Documentation
Language Benchmark of matrix multiplication 4

Olga Kalisiak 00325481

January 11, 2025

Github repository: link

Contents

1 Abstract

This project investigates the performance and feasibility of distributed matrix multiplication using Hazelcast, ZeroTier, and Python in a heterogeneous computing environment. Hazelcast, an in-memory data grid, is integrated with ZeroTier virtual networking to establish a distributed system. Additionally, Docker ensures consistency and portability across machines. Python's mpi4py is utilized for parallel matrix computations, achieving superior local performance compared to Java-based solutions. The experiments benchmark matrix sizes from 10x10 to 10,000x10,000 and highlight challenges such as communication overhead, memory limitations, and network issues caused by security software, including Windows Defender. Despite these challenges, the study provides insights into optimizing distributed computing systems and identifies areas for future enhancement, such as improving fault tolerance, reducing overhead, and refining security configurations for seamless deployment.

2 Introduction

Distributed computing has become a cornerstone of modern software architecture, enabling scalability and fault tolerance in various applications. This project explores the integration of Hazelcast, an in-memory data grid, with ZeroTier, a virtual networking solution, for distributed computation. Additionally, Docker is used to containerize the setup, ensuring portability and ease of deployment. The project also implements a Python-based solution for matrix multiplication, leveraging MPI for parallel processing. The goal was to evaluate the feasibility and performance of these systems in a distributed environment.

3 Problem statement

Efficiently managing and processing large datasets on multiple machines poses significant challenges, particularly to ensure low-latency communication and maintaining fault tolerance. Hazelcast offers a platform for distributed data processing, but its performance and reliability depend on the underlying network setup. ZeroTier was chosen to facilitate virtual networking and overcome the potential limitations of local network configurations. Despite these tools, unforeseen issues, such as conflicts with local security software (e.g., Windows Defender), can disrupt communication and impact overall performance.

4 Methodology

4.1 Hazelcast Configuration

- Hazelcast version 5.5.1 was configured for TCP/IP communication.
- A custom XML configuration file specified the member IPs for cluster formation.

4.2 ZeroTier Networking

- ZeroTier was used to establish a virtual network between two machines with the assigned IPs: 192.168.195.142 and 192.168.195.22
- The network was tested for connectivity prior to deploying Hazelcast.

4.3 Docker Integration

- Hazelcast and the application logic were containerized using Docker.
- The images were built to ensure consistency between machines.

4.4 Python implementation

- A distributed matrix multiplication task was implemented using Python and mpi4py for parallel processing.
- The Python solution demonstrated faster execution than the Java implementation, with the largest matrices tested being 10,000x10,000.
- Despite running locally successfully, connectivity issues caused by security software (e.g. Windows Defender) prevented full deployment on multiple machines.

4.5 Experiment Setup

- A distributed matrix multiplication task was implemented to benchmark the system.
- Matrices of sizes ranging from 10x10 to 10,000x10,000 were processed using both Hazelcast and Python.

4.6 Machines specifications

I conducted tests on machines with the following specifications:

4.6.1 Machine 1

- Laptop model: Acer Nitro 5
- Processor: AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz
- Installed RAM: 16.0 GB (available: 15.4 GB)
- Product ID: 00328-00801-21601-AA448
- System Type: 64-bit operating system, x64 processor
- Operating system: Windows 11

4.6.2 Machine 2

- Laptop model: Lenovo Legion Y540
- Processor: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
- Installed RAM: 32.0 GB
- Product ID: 00325-81552-58621-AAOEM
- System Type: 64-bit operating system, x64 processor
- Operating system: Windows 11

4.7 Languages and compilers

- Java (version 23.0.1, IntelliJ IDEA Community Edition 2024.2.2)
- Python (version 3.10.11, PyCharm 2024.2.3)

5 Experiments

The codes of these experiments are available in the repository: Github repository

The initial experiment faced significant issues due to Windows Defender blocking essential communication ports required by Hazelcast. Despite configuring the firewall and adding exceptions, a stable connection between the two machines could not be established. Consequently, the distributed processing functionality was not fully realized.

