Java Matrix Multiplication Benchmark: Threads and Streams

Alejandro Jesús González Santana

December 2023

Contents

Github Repository: https://github.com/Alejeglez/FourthBenchmarkingTaskMatrixMultiplication

---

Abstract

One of the most challenging aspects to consider when initiating a software project is resource management. While our primary focus is often on achieving desired outcomes, this time we will try to obtain the most from the processing available resources.

Our prior work involved comparing matrix multiplication across different programming languages and experimenting with diverse formats. However, the current focus shifts towards optimizing operations, leveraging available resources judiciously to minimize execution time. Until now, wehave exclusively executed matrix multiplication sequentially. This paper delves deeper into the realms of threads and streams.

The findings of this experiment underscore the significant reduction in execution time achievable by using previously untapped resources(cores). Furthermore, we explore the development of specialized threads and streams for parallel operations.

1 Introduction

Software engineering aims to simplify tasks and maximize productivity by searching efficient solutions. To assess the performance of a program resources usage, benchmarking serves as a crucial tool for gauging resource utilization.

Matrix multiplication proves to be a practical standard for evaluating how well a programming language manages resources during execution. Yet, from our prior experience, we know that these operations can become computationally intensive, resulting in a steep rise in complexity. This increased complexity often translates into longer processing times that we could face using different matrix representations.

However, our previous matrix multiplications were executed sequentially, utilizing a single core to handle each individual multiplication operation. This approach involved the core processing each number-by-number multiplication independently. The remedy to this lies in embracing parallelism.

Parallelism is a method of executing operations by distributing tasks across multiple processing units. In a single computer, these processing units are the CPU cores (threads). In a more intricate distributed programming architecture, such as a cluster, these units can be individual computers. The extensive documentation on this topic boasts approximately 169,296 papers, according to the scientific paper search engine Core[1].

[1]Core website: `https://core.ac.uk/search?q=parallelism`

Various methods can be employed to implement parallelism, with threads being one of the most common. Threads serve as a logical abstraction of CPU cores. The Streams API, on the other hand, is designed for processing collections of objects. It offers methods that can be chained together to perform operations on data. When creating a Stream, it is possible to specify parallel operation. In Java, threads can also be instantiated as objects.

The primary objective of this research is to validate the outcomes of these operations and compare the time differences between the two approaches against the classical sequential method. Our aim is to demonstrate that equivalent results can be achieved with a significant reduction in execution time.

2 Problem Statement

Matrix multiplication has become a focal point of study due to its scalability from a complexity perspective. However, another crucial aspect of this problem is its divisibility. As observed in previous works, the execution times of matrix multiplication significantly increase with the growth in the number of rows and columns. Performing a matrix multiplication of size 2x2 is computationally lighter and more efficient for a computer than one of size 128x128.

To address this challenge and capitalize on divisibility, we propose to divide the matrices involved into submatrices. This introduces new submatrix operations for matrix multiplication, such as rotation by row or column and matrix composition. We will delve deeper into these methods in the following sections.

The focus on submatrices becomes essential when performing matrix multiplication with threads, as each thread must specify the area of the matrix it operates on. In contrast, the Stream API implementation does not necessitate submatrices and can operate on entire matrices as arguments.

The parallel implementation aims to maximize core efficiency by distributing tasks among them. This capitalizes on divisibility, making operations more resource-efficient. Additionally, task overlapping enables simultaneous execution in different processing units.

Our approach involves designing tests using JUnit[2] to validate the performance of matrix multiplications across parallelism methods. We have also employed JMH (Java Microbenchmark Harness) to measure the average time (ms/-operation) for each execution method across multiple iterations and forks, along with throughput (operations/ms). The objective is to demonstrate that these parallel methods can outperform conventional ones by managing lighter matrices and leveraging available resources.

[2]JUnit reference https://junit.org/junit5/

3 Methodology

Machine specs:

One of the key aspects in bechmarking is the hardware used and the operating system. In this field, the specification of the machine which ran the multiplications is:

- Model: Lenovo Ideapad 330.
- CPU : Intel Core i7-8750H (6 cores, 2,2 GHz - 4,1 GHz, 9 MB).
- GPU : NVIDIA GeForce GTX 1050 4 GB.
- RAM : 16 GB 2400 MHz DDR4.
- HDD: 1 TB.
- SSD: 256 GB.
- OS: Microsoft Windows 10 Pro 64 bits.
- Logical processors: 12.

