixHelper.printMatrix(B);
System.out.println("\nMatrix C:\n");
MatrixHelper.printMatrix(C);

System.out.println("\nSequential time: " + (end - start));
}
}

```

Результати виконання коду:

Колективні операції забезпечують обмін повідомленнями між усіма процесами. Вони дозволяють ефективно розподіляти дані і збирати результати. Основними типами колективних операцій є `scatter`, `bcast`, `gather`, і `gather all`.

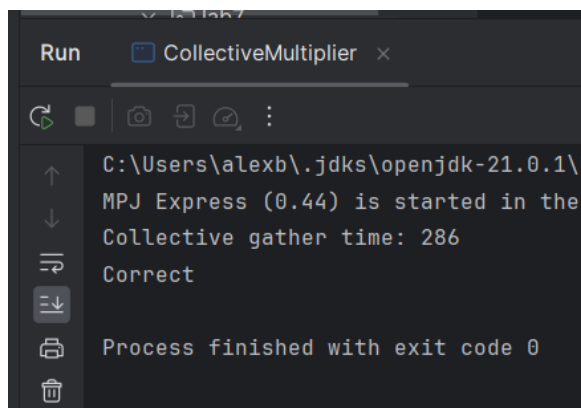
Операція `scatter` розподіляє різні частини одного великого масиву даних від одного процесу (кореневого) до всіх інших процесів у комунікаторі. Кожен процес отримує унікальний підмасив даних.

Операція `bcast` (`broadcast`) передає дані від одного процесу (кореневого) до всіх інших процесів у комунікаторі. Це ефективний спосіб розповсюдження одних і тих же даних до всіх процесів.

Операція `gather` збирає дані від усіх процесів у комунікаторі і об'єднує їх у одному кореновому процесі. Кожен процес відправляє свої дані кореновому процесу, який зберігає їх у масиві.

Операція `allgather` є розширенням `gather`. Вона збирає дані від усіх процесів і розподіляє їх всім процесам у комунікаторі. Таким чином, кожен процес отримує повний набір зібраних даних.

Реалізований алгоритм паралельного множення матриць з використанням методів колективного обміну повідомленнями, протестовано на 5 процесах з розмірністю матриці 500x500 з використанням операції `gather`.



```
C:\Users\alexh\.jdk\openjdk-21.0.1\bin
MPJ Express (0.44) is started in the
Collective gather time: 286
Correct
Process finished with exit code 0
```

Рисунок 2.1 – Тестування алгоритму паралельного множення матриць з використанням методів колективного обміну повідомленнями з використанням операції `gather`.

Маємо коректний результат.

Аналогічно протестуємо правильність роботи алгоритму з використанням операції `allgather`.

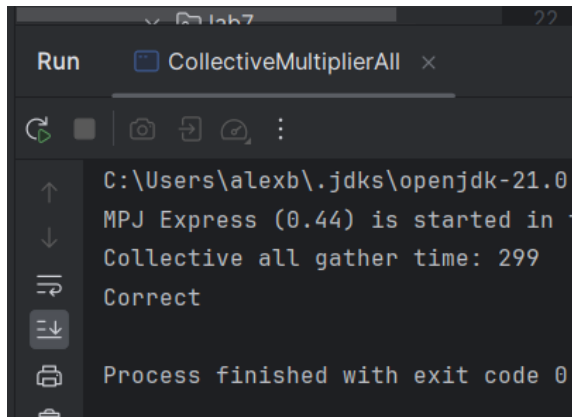


Рисунок 2.2 – Тестування алгоритму паралельного множення матриць з використанням методів колективного обміну повідомленнями з використанням операції allgather.

Маємо коректний результат.

Аналіз

Тестування з використанням операції gather.

size np	2	5	10	20
500	0.581	0.569	0.543	0.454
1000	3.987	3.360	2.966	3.285
1500	24.709	13.358	10.276	12.404

Тестування з використанням операції allgather.