5.1 Network Connectivity Validation

During the experimentation, it was crucial to validate the connectivity between the two machines on the ZeroTier virtual network. A simple test using PowerShell's Test-NetConnection command confirmed that the machines were able to establish a TCP connection to each other, indicating that the underlying network setup was functioning properly.

The following output from the test shows that the machine with IP address 192.168.195.142 successfully communicated with another machine at IP 192.168.195.22 on port 8677, which is the port used by Hazelcast for internal communication:

```
PS C:\Users\Olga\Desktop\studia\distributed_matrix_multiplication> Test-NetConnection
-ComputerName 192.168.195.22 -Port 8677
>>

ComputerName     : 192.168.195.22
RemoteAddress    : 192.168.195.22
RemotePort       : 8677
InterfaceAlias   : ZeroTier One [159924d630271ca0]
SourceAddress    : 192.168.195.142
TcpTestSucceeded : True
```

This result shows that the virtual network created by ZeroTier is functioning as expected and that the machines are capable of connecting to each other. Despite this, even after disabling Windows Defender to rule out firewall-related issues, the communication between the machines still did not work as expected. The security software seemed to continue interfering with the network traffic, preventing Hazelcast from fully utilizing the connection for distributed computation.

This highlights that while the network layer was correctly configured and the firewall was disabled, external security settings might still affect the communication, blocking critical ports and impeding the distributed processing setup.

5.2 Performance Observations

5.2.1 Communication Overheads

- Using Hazelcast introduced noticeable overhead due to serialization and network communication, resulting in slower performance compared to local computation.
- The overhead was particularly evident for smaller tasks, where the communication cost outweighed the benefits of distribution.

5.2.2 Memory Limitations

- For matrices of 500x500 size, the system encountered heap memory issues. This occurred because of Hazelcast's need to replicate and manage data across nodes, leading to higher memory consumption.
- Without proper memory tuning, such as increasing the Java heap size, the system could not handle larger workloads.

5.2.3 Python Performance

- The Python-based approach using mpi4py executed faster than the Java-based Hazelcast solution.
- The system was able to process matrices up to 10,000x10,000, with performance metrics recorded for memory and CPU usage.
- However, like with Hazelcast, distributed execution across machines was hindered by security software conflicts.

5.3 Results

5.3.1 Distributed matrix multiplication in Java

Matrix Size	Time [ms]	Memory [MB]
10	128	19
50	200	18
100	476	48
150	799	112
200	1108	344
250	1350	347
300	2155	653
350	2515	732
400	4865	1242