The programs were executed on an HDD disk, so the results may differ compared to running them on an SSD disk.

3.1 Project Structure

For this project there are only a few classes so we have suprimed the necessity of more packages:

```
Project
└─ java
    ├─ DenseMatrix (class which represents the object where a matrix
    │  will be stored in Dense format, is an implementation of the
    │  interface Matrix)
    ├─ DenseMatrixBuilder (class whose main goal is to create Matrix
    │  objects, is an implementation of the interface Matrix)
    ├─ MatrixOperations (class with the matrices multiplication functions,
    │  and functions to divide matrices in submatrices, as well as,
    │  to sum matrices)
    ├─ SubMatricesOperations (class with to make sums and multiplications
    │  oriented to threads, as well as, rotating matrices and join
    │  submatrices in a matrix)
    ├─ Streams (class to make matrix multiplications oriented to Java
    │  Streams Api)
    ├─ ThreadSum (class that extends the class Thread to operate a
    │  matrix sum)
    └─ ThreadMultiplication (class that extends the class Thread to
       operate a matrix multiplication)
```

  - Matrix (Interface that establish the bases of a matrix for the subsequent implementations)
  - Matrix Builder (Interface that establish the bases of a matrix builder for the subsequent implementations)

- Project
  - test
    - MatrixMultiplicationTest (class to make test from the module Junit to check the consistence of the results)
    - MatrixMultiplicationBenchmark (class that implements JMH to measure the average time and throughput of the implementations)
    - RandomMatrixGenerator (class to create a random matrix of the indicated size for the experiments)

3.2 Matrix Operations in threads

Calculate smallest submatrix size

The primary objective of threads is to execute a portion of a split task. When dealing with matrices, it's logical to divide them into a number of submatrices, maximizing the count just below the number of available logical cores. To accomplish this, we have devised the following functions:

```java
    private static int calculateSmallestSubmatrixSize(int size, int numThreads) {
        int smallestSubmatrixSize = 1;
        while (true) {
            if (size % smallestSubmatrixSize == 0){
                int submatricesRow = size / smallestSubmatrixSize;
                int totalSubmatrices = submatricesRow * submatricesRow;
                if (totalSubmatrices <= numThreads) {
                    break;
                }
            }
            smallestSubmatrixSize++;
        }
        return smallestSubmatrixSize;
    }

    private static int getNumberOfThreads() {
        return Runtime.getRuntime().availableProcessors();
    }
```

Listing 1: Calculate Smallest Submatrix size

As we can see, we increment the smallestSubmatrixSize by one. This smallestSubmatrixSize is crucial to ensure that the total number of submatrices, calculated by multiplying the number of matrices per row and the number of matrices per column (since we are working with square matrices, these values are equal), remains less than or equal to the number of available threads.

Extract a submatrix

We utilize this function to extract a particular segment of the matrix. It necessitates specifying the indices of the initial row and column, as well as the indices of the concluding row and column that define the region of interest within the matrix. To facilitate this, we employ a DenseMatrixBuilder to generate a submatrix by incorporating the values obtained from this designated section of the original matrix. The following code snippet illustrates this process:

```java
public  Matrix<Long> extractSubMatrix(DenseMatrix matrix, int startRow, int endRow, int startCol, int endCol, int matrixSize) {
    DenseMatrixBuilder  denseMatrixBuilder = new DenseMatrixBuilder(matrixSize);
    int iSub = 0;
    for(int i = startRow; i < endRow; i++){
        int jSub = 0;
        for(int j = startCol; j < endCol; j++){
            denseMatrixBuilder.set(iSub, jSub, matrix.get(i, j));
            jSub += 1;
        }
        iSub +=1;
    }
    return denseMatrixBuilder.get();
}
```

