Matrix Multiplication in Java

Elisa-Marie Breeze Rebstock

November 2024

Abstract

In this paper, different algorithms are going to be evaluated for matrix multiplication in Java, focusing on three different multiplication methods: basic matrix multiplication, parallel matrix multiplication, and vectorized matrix multiplication. The performance of each algorithm is measured in terms of execution time and memory usage through benchmark tests on matrices of varying sizes, showing that parallelism and vectorization can improve the performance of matrix multiplication significantly compared to the basic approach.

1 Introduction

In this paper you will see the implementation and analysis of the performance of three different matrix multiplication algorithms: the basic multiplication, vectorized and parallel multiplication, and in the case of the parallel method, it will be tested with different numbers of threads. We will focus on the Java programming language, and implement numerous different algorithms, testing them each with benchmarks and memory usage calculations.

2 Methodology

Link to the GitHub Repository of this assignment:
https://github.com/ElisaBreeze/BigData.git

To compare the different algorithms, I created matrices with different sizes. I used these matrices in the different algorithms to see which one performed better: basic, using parallelism, or vectorized. Based off the first and second tasks, I used the same benchmark method, as well as recycling the basic matrix multiplication algorithm.

For the memory usage calculation, I used a new method, which is simpler and doesn't affect the benchmark as much: adding "-prof gc" to the arguments in the configuration of the benchmark file.

The other algorithms used in this investigation were researched thoroughly and then I applied them to the project. Each one of the new algorithms will be explained in a more extensive way in its subsection:

2.1 Basic Matrix Multiplication

Here we can see the basic matrix multiplication code, that uses 3 loops to multiplicate matrix A and B, creating the result matrix, C. This code was taken from the task 1, where I decided to implement a similar code to the example code given by the teachers, but making a few changes that made the code look tidier and simpler: in the creation of the matrix in the testing code, I added the solution matrix (c) to the setup code, in order to have it all in one place and be more organized.

I also passed on the size of the matrix as a value to the matrix multiplication function, as it was already initialized for the setup function. This way, the code was more optimized because it didn't have to calculate the matrix size in the matrix multiplication code, but instead it was given to it automatically.

```
public double[][] multiply(double[][] a, double[][] b, double[][] c,
     int n){
       for (int i = 0; i < n; i++) {
           for (int j = 0; j < n; j++) {
               for (int k = 0; k < n; k++) {
                   c[i][j] += a[i][k] * b[k][j];
               }
           }
       }
       return c;
}
```

2.2 Matrix Multiplication using Parallelisms

The idea of Parallel matrix multiplication is to speed up the execution time by dividing the work with multiple threads. In this implementation, we use Java's "ExecutorService" to manage a pool of threads, and each thread will compute one row of the result matrix in parallel. With this, the idea is to take advantage of using more than one processor core to execute the multiplication concurrently. In the test, we try with 2, 4 and 8 threads. Using "ExecutorService", helps us to manage more easily the threads, making sure they are initialized and terminated properly.

The relevant information regarding this algorithm was sourced from the theory classes, as well as information gathered from other courses of this Degree, like for example IC2 and DACD.

```
public static double[][] multiply(double[][] a, double[][] b, double
     [][] c, int n, int threads) throws InterruptedException {
       int numThreads = threads;
       ExecutorService executor = Executors.newFixedThreadPool(
            numThreads);
       for (int i = 0; i < n; i++) {
           final int row = i;
           executor.submit(() -> {
               for (int j = 0; j < n; j++) {
                   for (int k = 0; k < n; k++) {
                       c[row][j] += a[row][k] * b[k][j];
                   }
               }
           });
       }
       executor.shutdown();
       executor.awaitTermination(1, TimeUnit.HOURS);
       return c;
}
```

2.3 Matrix Multiplication using Vectorization

Vectorized matrix multiplication is an optimized approach that uses vectorization, which takes advantage of being able to handle multiple operations at the same time. The difference to the Parallel multiplication method, is that the parallel task is split across multiple threads, running in parallel, while in vectorized multiplication the process is optimized by the library or hardware without explicitly using threads, meaning it does it automatically.