Table 1: Performance Metrics for Matrix Multiplication

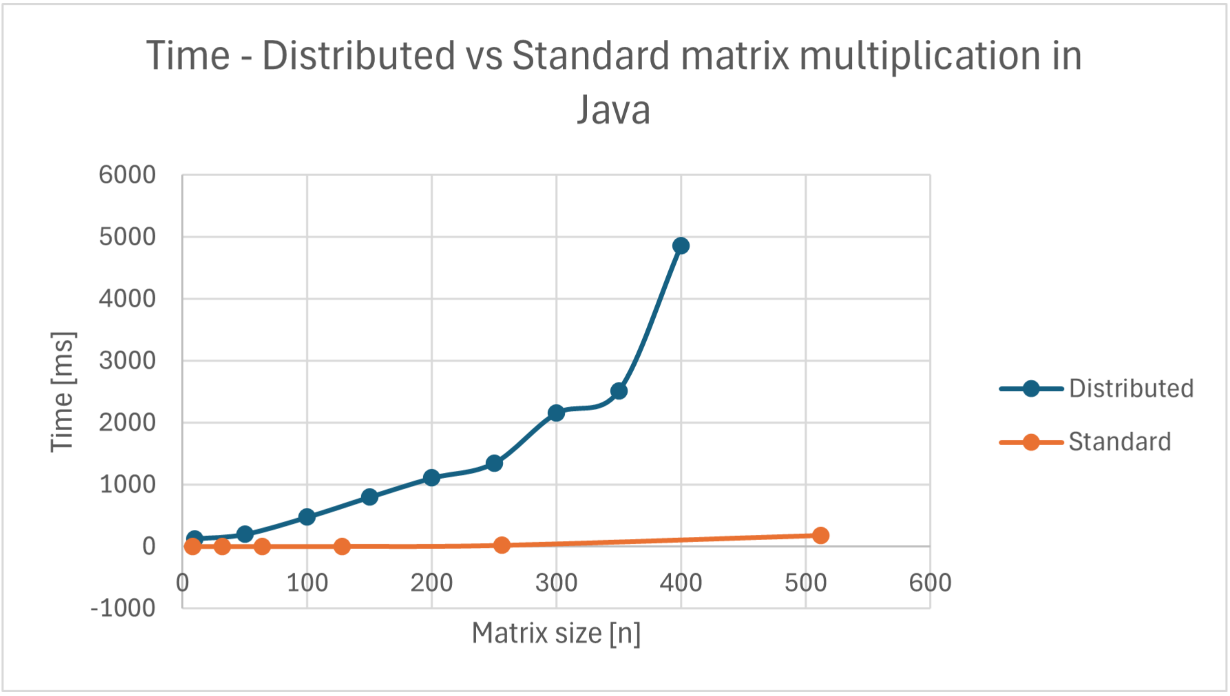

Figure 1: Performance time for distributed vs standard matrix multiplication in Java.

