Big Data Task 4 Report

Jakub Jazdzyk

December 27, 2024

Repository Link: GitHub Repository

1 Distributed Matrix Multiplication with Hazelcast in Java

1.1 Introduction

In this task, I implemented distributed matrix multiplication using Hazelcast in Java. Two computers were configured as nodes to form the cluster. The implementation involved splitting the matrices into slices and distributing the computations across the nodes. The IP addresses of the nodes were specified explicitly to enable proper communication between them.

```
Members {size:2, ver:2} [
    Member [192.168.1.44]:5701 - 04589fd7-151e-4551-a754-1da2910c888b
    Member [192.168.1.194]:5701 - 368ffbc0-9074-48b6-ad16-8bbc08c94061 this
]
```

Figure 1: Cluster configuration showing two connected nodes

1.2 Implementation Details

The nodes used the following IP addresses:

- Node 1: 192.168.1.44
- Node 2: 192.168.1.194

The Hazelcast configuration ensured efficient communication and task distribution.

1.3 Results

Time measurements for matrix multiplication of various sizes:

Matrix Size	Time (s)
100 x 100	0.68
200 x 200	0.60
1000 x 1000	13.00
2000 x 2000	63.41
4000 x 4000	256.82

Table 1: Execution times for different matrix sizes

1.4 CPU and Memory Usage

The CPU and Memory usage were recorded during the multiplication of matrices of sizes 1000, 2000, and 4000. The results are represented in the following plots:

- 1000 x 1000: `1000_java_plot.png` and `1000_java_plot_1.png`
- 2000 x 2000: `2000_java_plot.png` and `2000_java_plot_1.png`
- 4000 x 4000: `4000_java_plot.png` and `4000_java_plot_1.png`

1.5 Java Distributed Matrix Multiplication

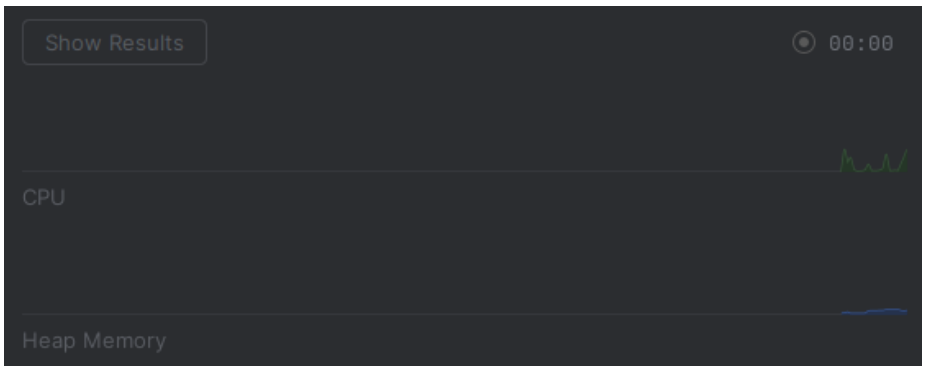

Figure 2: CPU and Memory Usage for 1000 x 1000 matrix multiplication (Java)

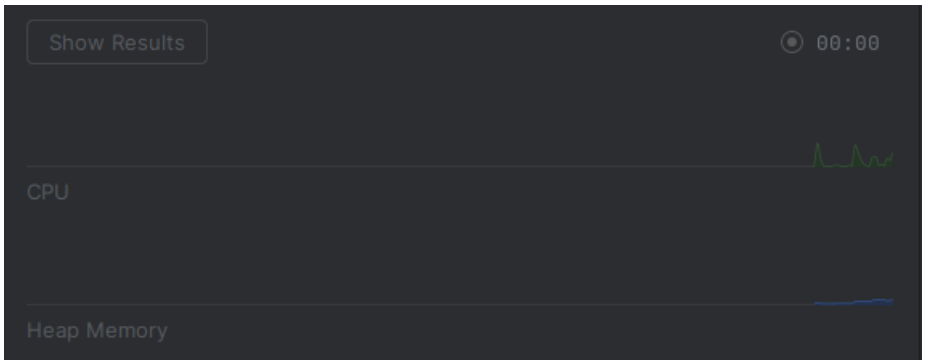

Figure 3: CPU and Memory Usage for 1000 x 1000 matrix multiplication on second node (Java)