In summary, parallel multiplication focuses on dividing the task into multiple smaller tasks, while vectorized multiplication focuses on optimizing individual operations by using CPU's ability to handle more than 1 element in a single operation.

In this method, I used the Apache Commons Math3 library, which allowed to do the operations in a much easier and more efficient way compared to the other methods.

Relevant information regarding this algorithm was sourced from the following website: - dev.to

```
public static RealMatrix multiply(RealMatrix A, RealMatrix B) {
      if (A.getColumnDimension() != B.getRowDimension()) {
          throw new IllegalArgumentException("Dimensions␣do␣not␣match␣
               requirements");
      }
      return A.multiply(B);
}
```

3 Experiments and Results

3.1 Benchmark Methodology

All the algorithms were tested through benchmark and memory usage calculation, using various matrices of varying sizes and in the case of the Parallel Method, different number of threads. The results were taken from the benchmark output, which includes the memory usage information by adding "-prof gc" to the program arguments before. From the end output of the benchmark, we then take the average execution time, and the value of "gc.alloc.rate.norm" for the memory usage. The results are shown in the result section below, where I show a table for each algorithm with the results for the different values tested, and after that, a new table with the efficiency and speedup values.

3.1.1 Benchmark

```
public class Benchmark {

    @State(Scope.Thread)
    public static class Operands {

        @Param({"16", "128", "1024", "2048"})
        public int matrixSize;
        double[][] a;
        double[][] b;
        double[][] c;

        @Setup
        public void setup() {
            a = new double[matrixSize][matrixSize];
            b = new double[matrixSize][matrixSize];
            c = new double[matrixSize][matrixSize];
            Random random = new Random();

            for (int i = 0; i < matrixSize; i++) {
                for (int j = 0; j < matrixSize; j++) {
                    a[i][j] = random.nextDouble();
                    b[i][j] = random.nextDouble();
                }
            }
        }
    }

    @State(Scope.Thread)
    public static class ThreadOperands {
        @Param({"2", "4", "8"})
        public int threads;
    }

    @org.openjdk.jmh.annotations.Benchmark
    public void benchmarkBasicMatrixMultiplication(Operands operands) {
        new BasicMatrixMultiplication().multiply(operands.a, operands.b,
              operands.c, operands.matrixSize);
    }

    @org.openjdk.jmh.annotations.Benchmark
    public void benchmarkParallelMatrixMultiplication(Operands operands,
          ThreadOperands threadOperands) throws InterruptedException {
        new ParallelMatrixMultiplication().multiply(operands.a, operands
             .b, operands.c, operands.matrixSize, threadOperands.threads
             );
    }

    @org.openjdk.jmh.annotations.Benchmark
    public void benchmarkMatrixMultiplicationVectorized(Operands
         operands) {
        new MatrixMultiplicationVectorized().multiply(MatrixUtils.
             createRealMatrix(operands.a), MatrixUtils.createRealMatrix(
             operands.b));
    }
}
```

3.2 Performance Results

The performance of each algorithm is measured in terms of execution time and memory usage. Each test was executed various times to ensure consistency. Below you can see the different algorithms with their respective results, for the chosen values and executed on the same laptop (MacOS M3) and environment (IntelliJ IDEA):

3.2.1 Basic Multiplication results

Matrix Size	Execution Time (ms)	Memory Usage (MB)
16x16	0.004	9.54×10^{-11}
128x128	2.23	9.44×10^{-5}
1024x1024	2453.59	0.103
2048x2048	68869.03	0.435

3.2.2 Multiplication using Parallelism Results

Matrix Size	Thread Number	Execution Time (ms)	Memory Usage (MB)
16x16	2	0.178	0.004
16x16	4	0.372	0.005
16x16	8	0.814	0.008
128x128	2	1.28	0.016
128x128	4	1.02	0.018
128x128	8	1.16	0.021
1024x1024	2	1155.66	0.112
1024x1024	4	769.00	0.113
1024x1024	8	776.66	0.116
2048x2048	2	26579.14	0.221
2048x2048	4	16769.12	0.222
2048x2048	8	14527.59	0.230