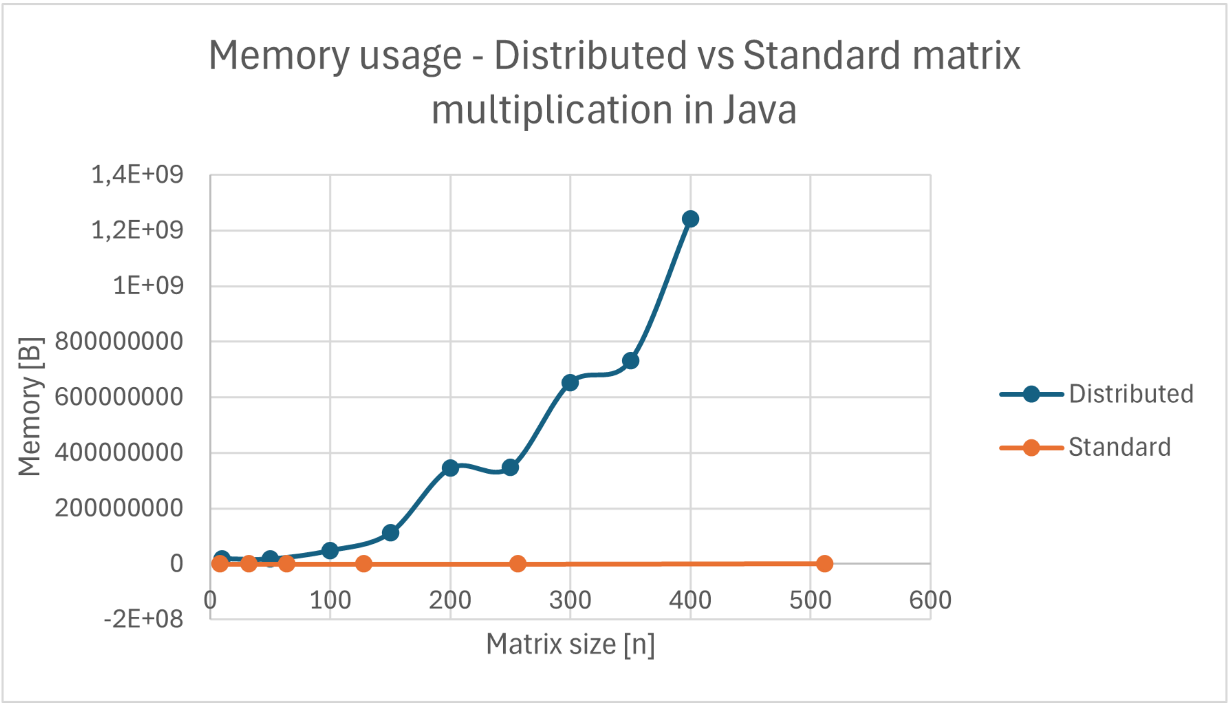

Figure 2: Memory usage for distributed vs standard matrix multiplication in Java.

5.3.2 Distributed matrix multiplication in Python

Matrix Size	Time [ms]	Memory [MB]	CPU Usage
10	0.99	32.99	0%
50	1.00	33.10	0%
100	1.00	33.65	11%
150	2.00	33.72	8.30%
200	3.00	34.37	8.30%
250	3.00	34.96	4.50%
300	7.00	35.88	3.30%
350	10.00	36.87	7.10%
400	8.00	36.81	2.40%
1000	40.11	56.95	4.20%
3000	671.09	249.01	40.80%
6000	3497.08	896.99	60.50%
10000	17479.81	1604.38	59.70%

Table 2: Performance Metrics for Matrix Multiplication in Python

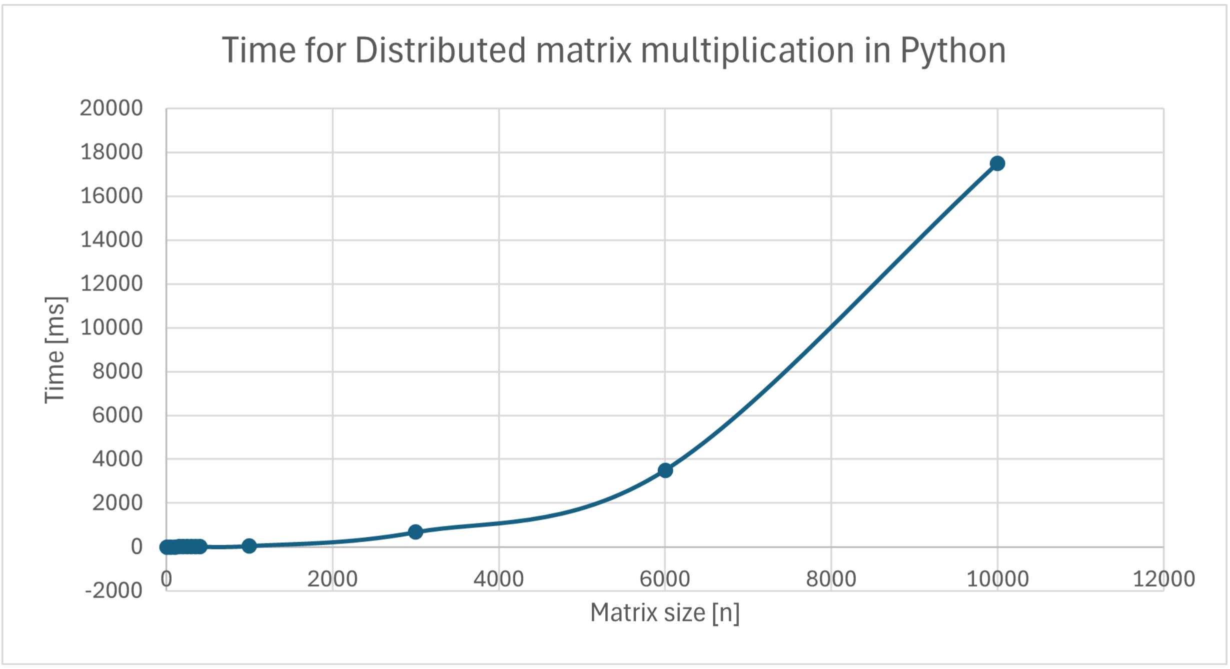

Figure 3: Performance time for distributed matrix multiplication in Python.