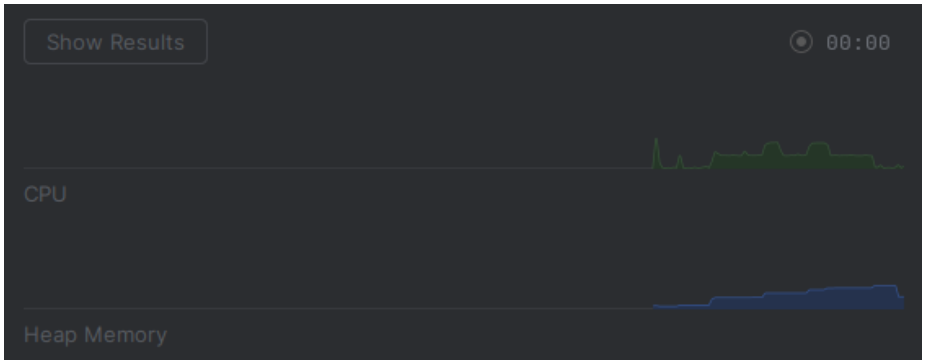

Figure 4: CPU and Memory Usage for 2000 x 2000 matrix multiplication (Java)

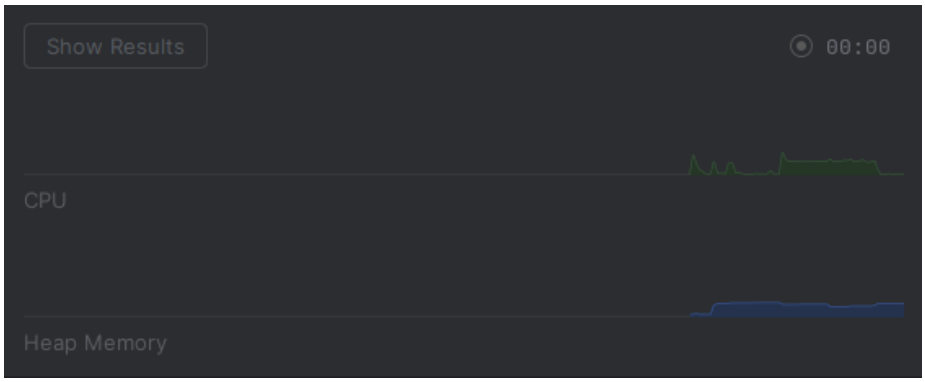

Figure 5: CPU and Memory Usage for 2000 x 2000 matrix multiplication on second node (Java)

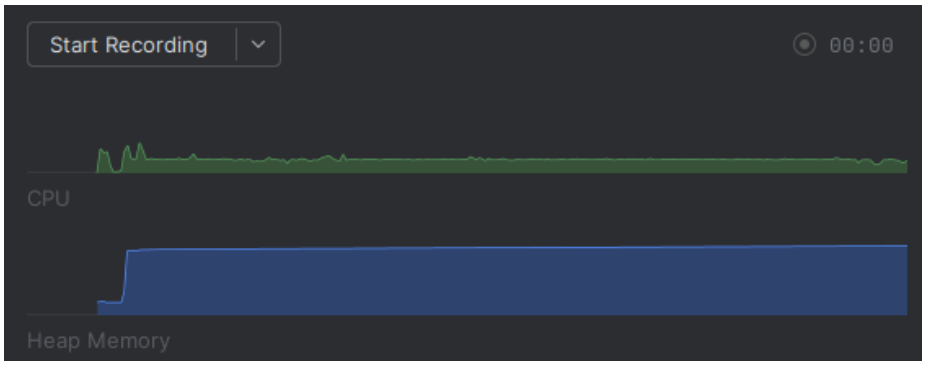

Figure 6: CPU and Memory Usage for 4000 x 4000 matrix multiplication (Java)

Figure 7: CPU and Memory Usage for 4000 x 4000 matrix multiplication on second node (Java)

1.6 Comparison and Observations

The execution times were compared with normal and parallel approaches. Observations include:

- Distributed computing significantly reduced computation time for larger matrices.
- Effective utilization of resources was achieved.