3.2.3 Vectorized Multiplication results

Matrix Size	Execution Time (ms)	Memory Usage (MB)
16x16	0.006	0.0069
128x128	1.41	0.375
1024x1024	660.12	24.04
2048x2048	5134.44	96.1

3.3 Speedup and Efficiency calculation

To calculate this, I used different formulas, all of the data with respect to the Basic Matrix Multiplication:

Speedup: BasicMethodTime/OptimizedMethodTime

Efficiency: Speedup/NumberOfThreads

Multiplication Type	Matrix Size	Thread Numbers	Speedup	Efficiency
Parallel	16x16	2	0.0225	0.0112
Parallel	16x16	4	0.0108	0.0027
Parallel	16x16	8	0.0049	0.0006
Parallel	128x128	2	1.7422	0.8711
Parallel	128x128	4	2.1863	0.5466
Parallel	128x128	8	1.9224	0.2403
Parallel	1024x1024	2	2.1231	1.0616
Parallel	1024x1024	4	3.1906	0.7977
Parallel	1024x1024	8	3.1592	0.3949
Parallel	2048x2048	2	2.5911	1.2955
Parallel	2048x2048	4	4.1069	1.0267
Parallel	2048x2048	8	4.7406	0.5926
Vectorized	16x16	-	0.5714	-
Vectorized	128x128	-	1.5816	-
Vectorized	1024x1024	-	3.4777	-
Vectorized	2048x2048	-	13.4132	-

4 Result Discussion

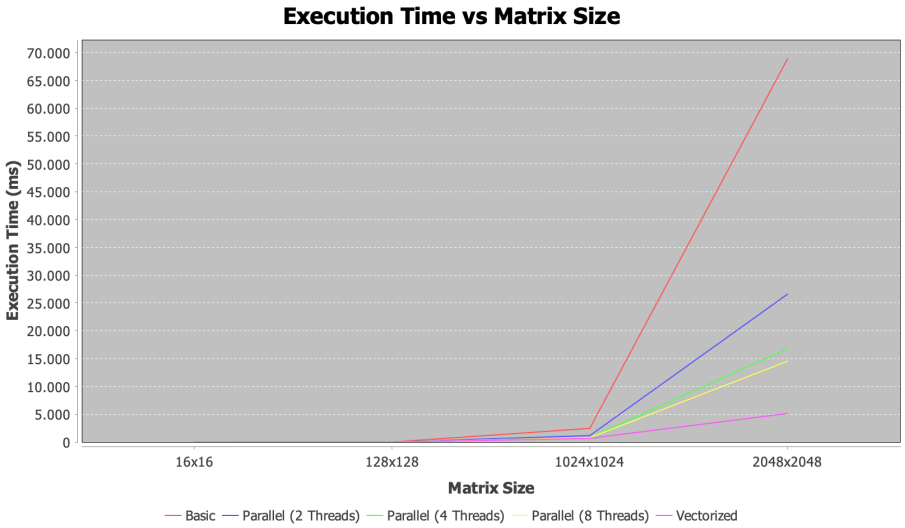
Evaluating the results of the different algorithms across different matrix sizes and sparsity levels, it is possible to discuss several important aspects:

