Distribution of Matrix Multiplication

Wojciech Dróżdż

December 28, 2024

Abstract

This paper focuses on speeding up the matrix multiplication algorithm by splitting the multiplication task into multiple sub-tasks and executing them on independent computing nodes. A slight performance improvement was achieved, although the networking overhead had a strong negative impact on performance.
The code for this paper can be found in the GitHub repository

1 Introduction

In the previous paper optimization methods for multi-threaded matrix multiplication were considered. A step further from this is distribution of the load across multiple computing nodes over the network. In modern big data solutions distributed computing is often used to handle single large computations or multiple smaller ones. The matrix multiplication problem falls into the first, more difficult category. Thankfully there exist solutions, which allow for easier distribution of the load. One of them is *Hazelcast*, which was used in this paper.

2 Methodology

2.1 Hardware

All of the tests were performed on a 2023 16 inch MacBook Pro and a Lenovo laptop running an Arch Linux distribution: MacBook:

- Apple M3 Pro - 4 GHz 12-core CPU with 6 performance cores and 6 efficiency cores
- 18GB RAM
- 1024GB of internal storage

Lenovo:

- AMD Ryzen 4800U - 3.7GHz 8 core CPU with 16 threads
- 16GB RAM
- 512GB of internal storage

In order to preserve consistency all of the tests were performed while the devices were plugged into a power source.

2.2 Interpreters and Compilers

- *Java* - Java 17 with JetBrains runtime

2.3 Benchmarking Tools

The *Java Microbenchmark Harness (JHM)*, part of the *OpenJDK* project, was chosen for its microbenchmarking features.

2.4 Benchmarking Methodology

The Benchmarks measure only the time it takes to multiply the matricies, all of the memory is pre-allocated before running the multiplication method. The memory allocation time is not measured. Testing was started with 10 warmup iterations before full testing, allowing for just-in-time compilations and appropriate cache optimizations. This was followed by 10 test iterations during which the shortest, longest, and average execution times were recorded.

3 Distributed computing approach

3.1 Java

In the *Java* implementation improvements from the previus papers were used - each computing node was performing the calculations using the multi-threaded version of the blocked column multiplication algorithm for maximum performance.

3.1.1 Splitting the task and load balancing

The splitting of the task shows an interesting challenge, which has to be solved. The easiest solution is to split the matrix into N members where N is the number of nodes in the cluster. This causes a problem though, because the machines may have vastly different capabilities. For example in the case of this paper the CPU of the MacBook was performing the same calculations about 2-3 times faster than the Lenovo laptop. This means that if the task is split equally between the nodes the final execution time may be slower than if ran on just the fastest machine.

To combat this issue the matrix is split into 10 parts for each machine that joins the cluster. Since as far as I am aware *Hazelnut* doesn't provide a load balancing solution and splits the tasks equally between the nodes, a custom solution was implemented.

On the executing machine a thread is created for each of the nodes in the clusters. Each of those threads will launch a single task from the tasks stack, wait for it's completion and repeat until the tasks run out. This guarantees that the faster node will do more tasks, while the weaker one will do a smaller portion.

3.1.2 Distribution of the matrix

The largest issue with distributing matrix multiplication tasks is the overhead of networking. Three basic approaches were considered:

- Deliver the necessary information for the calculation on per-element basis - this approach was quickly dismissed, as it means that for each element of the array two 1D arrays would have to be delivered over the network, which comes up to $O(n^3)$ of total memory transfer per node where n is the width of the matrix (assuming a square matrix).
- Calculate in one chunk per node and deliver the whole arrays - this approach requires $O(n^2)$ of memory transfer per node, but makes the problem of unequally performant nodes mentioned above impossible to solve.
- Calculate in multiple chunks per node, but deliver the whole arrays only once - This is the approach, which was chosen for the final implementation. The execution was split into three parts - delivering the input data, calling the multiplication tasks, retrieving the results. After the data is delivered it is kept on the node together with the results, until a task which retrieves it is called. This allows me to keep the $O(n^2)$ transfer complexity, while keeping the option of distributing the load based on the performance of a node.

3.1.3 Networking overhead

In the experiments the results are shown only for calculations. Time used to transfer the matrix data between the nodes and retreiving them is not included. In general on average the transfer of data took more time than the calculations themselves, which makes this approach rather impractical. To minimize this issue the operations are split into three groups.

- Deliver the input data - both matrices are delivered to all of the nodes
- Perform the requested calculations - the node will receive multiple calculation calls, which will calculate parts of the array. And save the results on the node.
- Fetch the results - all of the nodes are sent a finish task, which returns a partial result. All of the partial results are combined into a one final array.