We can see that the results of the distributed and regular approaches are of similar order of magnitude when compared to the previous task. This is because we are dealing with a large network and data transfer overhead. Additionally, in my project I use only two nodes, because I do not have more computers available for testing. Better results could be observed by connecting to the cluster e.g. 10 computers, or even more. Also we see that there is bigger memory and CPU usage in the task counting bigger matrices. This is a good sign according to the implementation.

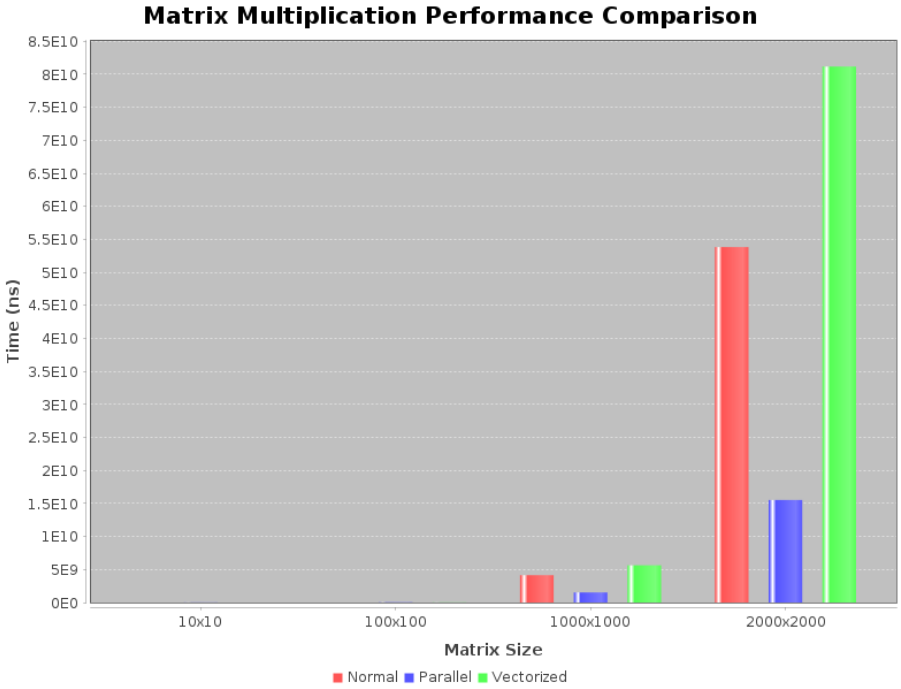

Figure 8: Comparison of time execution from previous task (normal and parallel), to compare

1.7 Conclusion

Distributed matrix multiplication using Hazelcast is a good way of dealing of large matrices multiplication.

2 Distributed Matrix Multiplication using MapReduce in Python

2.1 Introduction

This task involved implementing distributed matrix multiplication using the MapReduce framework in Python. Computations were performed on a single computer, simulating a distributed environment. I did many attempts of connecting computers using hazelcast-python-client, although I couldn't manage to make them work, so I decided to use different approach. I implemented map-reduce approach in python not using additional libraries, like MRJob.

