МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

Звіт

З лабораторної роботи № 2 з дисципліни
«Технології паралельних обчислень»

Тема: «Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності»

Виконав(ла) *ІП-14 Бабіч Денис* _____

(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучина О. Ю.* _____

(шифр, прізвище, ім'я, по батькові)

Київ 2024

ОСНОВНА ЧАСТИНА

Мета роботи: Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності.

Для порівняння результатів роботи паралельних алгоритмів множення матриці також був реалізований стандартний послідовний алгоритм множення матриць.

Лістинг методу multiplySequential

```
public static Result multiplySequential(MatrixInt matrix1, MatrixInt matrix2)
{
    if (!MatrixInt.areMultipliable(matrix1, matrix2))
    {
        throw new IllegalArgumentException("Matrices are not multipliable.");
    }

    MatrixInt resultingMatrix = new MatrixInt(matrix1.rows, matrix2.columns);

    long timestepStart = System.currentTimeMillis();

    for (int rowIndex = 0; rowIndex < matrix1.rows; ++rowIndex)
    {
        for (int columnIndex = 0; columnIndex < matrix2.columns; ++columnIndex)
        {
                for (int dimensionIndex = 0; dimensionIndex < matrix1.columns;
++dimensionIndex)
            {
                                resultingMatrix.matrix[rowIndex][columnIndex] +=
matrix1.matrix[rowIndex][dimensionIndex] *
matrix2.matrix[dimensionIndex][columnIndex];
```

```
            }
        }
    }

    long timestepEnd = System.currentTimeMillis();

    long executionTime = timestepEnd - timestepStart;

    Result result = new Result(resultingMatrix, executionTime);

    return result;
}
```

```
Dimensions: 500 | Execution time: 0.297 seconds, (297 milliseconds).
Dimensions: 1000 | Execution time: 2.777 seconds, (2777 milliseconds).
Dimensions: 1500 | Execution time: 22.334 seconds, (22334 milliseconds).
Dimensions: 2000 | Execution time: 63.076 seconds, (63076 milliseconds).
Dimensions: 2500 | Execution time: 131.841 seconds, (131841 milliseconds).
Dimensions: 3000 | Execution time: 240.793 seconds, (240793 milliseconds).
```

Рисунок 1.1 – Результати виконання послідовного алгоритму

1.1. Реалізуйте стрічковий алгоритм множення матриць. Результат множення записуйте в об’єкт класу Result.