Listing 2: Extract Submatrix

Divide a Matrix

We require an array of DenseMatrix objects to provide input to the threads. Through iterations of the extractSubMatrix function, we adjust the indices based on the submatrix size and store the results in an array.

```java
public DenseMatrix[] divideDenseMatrix(DenseMatrix matrix){
    int matrixSize = calculateSmallestSubmatrixSize(matrix.size(), getNumberOfThreads());
    int numberOfMatrices = (matrix.size() / matrixSize) * (matrix.size() / matrixSize);
    int numberOfRows = matrix.size() / matrixSize;
    DenseMatrix[] denseMatrices = new DenseMatrix[numberOfMatrices];
    int matrixNumber = 0;
    for (int i = 0; i < numberOfRows; i++) {
        for (int j = 0; j < numberOfRows; j++) {
            int startRow = i * matrixSize;
            int endRow = startRow + matrixSize;
            int startCol = j * matrixSize;
            int endCol = startCol + matrixSize;

            DenseMatrix subMatrix = (DenseMatrix) extractSubMatrix(matrix, startRow, endRow, startCol, endCol, matrixSize);
            denseMatrices[matrixNumber] = subMatrix;
```

```
                matrixNumber += 1;
            }
        }
        return denseMatrices;
    }
```

Listing 3: Dividing matrix

Rotate Matrices

If we split the matrix in several submatrices we can not simply operate them as a multiplication between submatrices in the same position. We will need to rotate the matrix based on the number of submatrices. The initial alignment will always be to have the last column or row moved one to the left or one up, we will just see it in the following example:

Suppose we have a multiplication of a matrix A multiplied by a matrix B, where each of the is divided in 2x2 submatrices(indicating by original position (row-column)):

$$A = \begin{pmatrix} 1-1 & 1-2 \\ 2-1 & 2-2 \end{pmatrix}$$

$$B = \begin{pmatrix} 1-1 & 1-2 \\ 2-1 & 2-2 \end{pmatrix}$$

Example of Divided Dense Matrices

We need to rotate the matrix A by row (as is the first matrix) and the matrix B by column (as is the second matrix). In the first iteration will need to rotate the last row of A to one to the left and the last column of B to one up. This is called initial alignment:

$$A = \begin{pmatrix} 1-1 & 1-2 \\ 2-2 & 2-1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1-1 & 2-2 \\ 2-1 & 1-2 \end{pmatrix}$$

Example of Initial Alignment

We perform the multiplication as usually and store the results in a array. Then we perform a second iteration. We need to iterate as the numbers of submatrices by row we have. So the second iteration will involve again the last row and column returning them to their original state and will alter the first row and column rotated them one to the left and up respectively:

$$A = \begin{pmatrix} 1-2 & 1-1 \\ 2-1 & 2-2 \end{pmatrix}$$

$$B = \begin{pmatrix} 2-1 & 1-2 \\ 1-1 & 2-2 \end{pmatrix}$$

Example of Last Alignment

Finally, the multiplication of these two submatrices is executed, and the result is stored in an array. Each element in for the result array is obtained by summing the corresponding elements of the DenseMatrix[] arrays which share position. It is crucial to emphasize that our code is designed exclusively for square matrices with a size that is a power of two. The outcome of this operation is an array of DenseMatrix[], necessitating a recomposition.

In summary, for matrix rotation, the process commences in the first iteration with the last row or column, and in subsequent iterations, a new row above or column to the left is incorporated. This iterative procedure repeats for the number of submatrices by row (or column).

We have implement it with the following code:

```java
public DenseMatrix[] rotateMatricesByRow(DenseMatrix[]
denseMatrices,  int iteration) {
    int numberOfRows = (int) Math.sqrt(denseMatrices.length);
    for (int i = 0; i < iteration; i++){
        int lastIndexRow = denseMatrices.length-1-(i*
numberOfRows);
        int firstIndexRow = denseMatrices.length-1-((i+1)*
numberOfRows-1);
        DenseMatrix temp = denseMatrices[firstIndexRow];
        for (int j = firstIndexRow; j < lastIndexRow; j++) {
            denseMatrices[j] = denseMatrices[j + 1];
        }
        denseMatrices[lastIndexRow] = temp;
    }
    return denseMatrices;
}

public DenseMatrix[] rotateMatricesByColumn(DenseMatrix[]
denseMatrices, int iteration){
    int numberOfRows = (int) Math.sqrt(denseMatrices.length);
    for(int i = 0; i < iteration;i++){
        int lastIndexColumn = denseMatrices.length-1-i;
        int firstIndexColumn = lastIndexColumn-numberOfRows*(numberOfRows-1);
        DenseMatrix temp = denseMatrices[firstIndexColumn];
        for (int j = firstIndexColumn; j < lastIndexColumn; j+=numberOfRows){
            denseMatrices[j] = denseMatrices[j+numberOfRows];
        }
        denseMatrices[lastIndexColumn] = temp;
    }
    return denseMatrices;
}
}
```

Listing 4: Rotating Matrices

Compose Matrix

The outcomes from the parallel matrix multiplication operations using threads generate an array of Dense Matrices. To reconstruct the original matrix, we leverage information about the number of rows and columns. This process involves sequentially populating the matrix by filling rows from left to right. After completing a row, the column index is reset to 0, and the row index is incremented. This transformation is crucial for result verification in tests.

