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;

}

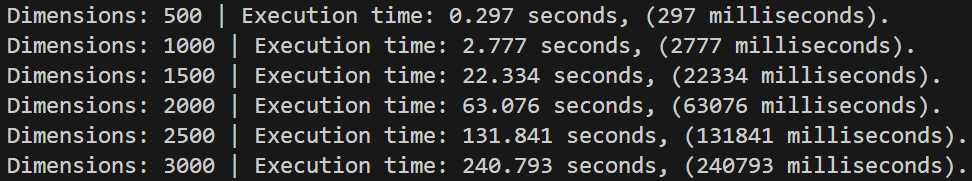


Рисунок 1.1 – Результати виконання послідовного алгоритму

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;

}

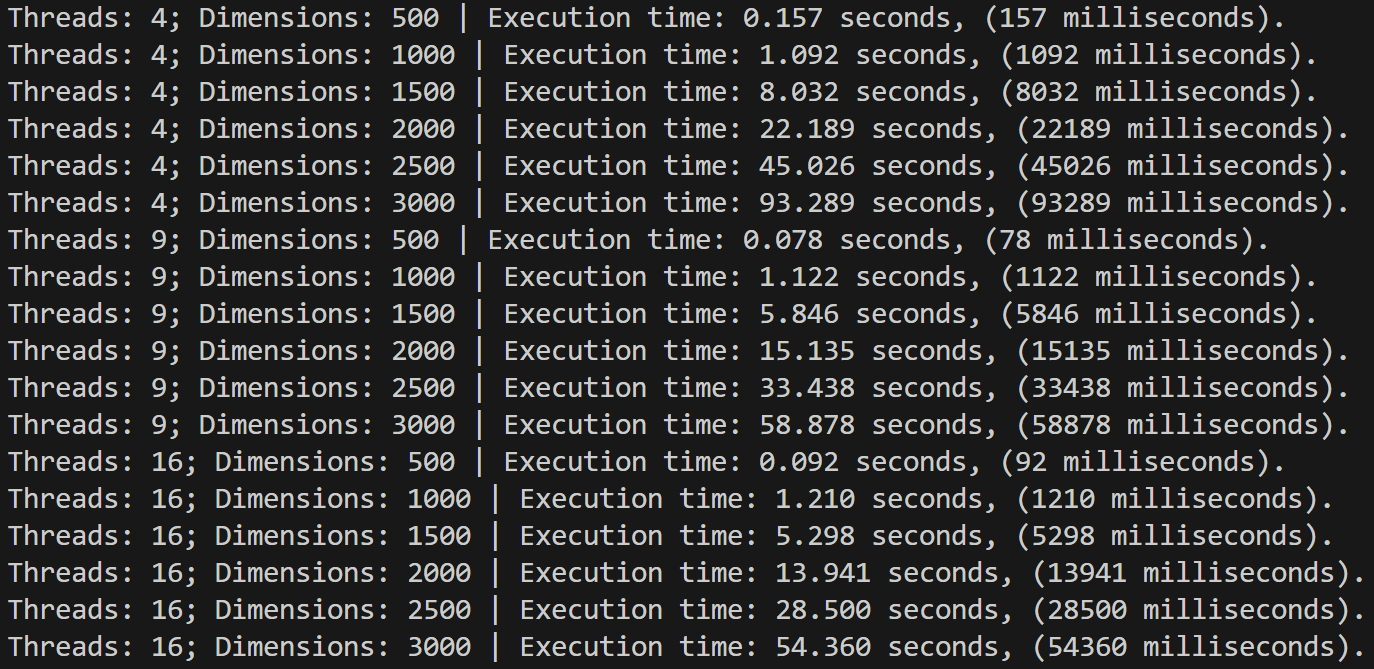


Рисунок 1.2 – Результати виконання стрічкового алгоритму

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. 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;