Лістинг методу multiplyStriped

```
public static Result multiplyStriped(MatrixInt matrix1, MatrixInt matrix2, int threadsCount)
  {
    if (!MatrixInt.areMultipliable(matrix1, matrix2))
    {
      throw new IllegalArgumentException("Matrices are not multipliable.");
    }

    MatrixInt resultingMatrix = new MatrixInt(matrix1.rows, matrix2.columns);

    long timestepStart = System.currentTimeMillis();

    int totalTasks = matrix1.rows * matrix2.columns;
    int tasksPerThread = totalTasks / threadsCount;

    Thread[] threads = new Thread[threadsCount];

    for (int i = 0; i < threadsCount; ++i)
    {
      final int THREAD_ID = i;

      threads[i] = new Thread(() ->
      {
        int startTaskIndex = THREAD_ID * tasksPerThread;
        int endTaskIndex = (THREAD_ID == threadsCount - 1) ? totalTasks : startTaskIndex + tasksPerThread;
```

```
        for (int j = startTaskIndex; j < endTaskIndex; ++j)
        {
            int rowIndex = j / matrix1.columns;
            int columnIndex = j % matrix2.columns;

            int[] row = matrix1.getRow(rowIndex);
            int[] column = matrix2.getColumn(columnIndex);

            int result = 0;

            for (int k = 0; k < row.length; ++k)
            {
                result += row[k] * column[k];
            }

            resultingMatrix.set(rowIndex, columnIndex, result);
        }
    });
}

for (Thread thread : threads)
{
    thread.start();
}

for (Thread thread : threads)
{
    try
```

```
            {
                thread.join();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        long timestepEnd = System.currentTimeMillis();

        long executionTime = timestepEnd - timestepStart;

        Result result = new Result(resultingMatrix, executionTime);

        return result;
    }
```

```
Threads: 4; Dimensions: 500 | Execution time: 0.157 seconds, (157 milliseconds).
Threads: 4; Dimensions: 1000 | Execution time: 1.092 seconds, (1092 milliseconds).
Threads: 4; Dimensions: 1500 | Execution time: 8.032 seconds, (8032 milliseconds).
Threads: 4; Dimensions: 2000 | Execution time: 22.189 seconds, (22189 milliseconds).
Threads: 4; Dimensions: 2500 | Execution time: 45.026 seconds, (45026 milliseconds).
Threads: 4; Dimensions: 3000 | Execution time: 93.289 seconds, (93289 milliseconds).
Threads: 9; Dimensions: 500 | Execution time: 0.078 seconds, (78 milliseconds).
Threads: 9; Dimensions: 1000 | Execution time: 1.122 seconds, (1122 milliseconds).
Threads: 9; Dimensions: 1500 | Execution time: 5.846 seconds, (5846 milliseconds).
Threads: 9; Dimensions: 2000 | Execution time: 15.135 seconds, (15135 milliseconds).
Threads: 9; Dimensions: 2500 | Execution time: 33.438 seconds, (33438 milliseconds).
Threads: 9; Dimensions: 3000 | Execution time: 58.878 seconds, (58878 milliseconds).
Threads: 16; Dimensions: 500 | Execution time: 0.092 seconds, (92 milliseconds).
Threads: 16; Dimensions: 1000 | Execution time: 1.210 seconds, (1210 milliseconds).
Threads: 16; Dimensions: 1500 | Execution time: 5.298 seconds, (5298 milliseconds).
Threads: 16; Dimensions: 2000 | Execution time: 13.941 seconds, (13941 milliseconds).
Threads: 16; Dimensions: 2500 | Execution time: 28.500 seconds, (28500 milliseconds).
Threads: 16; Dimensions: 3000 | Execution time: 54.360 seconds, (54360 milliseconds).
```

Рисунок 1.2 – Результати виконання стрічкового алгоритму

Таблиця 1.1 – Результати роботи стрічкового алгоритму

Matrix Size	Serial algorithm	Stripe algorithm					
		4 processors		9 processors		16 processors	
		Time	Speed up	Time	Speed up	Time	Speed up
500	0.297	0.157	1.89172	0.078	3.80769	0.092	3.22826
1000	2.777	1.092	2.54304	1.122	2.47504	1.21	2.29504
1500	22.334	8.032	2.78063	5.846	3.82039	5.298	4.21555
2000	63.076	22.19	2.84254	15.135	4.16756	13.941	4.5245
2500	131.841	45.03	2.92785	33.438	3.94285	28.5	4.626
3000	240.793	93.29	2.58112	58.878	4.08969	54.36	4.4296

1.2. Реалізуйте алгоритм Фокса множення матриць.