The corresponding code is as follows:

```java
public DenseMatrix composeMatrix(DenseMatrix[] denseMatrices){
    int subMatrixSize = denseMatrices[0].size();
    int numRowsOriginal = (int) Math.sqrt(denseMatrices.length);

    DenseMatrixBuilder denseMatrixBuilder = new DenseMatrixBuilder(numRowsOriginal * subMatrixSize);

    int rowIndex = 0;
    int colIndex = 0;

    for (DenseMatrix matrix : denseMatrices) {
        if (matrix != null) {
            for (int i = 0; i < subMatrixSize; i++) {
                for (int j = 0; j < subMatrixSize; j++) {
                    denseMatrixBuilder.set(rowIndex * subMatrixSize + i, colIndex * subMatrixSize + j, matrix.get(i, j));
                }
            }
        }

        colIndex +=1;

        if (colIndex == numRowsOriginal) {
            colIndex = 0;
            rowIndex++;
        }
    }
```

```

        return (DenseMatrix) denseMatrixBuilder.get();
    }
```

Listing 5: Compose Matrix

3.3 Thread Extensions

In this research, we have chosen not to utilize the ExecutorService for parallel operations. Instead, we adopt a custom approach by creating two thread classes—one for matrix multiplication (ThreadMultiplication) and another for summing the results (ThreadSum). Each thread class extends the Java Thread class, inheriting all the functionalities required for a runnable thread. A thread executes a task specified in its run method.

For the ThreadMultiplication class, matrices to be multiplied are passed as parameters, and the run method simply calls the multiplyDense method (from MatrixOperations), as seen in previous works. The result is stored in a variable with a getter for retrieval.

Conversely, the ThreadSum class requires a list of results, organized as arrays of Dense Matrices. This is because the results of each multiplication submatrix are stored in an array for each iteration, necessitating grouping. The class also needs the index of the position of the array it is responsible for summing (to sum the submatrices in the same area of the original matrix). It includes a run method for iterating through the list of results to perform the summation and a getter to extract the final result.

The implementation for ThreadSum will be:

```
    ppublic class ThreadSum extends Thread {
    private List<DenseMatrix[]> listResults;
    private int columnIndex;
    private MatrixOperations matrixOperations;
    private DenseMatrix resultMatrix;

    public ThreadSum(List<DenseMatrix[]> listResults, int
    columnIndex, MatrixOperations matrixOperations) {
        this.listResults = listResults;
        this.columnIndex = columnIndex;
        this.matrixOperations = matrixOperations;
    }

    @Override
    public void run() {
        int size = listResults.size();
        DenseMatrix currentMatrix = listResults.get(0)[columnIndex
    ];

        for (int k = 1; k < size; k++) {
            currentMatrix = matrixOperations.sumDense(currentMatrix
    , listResults.get(k)[columnIndex]);
        }
```

```java

        resultMatrix = currentMatrix;
    }

    public DenseMatrix getResultMatrix() {
        return resultMatrix;
    }
}
```

Listing 6: ThreadSum Implementation

The implementation of ThreadMultiplication is quite similar so it can be reviewed in the attached source code.

3.4 Operating with Threads

Thread objects can be instantiated based on the prior implementations to carry out specific tasks. The number of threads to be created should match the number of tasks to be executed simultaneously. In the case of both addition and multiplication operations, the number of threads required equals the count of submatrices resulting from splitting the original matrix.

Once the threads are created, the .start() method is invoked to activate them and initiate the operations. Subsequently, a loop is employed to await the completion of all threads through the use of the .join() method for each thread. This ensures that, before accessing and storing the results obtained from the getter method, the outcomes are coherent.