}

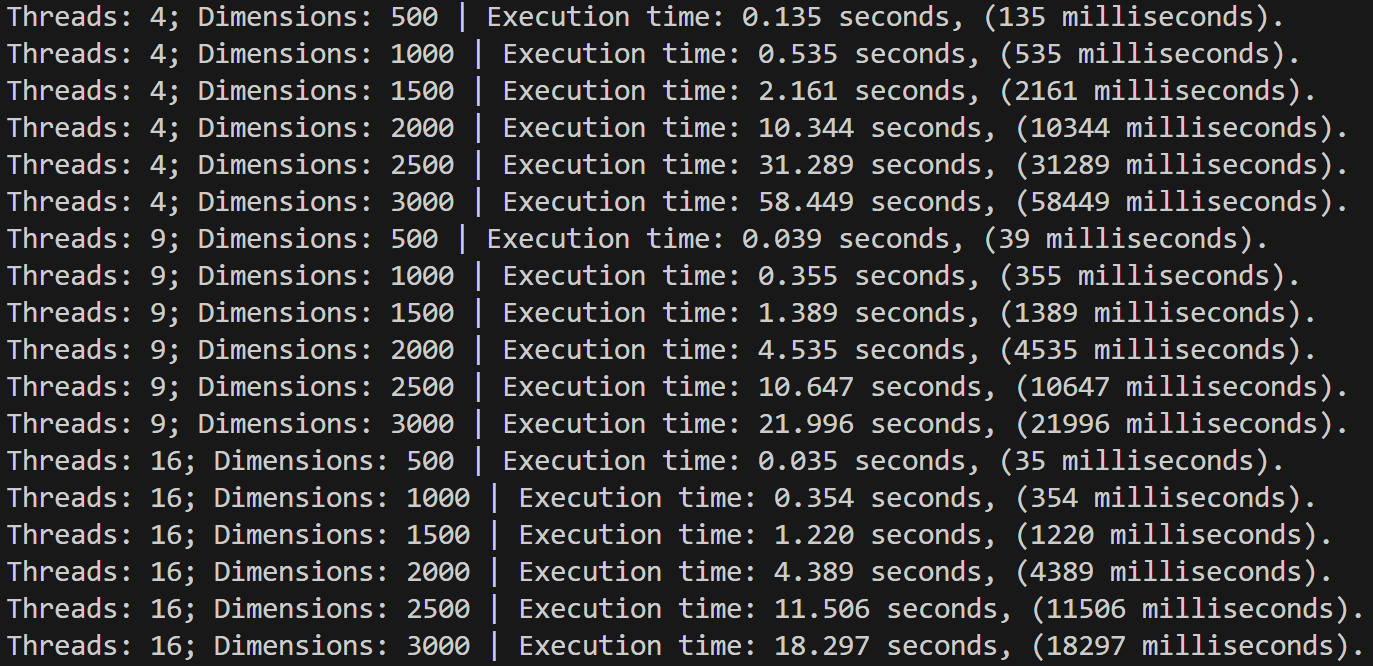


Рисунок 1.3 – Результати роботи алгоритма Фокса

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 |

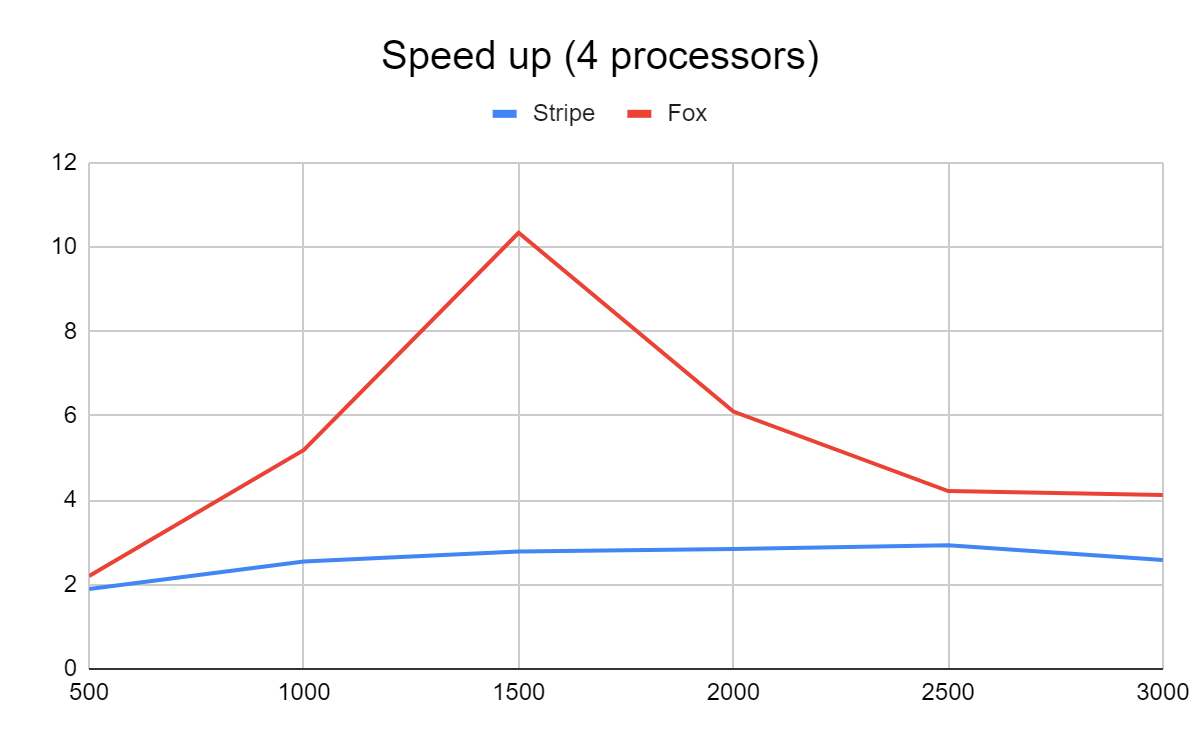