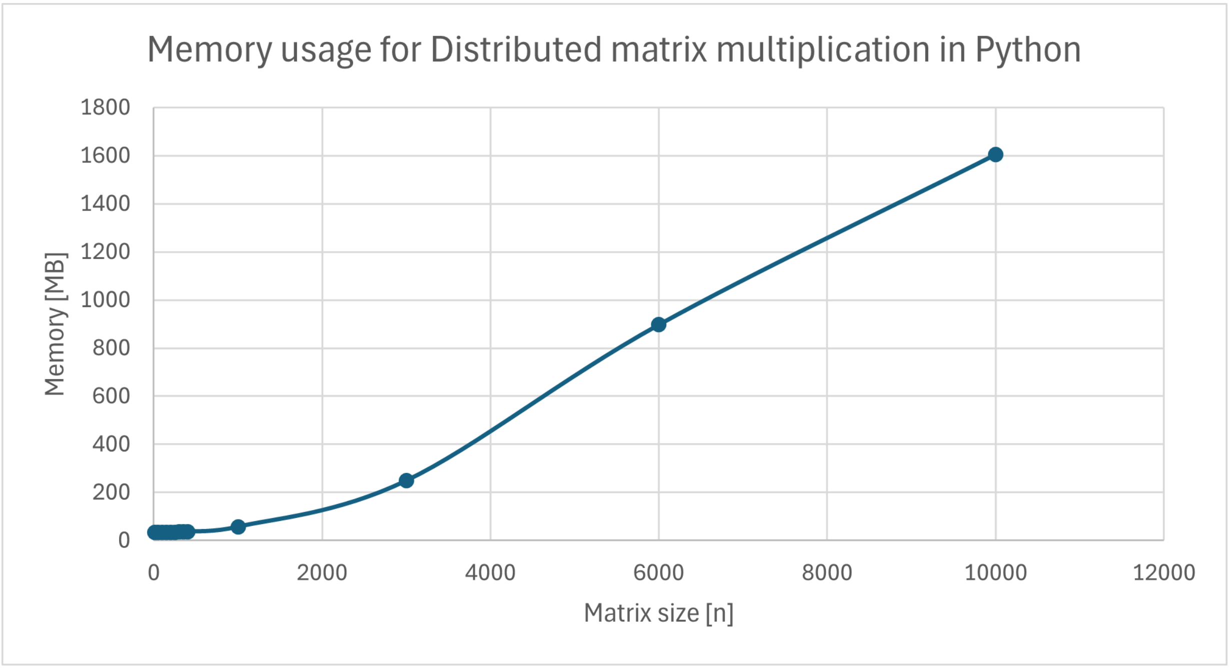

Figure 4: Memory usage for distributed matrix multiplication in Python.