To illustrate, let's revisit the example for sumSubmatrices. The implementation for multiplying by threads(multiplyByThreads) is akin to this process.

```java
    private DenseMatrix[] sumSubmatrices(List<DenseMatrix[]> listResults) {
        int numColumns = listResults.get(0).length;
        DenseMatrix[] sumMatrices = new DenseMatrix[numColumns];
        Thread[] threads = new ThreadSum[numColumns];
        MatrixOperations matrixOperations = new MatrixOperations();

        for (int i = 0; i < numColumns; i++) {
            threads[i] = new ThreadSum(listResults, i, matrixOperations);
            threads[i].start();
        }

        for (int i = 0; i < numColumns; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (threads[i] instanceof ThreadSum) {
                ThreadSum threadSum = (ThreadSum) threads[i];
                sumMatrices[i] = threadSum.getResultMatrix();
            }
        }

        return sumMatrices;
    }
```
Listing 7: Using ThreadSum

3.5 Putting all together

The key function encapsulating the majority of the threaded matrix multiplication process is multiplySubmatrices. This function orchestrates matrix rotation, invokes the multiplyByThreads function, and aggregates its results (Dense Matrix array) into lists. Subsequently, these lists are passed as arguments to the sumSubmatrices function, and the final result of the entire process is obtained using composeMatrix.

The code will be the following one:

```java
    public DenseMatrix multiplySubmatrices(DenseMatrix[] matricesA,
     DenseMatrix[] matricesB){
        int numberOfRows = (int) Math.sqrt(matricesA.length);
        List<DenseMatrix[]> listResults = new ArrayList<>();
        for(int i = 1; i <= numberOfRows; i++){
            matricesA  = rotateMatricesByRow(matricesA.clone(), i);
            matricesB = rotateMatricesByColumn(matricesB.clone(), i
    );

            DenseMatrix[] denseMatricesResult = multiplyByThreads(
    matricesA, matricesB);
            listResults.add(denseMatricesResult);
        }
        DenseMatrix[] resultMatrices = sumSubmatrices(listResults);
        return composeMatrix(resultMatrices);
    }
```
Listing 8: multiplySubmatrices

3.6 Test JMH

The MatrixMultiplicationBenchmark incorporates methods to measure execution time in milliseconds per operation and throughput in operations per millisecond. We employ four groups of iterations, with two dedicated to warm-up (specified in @Fork). The warm-up and measurement consist of 5 iterations each, lasting one second. The benchmark runs twice, once for throughput and the other for average time. For operands, we utilize two random matrices created with the RandomMatrixGenerator class.

The benchmark covers sizes specified in @Params, ranging from 2 to 1024 in the power series of two. Three tests are conducted: one for Dense Sequential Multiplication, another for the Threads implementation, and the last one for Streams.

The code is similar to the ones we have used in previous work so it can be checked it there or in the source code.

3.7 Test Junit

The primary objective is to verify the equivalence of results between the Threads and Streams implementations with those of the sequential method. The tests are conducted using the same sizes as specified in the JMH tests.

The test code is as follows:

```java
@Test
public static void testMatrixMultiplicationParallelism(){
    DenseMatrix matrix = submatricesOperations.multiplySubmatrices(divideMatrixA(), divideMatrixB());
    assertEquals(initializeExpectedResultConventional().getValues(), matrix.getValues());
    System.out.println("Completed test using threads implementation with size " + matrix.size() + "x" + matrix.size());
    System.out.println("Expected result from dense multiplication without parallelism is equal to the result from the threads implementation!");
}

@Test
public  static  void testMatrixMultiplicationStreams(){
    DenseMatrix matrix = streams.multiplyDense(matrixA, matrixB);
    assertEquals(initializeExpectedResultConventional().getValues(), matrix.getValues());
    System.out.println("Completed test using streams implementation with size " + matrix.size() + "x" + matrix.size());
    System.out.println("Expected result from dense multiplication without parallelism is equal to the result from the streams implementation!");
}
```

Listing 9: Tests Junit

4 Experiment

4.1 Results Junit tests

The outcomes of this test reveal that both the Threads and Streams implementations yield identical results to the sequential implementation. However, this testing approach provides a more lightweight perspective on execution time, as illustrated in the following table:

As depicted in the table, the execution times during the verification with sequential multiplication show similarity between the two implementations. To gain a clearer perspective, let's visualize the data with a graph:

Size	Threads (in ms)	Streams (in ms)
2	31	10
4	11	3
8	7	2
16	16	3
32	12	4
64	24	13
128	43	76
256	261	200
512	1636	1385
1024	20550	20086