Рисунок 1.4 – Порівняння отриманого прискорення на 4 процесорах

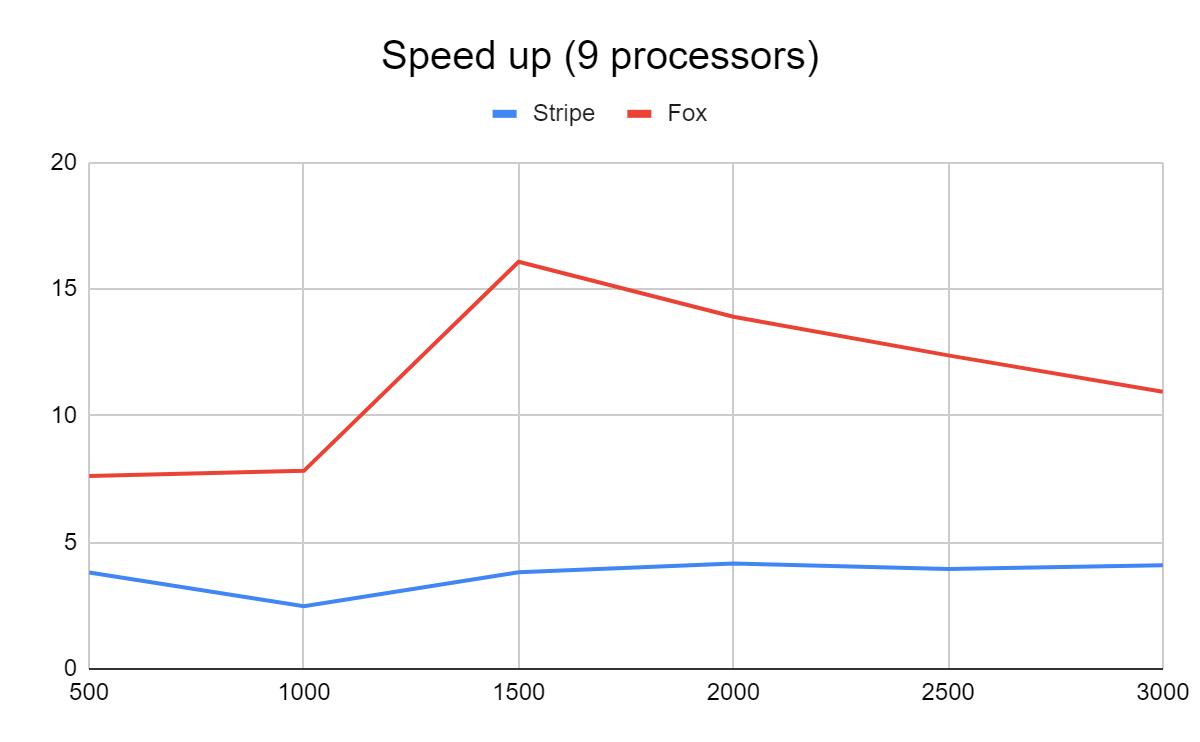


Рисунок 1.5 – Порівняння отриманого прискорення на 9 процесорах