Лістинг методу multiplyFox

```
public static Result multiplyFox(MatrixInt matrix1, MatrixInt matrix2, int threadsCount)
    {
        if (!MatrixInt.areMultipliable(matrix1, matrix2))
        {
            throw new IllegalArgumentException("Matrices are not multipliable.");
        }

        int threadsPerBlock = (int) Math.sqrt(threadsCount);

        if (threadsPerBlock * threadsPerBlock != threadsCount)
        {
            throw new IllegalArgumentException("threadsCount is not power of 2.");
        }
```

```
MatrixInt resultingMatrix = new MatrixInt(matrix1.rows, matrix2.columns);

long timestepStart = System.currentTimeMillis();

int threadPayload = (int) Math.ceil((double) matrix1.rows / threadsPerBlock);

int threadIndex = 0;
Thread[] threads = new Thread[threadsCount];

for (int rowIndex = 0; rowIndex < matrix1.rows; rowIndex += threadPayload)
{
    for (int columnIndex = 0; columnIndex < matrix2.columns; columnIndex += threadPayload)
    {
        final int ROW_INDEX = rowIndex;
        final int COLUMN_INDEX = columnIndex;

        threads[threadIndex++] = new Thread(() ->
        {
            int matrix1RowSize =  (ROW_INDEX + threadPayload) > matrix1.rows ? (matrix1.rows - ROW_INDEX) : threadPayload;
                int matrix2ColumnSize = (COLUMN_INDEX + threadPayload) > matrix2.columns ? (matrix2.columns - COLUMN_INDEX) : threadPayload;

            for (int i = 0; i < matrix1.rows; i += threadPayload)
            {
                    int matrix2RowSize = (i + threadPayload) > matrix2.rows ? (matrix2.rows - i) : threadPayload;
```

```
                int matrix1ColumnSize = (i + threadPayload) > matrix1.columns ? (matrix1.columns - i) : threadPayload;

                MatrixInt block1 = copyMatrixIntBlock(matrix1, ROW_INDEX, ROW_INDEX + matrix1RowSize, i, i + matrix1ColumnSize);
                MatrixInt block2 = copyMatrixIntBlock(matrix2, i, i + matrix2RowSize, COLUMN_INDEX, COLUMN_INDEX + matrix2ColumnSize);

                MatrixInt resultingBlock = MatrixInt.multiplySequential(block1, block2).getMatrixInt();

                for (int j = 0; j < resultingBlock.rows; ++j)
                {
                   for (int k = 0; k < resultingBlock.columns; ++k)
                   {
                      resultingMatrix.set(j + ROW_INDEX, k + COLUMN_INDEX, resultingBlock.get(j, k) + resultingMatrix.get(j + ROW_INDEX, k + COLUMN_INDEX));
                   }
                }
             }
          });
       }
    }

    for (Thread thread : threads)
    {
       thread.start();
    }
```

```
        for (Thread thread : threads)
        {
            try
            {
                thread.join();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        long timestepEnd = System.currentTimeMillis();

        long executionTime = timestepEnd - timestepStart;

        Result result = new Result(resultingMatrix, executionTime);

        return result;
    }
```

Лістинг методу copyMatrixIntBlock

```
private static MatrixInt copyMatrixIntBlock(MatrixInt source, int rowStart, int rowFinish, int columnStart, int columnFinish)
    {
        final int OFFSET_ROW = rowFinish - rowStart;
        final int OFFSET_COLUMN = columnFinish - columnStart;
```

```
        MatrixInt matrix = new MatrixInt(rowFinish - rowStart, columnFinish - columnStart);

        for (int i = 0; i < OFFSET_ROW; ++i)
        {
            for (int j = 0; j < OFFSET_COLUMN; ++j)
            {
                matrix.set(i, j, source.get(i + rowStart, j + columnStart));
            }
        }

        return matrix;
    }
```

```
Threads: 4; Dimensions: 500 | Execution time: 0.135 seconds, (135 milliseconds).
Threads: 4; Dimensions: 1000 | Execution time: 0.535 seconds, (535 milliseconds).
Threads: 4; Dimensions: 1500 | Execution time: 2.161 seconds, (2161 milliseconds).
Threads: 4; Dimensions: 2000 | Execution time: 10.344 seconds, (10344 milliseconds).
Threads: 4; Dimensions: 2500 | Execution time: 31.289 seconds, (31289 milliseconds).
Threads: 4; Dimensions: 3000 | Execution time: 58.449 seconds, (58449 milliseconds).
Threads: 9; Dimensions: 500 | Execution time: 0.039 seconds, (39 milliseconds).
Threads: 9; Dimensions: 1000 | Execution time: 0.355 seconds, (355 milliseconds).
Threads: 9; Dimensions: 1500 | Execution time: 1.389 seconds, (1389 milliseconds).
Threads: 9; Dimensions: 2000 | Execution time: 4.535 seconds, (4535 milliseconds).
Threads: 9; Dimensions: 2500 | Execution time: 10.647 seconds, (10647 milliseconds).
Threads: 9; Dimensions: 3000 | Execution time: 21.996 seconds, (21996 milliseconds).
Threads: 16; Dimensions: 500 | Execution time: 0.035 seconds, (35 milliseconds).
Threads: 16; Dimensions: 1000 | Execution time: 0.354 seconds, (354 milliseconds).
Threads: 16; Dimensions: 1500 | Execution time: 1.220 seconds, (1220 milliseconds).
Threads: 16; Dimensions: 2000 | Execution time: 4.389 seconds, (4389 milliseconds).
Threads: 16; Dimensions: 2500 | Execution time: 11.506 seconds, (11506 milliseconds).
Threads: 16; Dimensions: 3000 | Execution time: 18.297 seconds, (18297 milliseconds).
```