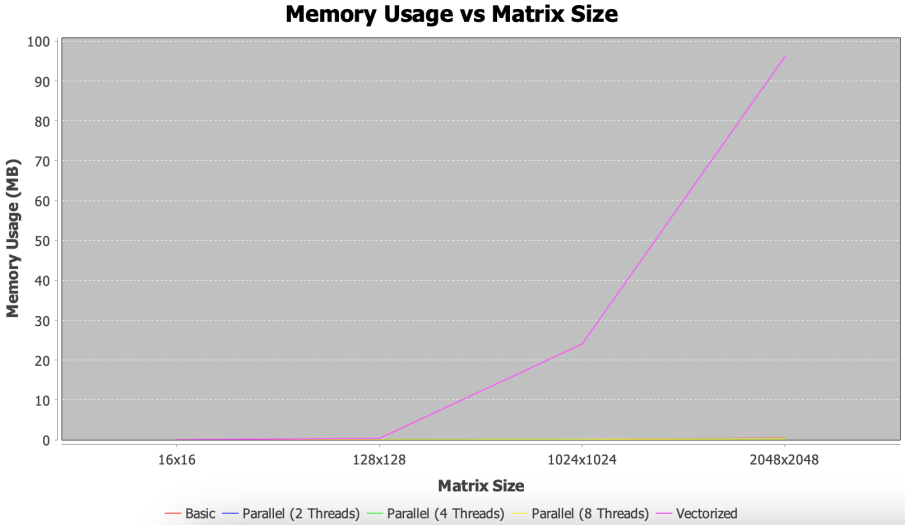
Basic Matrix Multiplication: In this case, we can see how the execution time increases by a lot as the matrix size grows. For the small matrix, the execution time is very low, but as the size of the matrix grows, the time grows exponentially, reaching 69 seconds for matrix size 2048x2048. These results are normal and expected for the basic multiplication method, as the complexity of the algorithm increases by $O(n\hat{3})$. When looking at the memory usage, we see that it also grows slightly with the matrix sizes, but the values stay low in general.

Parallel Matrix Multiplication: The results show that it reduces the execution time significantly for larger matrices compared to the basic multiplication, but for smaller matrices it seems to be slower. When comparing the different thread numbers, what we see is that in general the more threads there are, it it increases the memory usage, but we can also see that the execution time is reduced as the matrix size grows, when using more threads. On the other hand, the speedup improves with the matrix size and thread count, but you have to take into account that for smaller matrices this value is not very good. The efficiency drops a lot as the number of threads increase, especially for small matrices, which means that it is not very good when using resources, but for bigger matrices is shows a better efficiency.

Vectorized Matrix Multiplication: When looking at these results, we can see that this method has much lower execution times than both the basic and parallel methods, which makes it clear that it's the best method of the three tested, especially as the matrix size increases, which shows us that it's way more efficient. The memory usage on the other hand is much higher compared to the other two methods, which not so good. Looking at the speedup values, we can see that it shows the highest values for all sizes, which shows that this method is the most effective for optimization, if we don't take into account the high memory usage.

Here are the results shown graphically in the next two charts:


Execution Time vs Matrix Size
Execution Time (ms)
70.000
65.000
60.000
55.000
50.000
45.000
40.000
35.000
30.000
25.000
20.000
15.000
10.000
5.000
0
16x16
128x128
1024x1024
2048x2048
Matrix Size
Basic
Parallel (2 Threads)
Parallel (4 Threads)
Parallel (8 Threads)
Vectorized


Memory Usage vs Matrix Size
Memory Usage (MB)
100
90
80
70
60
50
40
30
20
10
0
16x16
128x128
1024x1024
2048x2048
Matrix Size
Basic
Parallel (2 Threads)
Parallel (4 Threads)
Parallel (8 Threads)
Vectorized

5 Conclusion

In conclusion, as we can see from the results of the benchmark and memory usage calculation, that the basic matrix multiplication is very inefficient for big matrices, and that the parallelization method has a better performance, especially as the matrix increases its size. On the other hand, the vectorized multiplication has an important speedup across all sizes, with high efficiency especially as the matrix grows. It's clear that this is the best method as it has the lowest execution time and it shows very clearly that it can take big matrices well. With this, it's important to highlight the problem of the memory usage in this method, as it's much higher than the memory usage in Basic and Parallel multiplication. Although this is a negative point for the vectorized method, in my opinion it's still the best option, as the execution times are so much better, but ultimately this decision depends on the specific project requirements, and what you desire to prioritize.