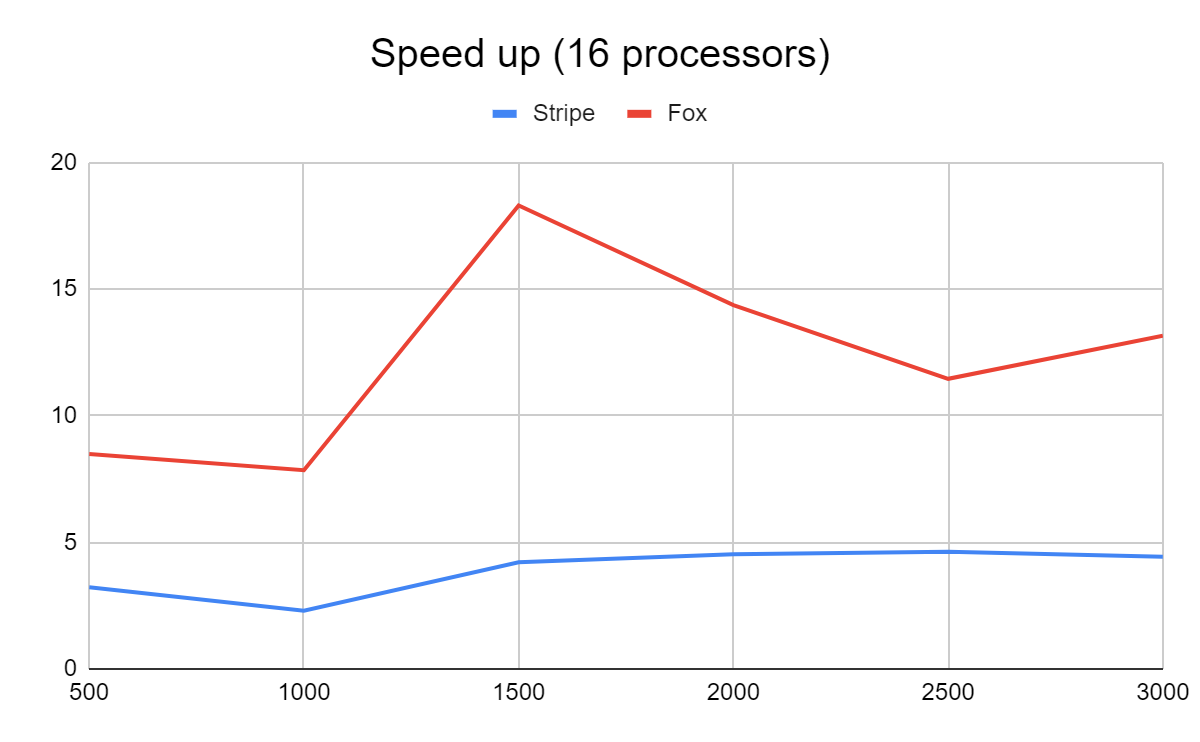


Рисунок 1.6 – Порівняння отриманого прискорення на 16 процесорах

# 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.