2.2 Results

The time, memory usage, and CPU usage for various matrix sizes were recorded:

- 100 x 100
- 200 x 200
- 500 x 500
- 1000 x 1000

2.3 Python Distributed Matrix Multiplication, Time Comparison (s)

Matrix Size	Normal	Parallel	Distributed
100 x 100	0.98	0.37	0.30
200 x 200	9.56	1.00	1.09
500 x 500	141.23	21.30	16.43
1000 x 1000	1191.20	307.12	296.19

Table 2: Results for Python MapReduce approach

2.4 Python Distributed Matrix Multiplication, Memory Comparison (MB)

Matrix Size	Normal	Parallel	Distributed
100 x 100	49.34	53.78	53.25
200 x 200	50.17	64.19	62.35
500 x 500	56.73	142.01	137.77
1000 x 1000	79.50	437.65	428.53

Table 3: Results for Python MapReduce approach

2.5 Comparison and Observations

Results were compared with the normal approach. Observations include:

- MapReduce showed better scalability for larger matrices.
- Resource usage remained within acceptable limits.
- Results similar to parallel approach.

2.6 Conclusion

MapReduce offers a viable solution for distributed computations, even in simulated environments.

3 Frequent Itemset Mining in Python

3.1 Introduction

The frequent itemset mining problem involves identifying itemsets that appear frequently in a transactional dataset. The Python implementation utilized efficient algorithms to achieve this.

3.2 Implementation Details

The dataset `transactions_dataset.png` was used for testing. Frequent itemsets were mined at various support levels:

- Support 0.0: `transactions_frequent_itemset00.png`
- Support 0.4: `transactions_frequent_itemset04.png`
- Support 0.8: `transactions_frequent_itemset08.png`

3.3 Conclusion

The results highlight the effectiveness of frequent itemset mining. Map Reduce method is a good method for detecting dependencies, for example in retail stores.

3.4 Frequent Itemset Mining

```
transactions = [
    ['milk', 'bread', 'butter'],
    ['bread', 'butter', 'jam'],
    ['milk', 'bread', 'butter', 'jam'],
    ['bread', 'jam']
]
```