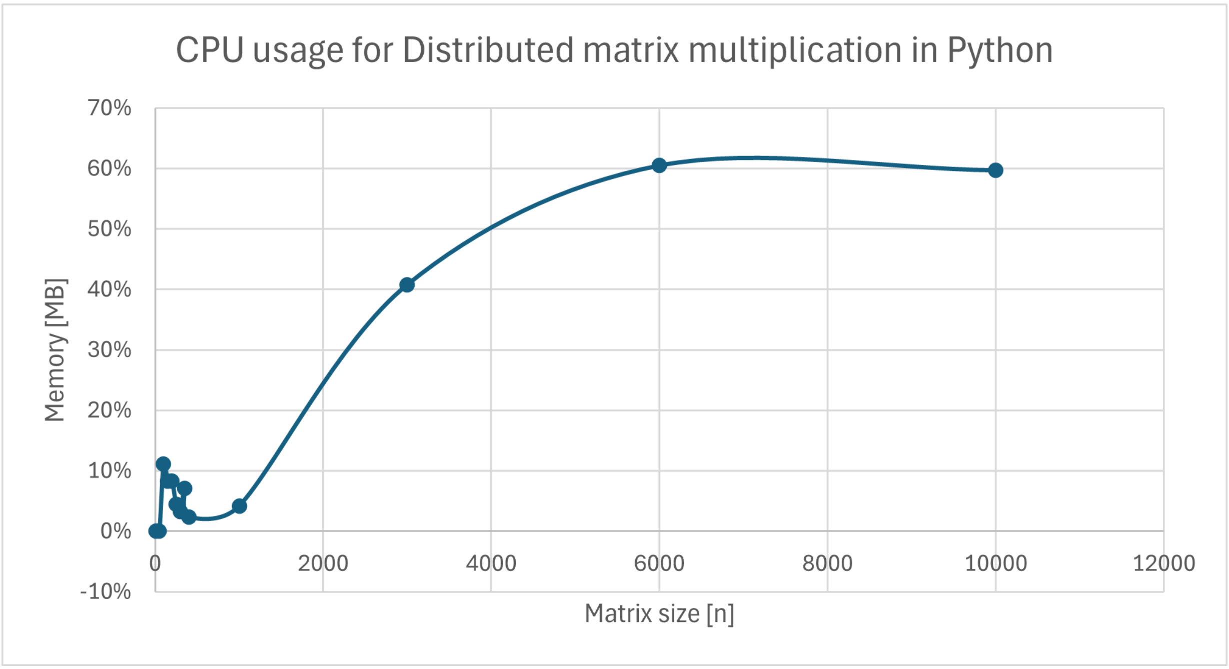

Figure 5: CPU usage for distributed matrix multiplication in Python.

5.3.3 Comparison of Distributed vs Standard matrix multiplication in Python

5.3.4 Distributed matrix multiplication in Python

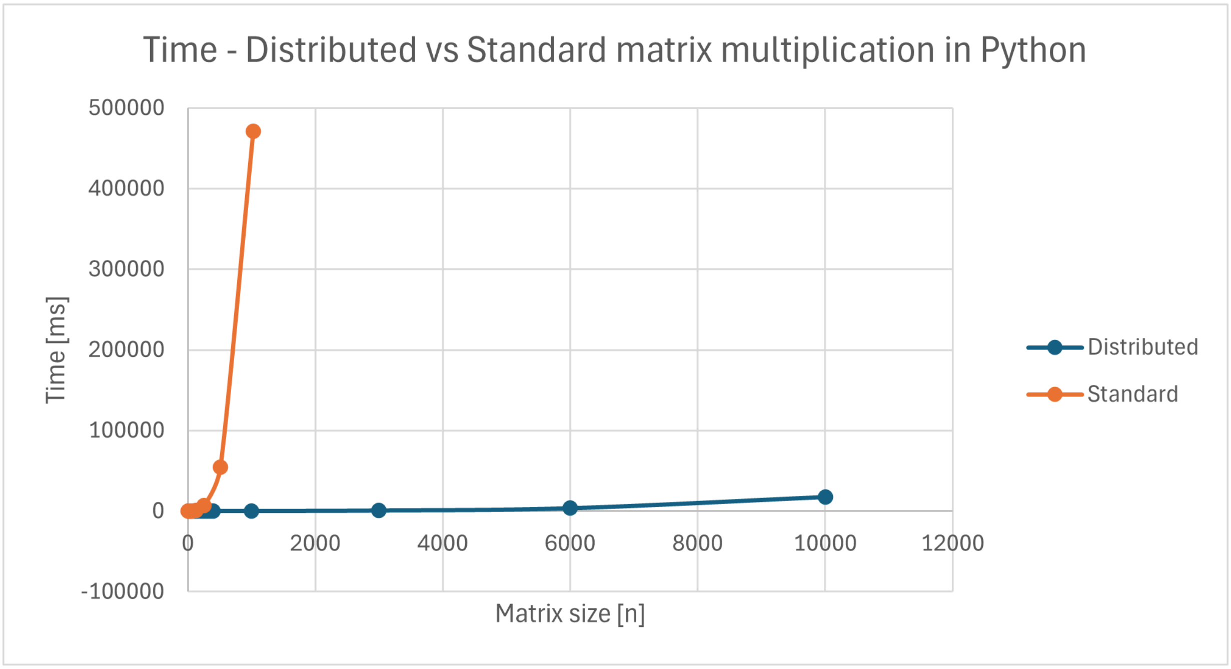

Figure 6: Performance time for distributed and standard matrix multiplication in Python.