3.2 Python

The python implementation of the task is different from the Java implementation. This is, because the Python implementation of Hazelcast is more limited and would require additional Java server serialization configuration to work correctly. The following method was developed:

- First the matricies are put into a shared map
- A shared queue containing a list of all rows is created
- Each cluster of the node pops a value from the queue and calculates it until the queue is empty
- The results are written into a shared results map, which is fetched at the end of the execution

3.2.1 Network overhead

The proposed method has some flaws, the main one is that each cluster has fetch an entire selected row and column from the original array, as the solution doesn't cache the array after fetching it. This means that a lot more data is fetched and that the overall performance is decreased.

4 Experiments

4.1 Java

The following results have been archived by running the tasks:

Benchmark	(size)	Score	Error	Units
Task4Main.MBCMultiply	50	0.407	± 0.014	ms/op
Task4Main.MBCMultiply	100	0.467	± 0.010	ms/op
Task4Main.MBCMultiply	200	1.087	± 0.028	ms/op
Task4Main.MBCMultiply	300	2.688	± 0.104	ms/op
Task4Main.MBCMultiply	500	11.051	± 0.580	ms/op
Task4Main.MBCMultiply	1000	70.456	± 8.939	ms/op
Task4Main.MBCMultiply	1500	218.261	± 55.503	ms/op
Task4Main.MBCMultiply	2000	526.909		ms/op
Task4Main.MBCMultiply	2500	1047.527		ms/op
Task4Main.MBCMultiply	3000	2181.038		ms/op
Task4Main.MBCMultiply	4000	6668.943		ms/op
Task4Main.MBCMultiply	4500	10150.216		ms/op
Task4Main.MBCMultiply	5000	13136.560		ms/op
Task4Main.distributedMBCMultiply	50	18.121	± 6.635	ms/op
Task4Main.distributedMBCMultiply	100	12.509	± 1.602	ms/op
Task4Main.distributedMBCMultiply	200	14.723	± 4.002	ms/op
Task4Main.distributedMBCMultiply	300	13.993	± 1.944	ms/op
Task4Main.distributedMBCMultiply	500	28.535	± 6.760	ms/op
Task4Main.distributedMBCMultiply	1000	78.764	± 11.302	ms/op
Task4Main.distributedMBCMultiply	1500	215.640	± 76.539	ms/op
Task4Main.distributedMBCMultiply	2000	667.419		ms/op
Task4Main.distributedMBCMultiply	2500	1124.073		ms/op
Task4Main.distributedMBCMultiply	3000	1730.150		ms/op
Task4Main.distributedMBCMultiply	4000	3988.783		ms/op
Task4Main.distributedMBCMultiply	4500	5318.377		ms/op
Task4Main.distributedMBCMultiply:p1.00	5000	9697.231		ms/op

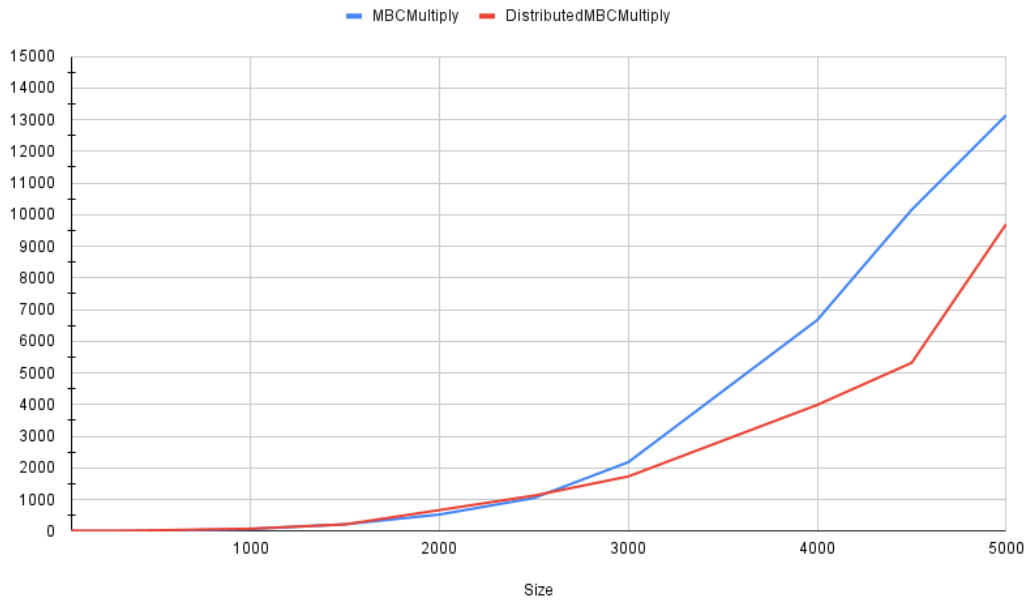

Figure 1: Execution times for the distributed and standard versions of the multiplication algorithm.

The experiments conclude, that the benefits of distributing the load start becoming visible only for arrays of size greater than 3000. For smaller arrays the network overhead is too large and the execution is slower. Adding an additional computation unit to the cluster seems to reduce the runtime by around 25%, which would be in line with the performance of M3 Pro Macbook and the significantly slower AMD Ryzen CPU.