Figure 9: Transactional dataset used for testing

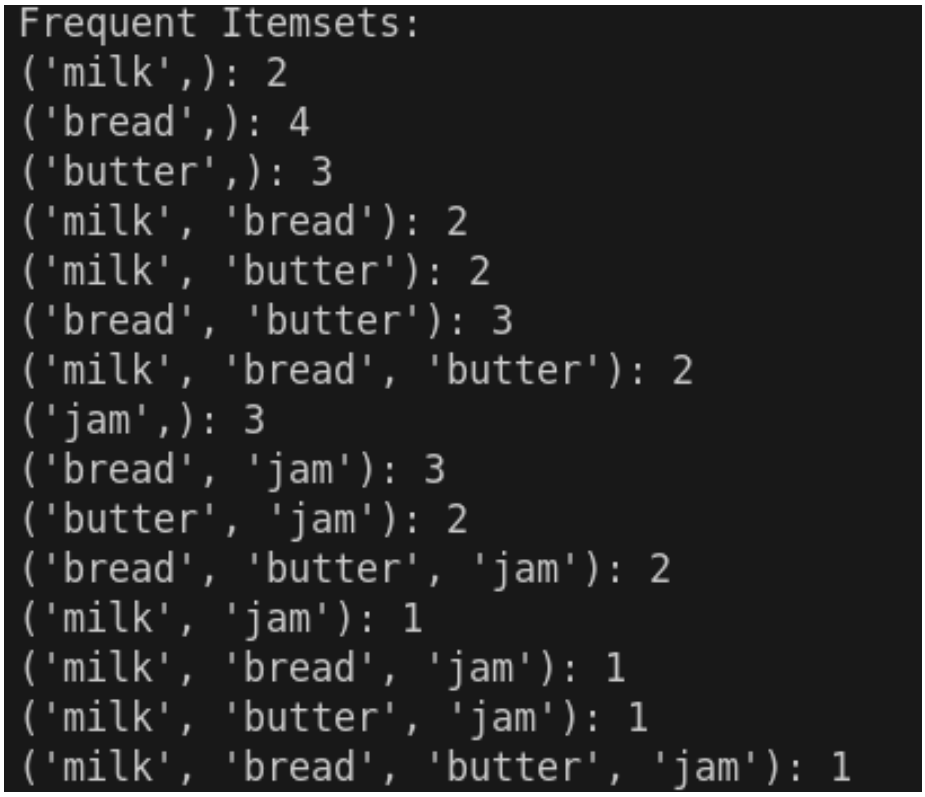

Figure 10: Frequent itemsets with support 0.0

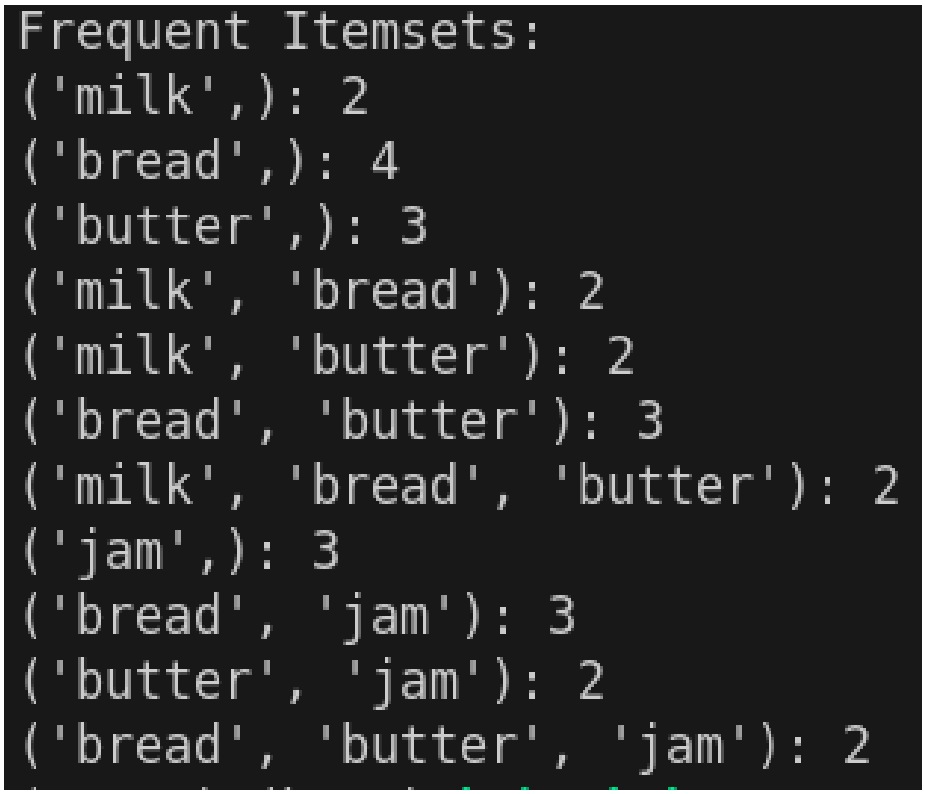

Figure 11: Frequent itemsets with support 0.4

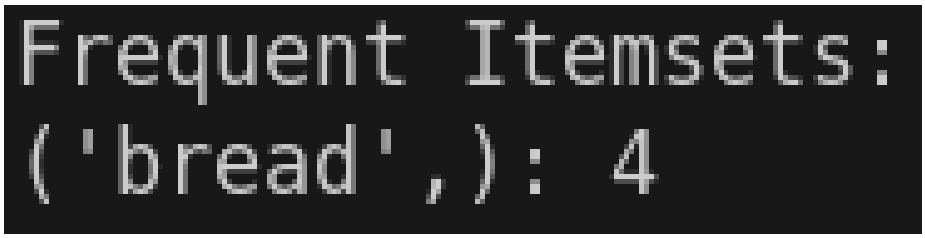

Figure 12: Frequent itemsets with support 0.8