Рисунок 1.3 – Результати роботи алгоритма Фокса

Таблиця 1.2 – Результати роботи алгоритма Фокса

Matrix Size	Serial algorithm	Fox algorithm					
		4 processors		9 processors		16 processors	
		Time	Speed up	Time	Speed up	Time	Speed up
500	0.297	0.135	2.2	0.039	7.61538	0.035	8.48571
1000	2.777	0.535	5.19065	0.355	7.82254	0.354	7.84463
1500	22.334	2.161	10.33503	1.389	16.07919	1.22	18.30656
2000	63.076	10.344	6.09783	4.535	13.90871	4.389	14.37138
2500	131.841	31.289	4.21365	10.647	12.38292	11.506	11.45846
3000	240.793	58.449	4.11971	21.996	10.94713	18.297	13.16024

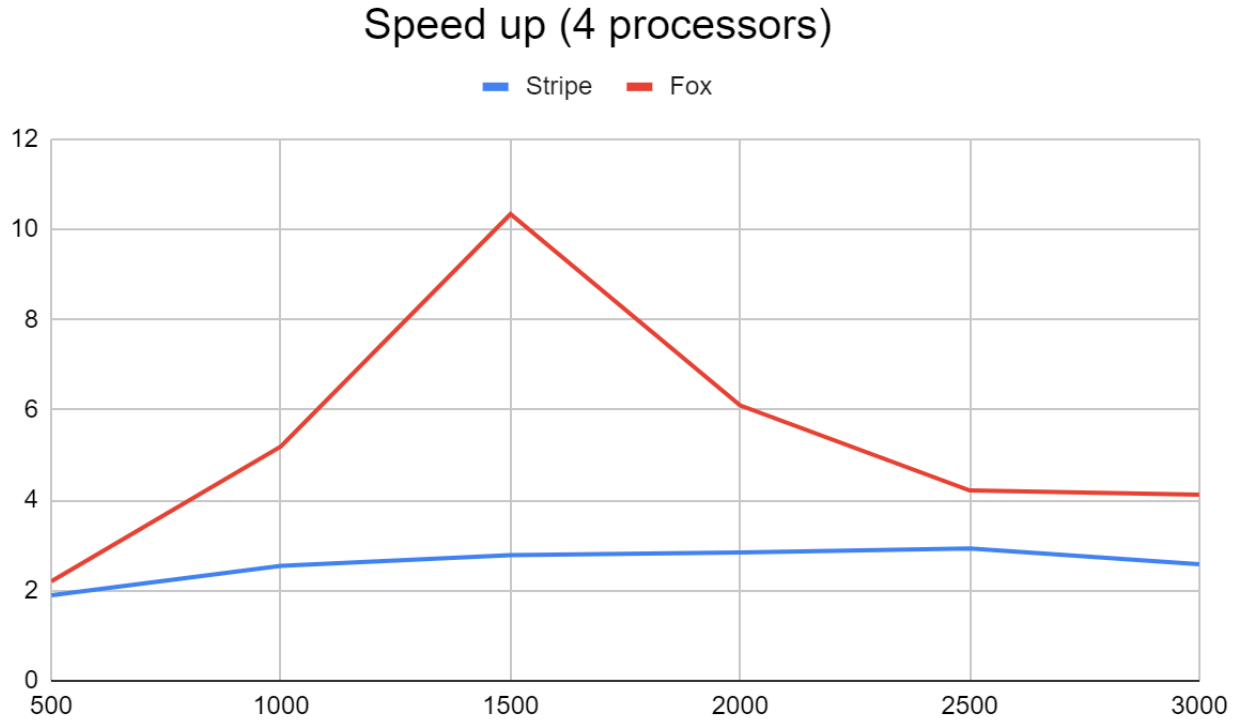

Рисунок 1.4 – Порівняння отриманого прискорення на 4 процесорах

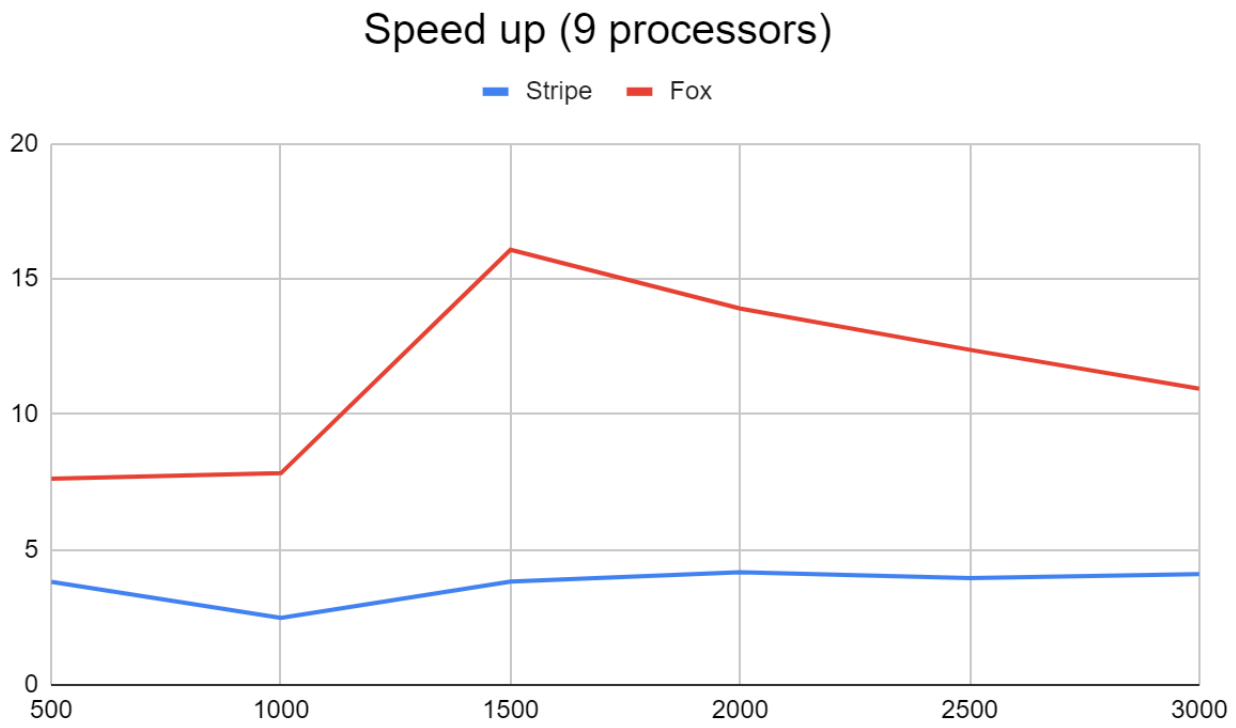

Рисунок 1.5 – Порівняння отриманого прискорення на 9 процесорах

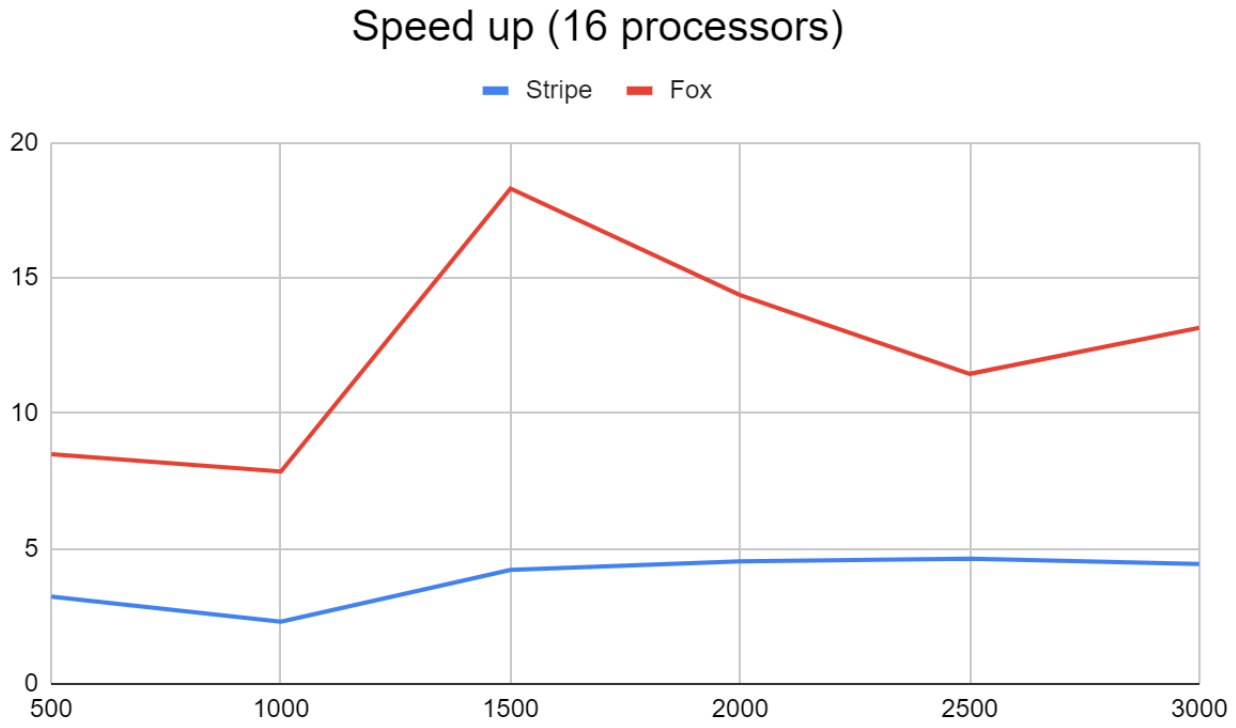

Рисунок 1.6 – Порівняння отриманого прискорення на 16 процесорах

ВИСНОВКИ

Під час виконання даної лабораторної роботи було реалізовано два алгоритми множення матриць: стрічковий алгоритм та алгоритм Фокса. Обидва алгоритми були реалізовані з використанням паралельних потоків для оптимізації процесу множення. Під час проведення експериментів, варіюючи розмірність матриць та кількість потоків, був зареєстрували час виконання обох алгоритмів. Результати дослідження показали, що алгоритм Фокса виявився ефективнішим в порівнянні зі стрічковим алгоритмом. Це може бути пояснено тим, що алгоритм Фокса ефективніше розподіляє обчислення між потоками, завдяки чому вдається досягти більшої продуктивності при оптимальній кількості потоків та розмірностей матриць. Завдяки чому можна зробити висновок про важливість вибору відповідного алгоритму для конкретної задачі та ресурсів, доступних для виконання обчислень.