It is worth noting that the performance improvements are visible only when we ignore the time that is necessary to transfer the input and output data of the calculations. For this paper we are

assuming, that the data is already present on all PCs when the execution starts and that the results can be fetched later.

4.2 Python

The following results have been archived by running the tasks. A custom measurment function was used:

Name (time in ms)	Min	Max	Mean
test_multiply_matrices[1]	0.000533	0.000542	0.000538
test_multiply_matrices[5]	0.025708	0.025717	0.025713
test_multiply_matrices[10]	0.203708	0.208483	0.206096
test_multiply_matrices[50]	26.023417	26.190758	26.107088
test_multiply_matrices[100]	212.143492	212.739783	212.441637
test_multiply_matrices[150]	711.709033	712.062725	711.885879
test_multiply_matrices[200]	1685.305342	1688.160000	1686.732671
test_multiply_matrices[300]	5893.233308	5946.454175	5919.843742

Name (time in ms)	Min	Max	Mean
test_distributed_multiply_matrices[1]	0.001598	0.001761	0.001679
test_distributed_multiply_matrices[5]	0.076770	0.084847	0.080809
test_distributed_multiply_matrices[10]	0.610062	0.671068	0.640565
test_distributed_multiply_matrices[50]	78.069249	85.876174	81.972712
test_distributed_multiply_matrices[100]	636.430946	700.074041	668.252494
test_distributed_multiply_matrices[150]	2135.128300	2348.641130	2241.884715
test_distributed_multiply_matrices[200]	5056.915785	5562.607364	5309.761575
test_distributed_multiply_matrices[300]	17679.700850	19447.671935	18563.686393

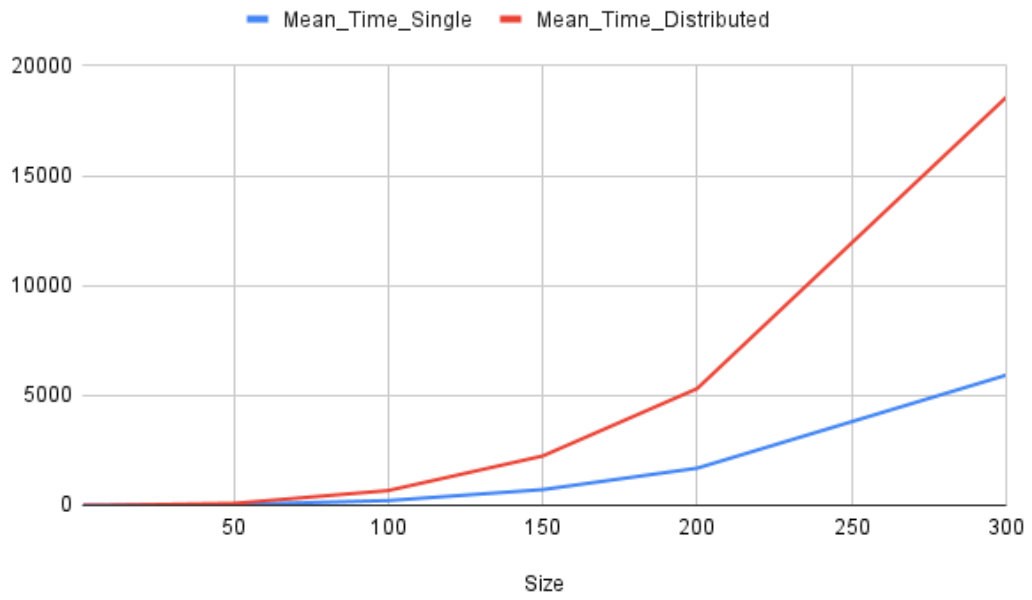

Figure 2: Execution times for the distributed and standard versions of the multiplication algorithm in Python.

As we can see from the results, the unoptimized distributed function performed worse than the base function.

5 Conclusion

This paper tested ways of using distributed computing for the problem of matrix multiplication. This method proves a lot less useful and more tricky than previously considered optimization methods. It is highly dependent on network conditions and may cause large overheads when implemented in a sub-optimal way (see the Python implementation). A in case of well optimized execution, which aims to minimize the network overhead the performance improvement is visible for large arrays.

In conclusion, distributed computing has the potential for completing very complex tasks with the use of multiple computers. The problem with matrix multiplication is that the A and B matricies have to be delivered to the nodes to do the calculations and then the results have to be sent back. For large arrays this may be from megabytes to gigabytes of data. This leads to a conclusion that matrix multiplication is a task which isn't the best candidate for solving it with distributed systems, except maybe for some extreme cases. In most cases delegating the multiplication task to the GPU will be a lot faster than dealing with network and serialization overheads.

6 Future work

A high speed, low latency method for sharing arrays between nodes has to be developed in order for distributed matrix multiplication to work. With such system the benefit of using multiple computing units for this task would have a huge potential.