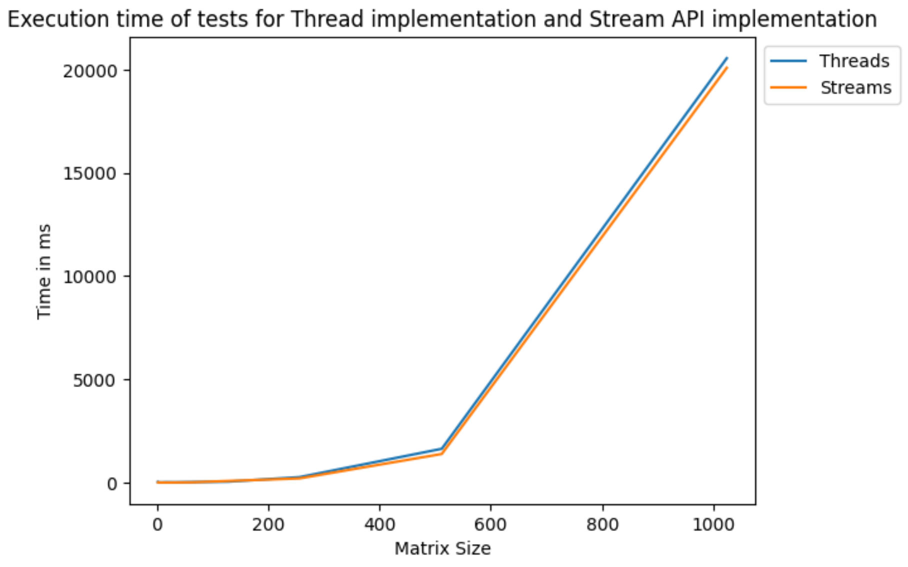

Figure 1: Execution Time test

From the graph shown in 1, it appears that the manual threads implementation takes slightly more time than the Streams implementation. However, for a more accurate assessment, we need to calculate the speedup and graphical differences based on the more elaborate JMH Test results. It's essential to note that JUnit primarily checks if two results are the same, so it is not a solid base.

4.2 Results JMH Tests

Average Time

In this case the obtained tables are similar to ones we have seen in our previous work, as they have a consider size and do not add that much value, we will proceed with the graphs.

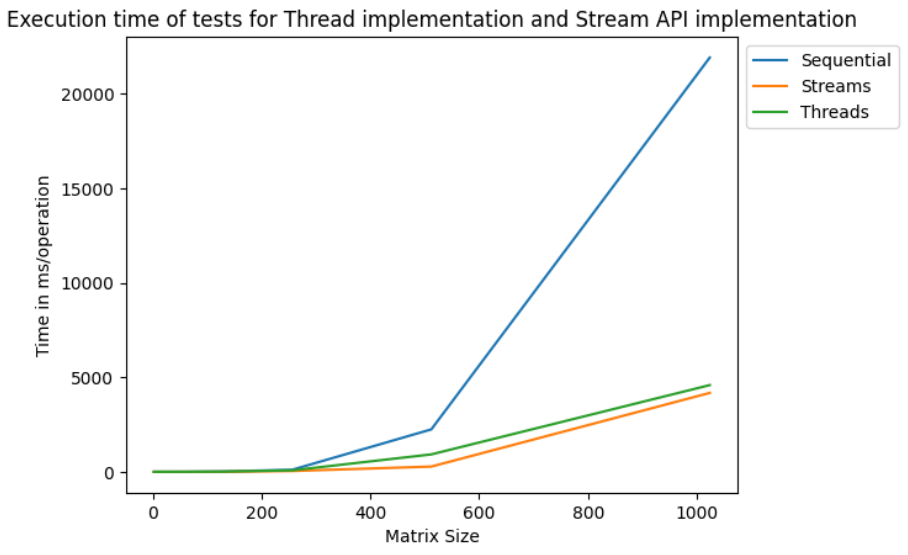

Figure 2: JMH Average time test

As depicted in 2, it is evident that the Streams execution solution outperforms the manually created thread approach for our specific problem. However, it is worth noting that both of these parallel implementations significantly outshine the sequential one. The differences observed between the parallel methods are not statistically significant. Now, let's examine the speedup measure:

Lets check out the means of execution time:

The means in milliseconds are:

	Mean of execution time in ms/operation
Sequential	2429.906710
Threads	562.21820
Streams	450.66520

An observation reveals a marginal difference of approximately 110 milliseconds per operation between Threads and Streams, which is not substantial. However, the disparities with the Sequential approach are quite substantial. As depicted in the graph, their growth is more pronounced, especially at the inflection point. This information underscores that the baseline for our speedup calculation should be the execution time of the sequential method. We will compute the speedup using the mean.