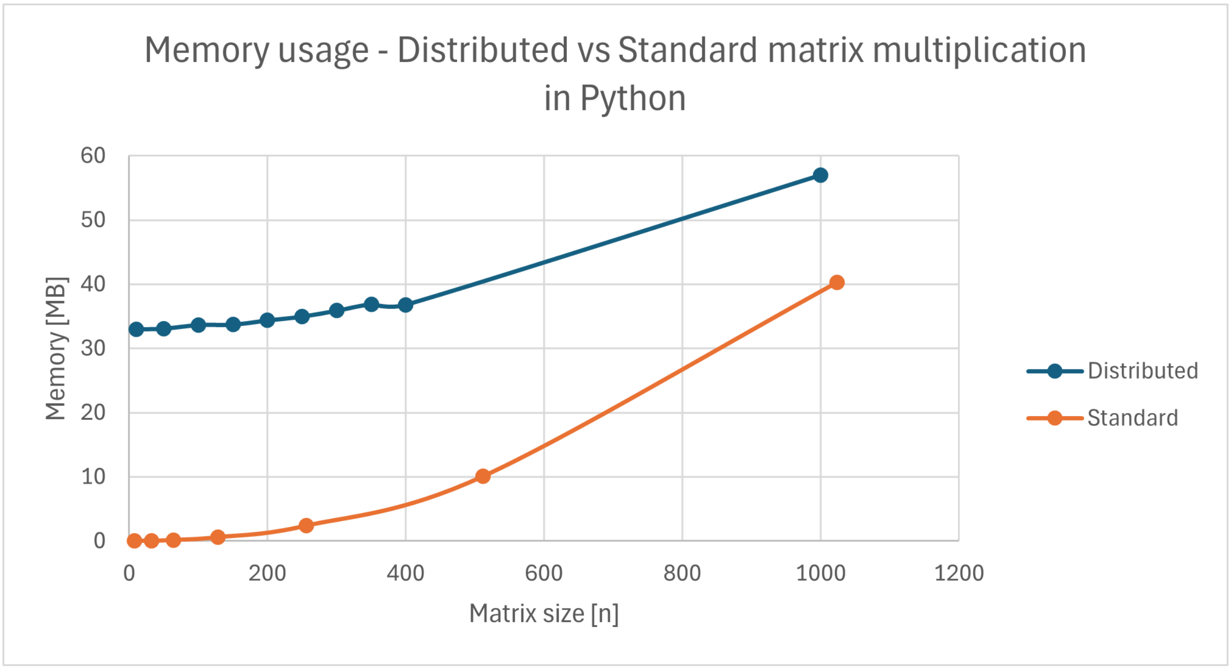

Figure 7: Memory usage for distributed and standard matrix multiplication in Python.

Although the system demonstrated the potential for distributed computation, several factors limited its effectiveness.

- Firewall and security software conflicts prevented seamless communication.
- Hazelcast's overhead reduced performance for tasks with relatively low computational complexity.
- Insufficient memory allocation hindered scalability for larger datasets.

6 Conclusion

This project highlighted the challenges of implementing a distributed computing system using Hazelcast, ZeroTier, and Python. While the tools offer robust features for distributed data processing and virtual networking, their integration requires careful consideration of network security settings and resource allocation.

7 Future work

Future work could explore optimizing memory management, refining network configurations, and addressing the overhead associated with distributed communication to improve system performance and reliability.