size np	2	5	10	20
500	0.631	0.686	0.610	0.759
1000	3.985	4.099	3.758	4.707
1500	25.659	14.129	12.567	13.671

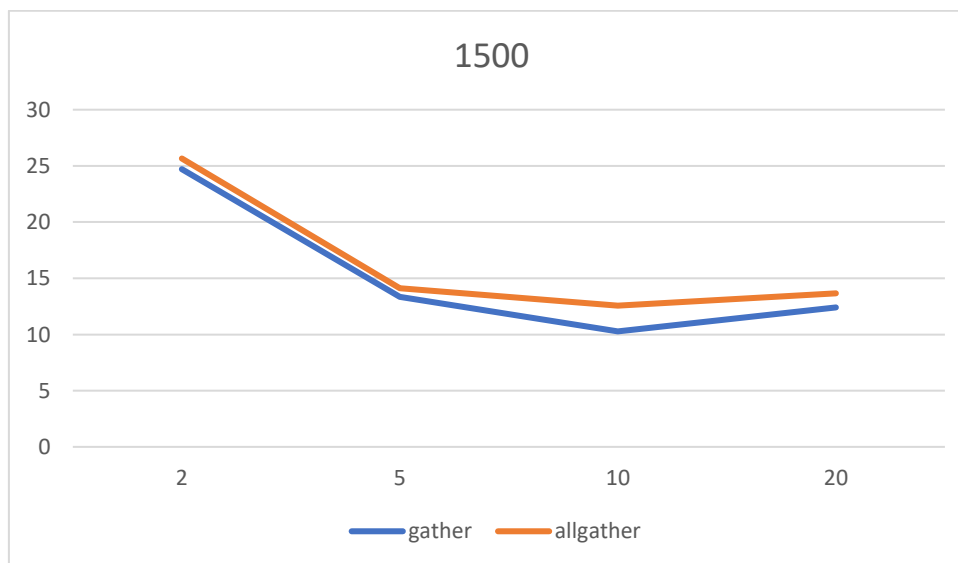
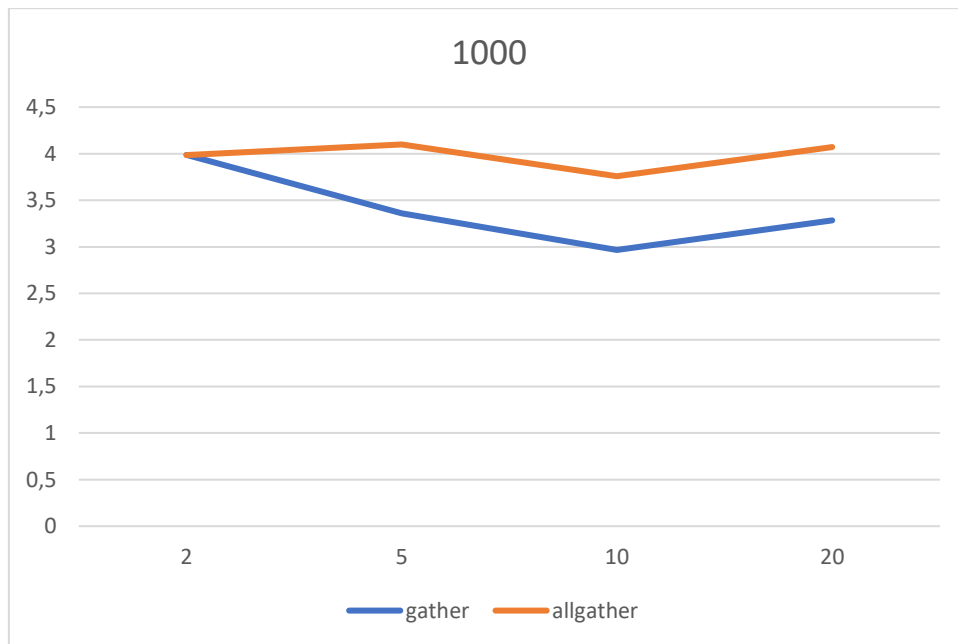
Тестування з блокуючими методами.

size np	2	6	10	16
500	1.015	0.463	0.439	0.671
1000	4.629	2.043	3.771	4.571
1500	34.679	21.233	16.084	20.224
2000	84.908	48.037	62.464	52.680

Тестування з неблокуючими методами.

	2	6	10	16
500	0.688	0.313	0.417	0.541
1000	3.123	1.255	3.399	3.670
1500	32.743	15.868	13.558	19.146
2000	78.507	37.957	49.718	42.469

Графіки залежності швидкості виконання від кількості процесів на різних розмірах матриць.



Узагальнюючи результати нашого дослідження, можемо зазначити, що колективний обмін повідомленнями працює зазвичай швидше, ніж

індивідуальні блокуючі та неблокуючі методи, завдяки використанню одразу всіх процесів, оптимізованим алгоритмам, зменшенню накладних витрат та зниженню кількості комунікаційних кроків. Використання колективних операцій дозволяє підвищити ефективність паралельних програм і краще використовувати ресурси обчислювальної системи.

Також з наведених вище даних робимо висновок, що операція `gather` швидше ніж `allgather`. Це можна пояснити тим, що `gather` потребує менше комунікаційних кроків і менший обсяг переданих даних. В `gather` лише кореневий процес отримує всі дані, тоді як в `allgather` кожен процес отримує дані від усіх інших процесів, що значно збільшує накладні витрати на комунікацію і загальний час виконання операції.

Висновок

В ході лабораторної роботи ми розробили алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. Також ми дослідили колективні методи та їх переваги й недоліки у порівнянні з методами один-до-одного, до того ж дослідили різні операції для обміну інформацією між процесорами та провели детальний аналіз їх ефективності.