The outcomes from the speedup indicator are as follows

The results from the speedup indicator are:

	Mean of execution time in ms/operation
Threads	4.321999
Streams	5.391822

It is evident from our analysis that Threads exhibit a performance improvement of 4.3 times over sequential execution, whereas Streams demonstrate an even more substantial enhancement with a speedup of 5.3 times. Although these speedup values may seem relatively modest, it is crucial to recognize the significant efficiency gains, especially in the context of large matrix multiplications like 1024, as clearly illustrated in the accompanying graph.

Throughput

Throughput, in our context, is defined as the number of operations per millisecond. This metric attempts to consider the workload balance of a thread. Consequently, it is noteworthy that, in this scenario, the sequential mode is anticipated to yield the highest throughput value. The accompanying graphs are presented below for visual reference:

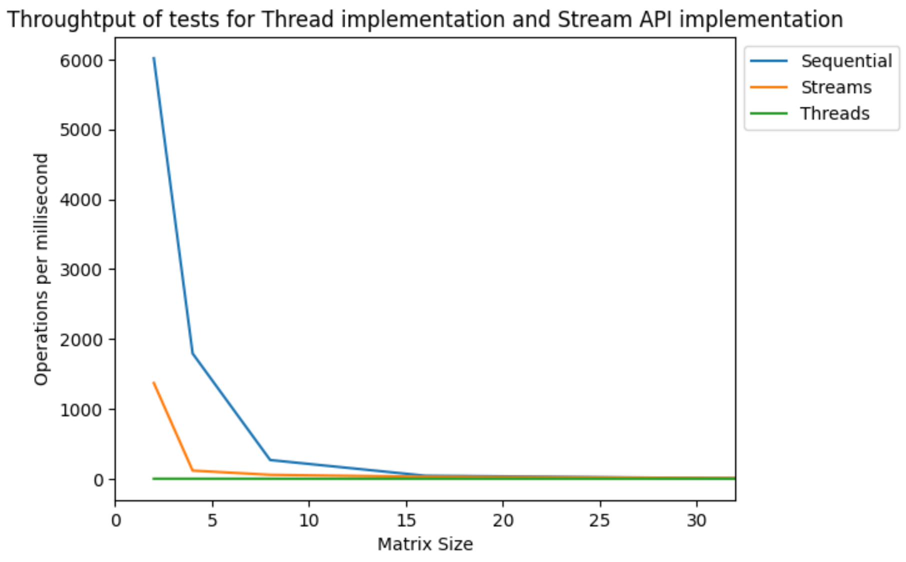

Figure 3: JMH Throughput until 128 matrix size

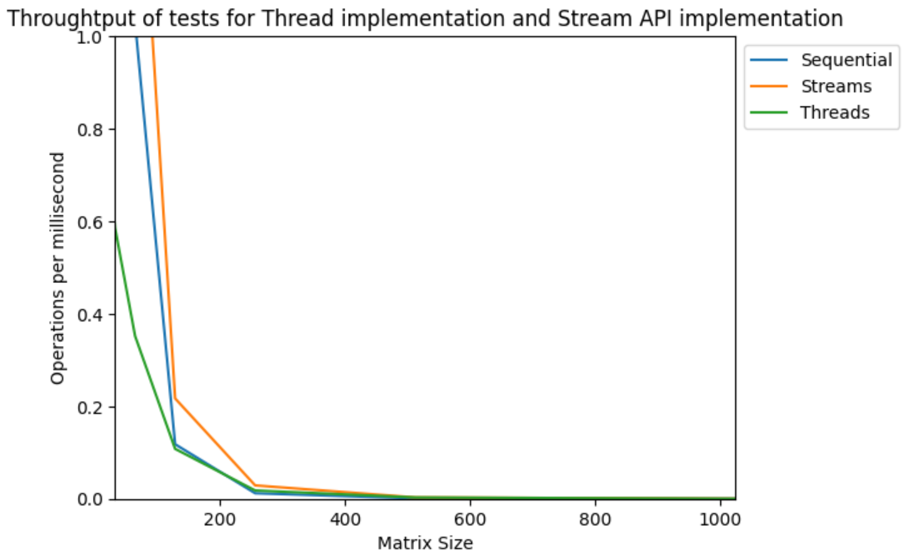

Figure 4: JMH Throughput from 128 matrix size

As anticipated, as depicted in figures 3 and 4, the sequential execution mode exhibits the highest throughput, surpassing 6000 operations per millisecond. In contrast, the throughput for Streams is initially lower, ranging between 1000 and 2000 operations per millisecond.

Notably, the Threads implementation maintains a consistent throughput, hovering between 0 and 1 operations per millisecond across all matrix sizes. This observation suggests that each thread achieves a more equitable distribution of tasks compared to the other modes.

In 4 we can see that Streams is over passed in some moments by Sequential in terms of a lower throughput. This makes difficult to determine how make a better load balance when we reach higher matrices. Maybe another possibility could be that the operations become more heavy in sequential so the number of them per milliseconds is significantly reduced while the execution time increase.

To dispel any uncertainties, we will delve into the speedup indicator shortly. However, as a preliminary step, let's examine the mean of throughputs:

	Mean of throughput in operations/ms
Sequential	813.10021
Threads	0.46791
Streams	158.43623

As observed, the sequential mode has the highest mean throughput, a trend discernible in Figure 3. The anticipated outcome for the Threads implementation is affirmed, as evidenced by a throughput mean of 0.47 operations/ms.

While the speedup indicator appears intuitively evident in this context, we will further validate it by calculating it, with the mean throughput of the sequential mode serving as our reference base:

	Mean of throughput in operations/ms compared to Sequential
Threads	1737.727790
Streams	5.132035

Based on the preceding table, a notable observation is that the Stream implementation exhibits nearly five times lower throughput than the sequential mode. In contrast, the Threads throughput is considerably lower, being 1737.73 times less than the Sequential throughput. This stark contrast in performance can be attributed to various factors, with the distribution of tasks emerging as a primary contributing factor.

5 Conclusion

Our expectations regarding the reduction in matrix multiplication time through novel distributed programming approaches have been substantiated. Both Threads and Streams implementations have demonstrated superior performance compared to the Sequential mode. However, it is imperative to acknowledge that the efficacy of these alternative approaches is intricately tied to our hardware resources, specifically the number of CPU cores at our disposal.

The average execution time has seen a remarkable reduction of over 4 times in both implementations, with Streams emerging as the fastest among them. The throughput metric provides valuable insights into the distribution of the workload, notably showcasing the Threads implementation with the minimum operations per millisecond. While open to various interpretations, a plausible explanation could be the manual assignment of tasks to each thread, indicating a deliberate and controlled approach to task allocation.

In the context of the Streams API implementation, it's known that Java operates in parallel with data collections, leveraging the parallel() mechanism. The intricacies of this mechanism, however, remain hidden from our direct view. It is plausible that it is set to use the minimum necessary cores while optimizing for time efficiency. Conversely, in the case of the sequential implementation, we have observed a discernible performance decline, likely influenced by the escalating sizes of matrices, making operations progressively more cumbersome.

In conclusion, the Streams API implementation is a recommended choice if your primary goal is to enhance execution time while abstracting away from throughput and task management intricacies. On the other hand, if your priority is to ensure a balanced distribution of operations per unit of time among threads, coupled with a more than decent execution time, the Threads implementation, with manual creation, would be the preferred approach. The choice depends on the specific priorities and considerations in optimizing performance.

A noteworthy point to consider is that the use of an HDD disk is not advisable for tasks of this nature. The experiments were conducted on an HDD, so the execution time might be higher compared to an SSD. It's crucial to acknowledge that the results of this experiment can vary based on your machine specifications.

In summary, optimizing execution time is significantly more efficient with distributed programming compared to sequential operations. If the necessary resources are available, adopting a distributed approach can lead to substantial improvements in performance while maintaining result accuracy. The sequential execution mode, on the other hand, should not be the default choice unless there are specific constraints necessitating its use.

6 Future Work

For a future research, a more comprehensive perspective could be achieved by expanding the matrix multiplication analysis beyond square matrices. Presently, the current program is restricted to square matrices, highlighting a potential area for future development.

Furthermore, based on insights gained from our preceding work, there is potential for performance enhancement through a synergy between sparse matrix formats and parallel programming. It's worth noting that the implementation of the sparse matrix format should leverage Java primitive data types to fully capitalize on the potential advantages. This avenue presents an intriguing opportunity for further exploration and optimization in future studies.