Distributed Matrix Multiplication using Hazelcast

Jose Luis Perdomo

January 12, 2025

Abstract

This work explores distributed computing approaches to matrix multiplication, focusing on scalability and the handling of very large matrices that cannot fit into the memory of a single machine. The task was implemented using Hazelcast, a distributed computing framework, and was tested using both Python and Java. The experiments were conducted on a single machine due to the lack of access to a cluster, with all matrix multiplications performed locally.

1 Introduction

Matrix multiplication is a fundamental operation in many fields such as computer science, engineering, and data analysis. With the increasing size of datasets, matrix multiplication for very large matrices becomes challenging due to memory limitations. Traditional approaches may not be sufficient to handle matrices that do not fit into the memory of a single machine.

In this work, we explore the use of distributed computing frameworks, specifically Hazelcast, to implement matrix multiplication in a distributed manner. We implemented the solution in both Python and Java, testing its scalability and performance as the size of the matrices increases. The work aims to evaluate how distributed systems handle large matrices, considering factors such as network overhead, resource utilization, and memory efficiency.

The experiments were performed using a single machine as access to a cluster was not possible. Despite this limitation, we managed to explore the feasibility of matrix multiplication in a distributed environment with the available hardware resources.

2 Experiments

2.1 Python Experiment

The Python implementation uses the Hazelcast Python client to perform distributed matrix multiplication. We divided the matrix into smaller blocks and performed the matrix multiplication on these blocks concurrently. The experiment aimed to evaluate the scalability of the solution as the matrix size increases.

The Python code, which uses the ‘concurrent.futures.ThreadPoolExecutor‘ to manage parallel tasks, was designed to test the ability of Hazelcast to handle matrix blocks in parallel.

2.2 Distributed Matrix Multiplication with Hazelcast in Python

In this subsection, we implement a distributed matrix multiplication solution using Hazelcast in Python. The goal is to divide the matrix multiplication task into smaller blocks, distribute them across the cluster, and then combine the results efficiently.

2.2.1 Connecting to Hazelcast Cluster

The `connect_hazelcast` function initializes the Hazelcast client and connects to a Hazelcast cluster. The cluster’s IP address and name are specified in the configuration, along with other settings such as connection timeout.

```
def connect_hazelcast():
    client = HazelcastClient(
        cluster_members=["192.168.1.22"],
        cluster_name="dev",
        connection_timeout=5000,
        smart_routing=True,
        use_public_ip=True
    )
    print("Connected to the Hazelcast cluster.")
    return client
```

2.2.2 Matrix Generation and Block Extraction

The `generate_matrix` function generates random matrices using NumPy, and the `extract_block` function extracts a sub-matrix (or block) of a given size from the input matrix. These blocks are the units that will be distributed for multiplication.

```
def generate_matrix(rows, cols):
    return np.random.randint(0, 100, size=(rows, cols))

def extract_block(matrix, start_row, start_col, size):
    return matrix[start_row:start_row + size, start_col:start_col + size]
```

2.2.3 Matrix Block Multiplication

The `multiply_blocks` function performs matrix multiplication on two blocks using NumPy's `dot` function. This function is used to multiply sub-matrices in parallel.

```
def multiply_blocks(block_a, block_b):
    return np.dot(block_a, block_b)
```

2.2.4 Assembling the Final Matrix

Once the blocks are multiplied, the `assemble_matrix` function reconstructs the final result matrix by retrieving the results from the Hazelcast map and placing the blocks in the correct positions.

```
def assemble_matrix(result_map, matrix_size, block_size):
    result_matrix = np.zeros((matrix_size, matrix_size), dtype=int)
    blocks = matrix_size // block_size

    for i in range(blocks):
        for j in range(blocks):
            block = np.array(result_map.get(f"{i}_{j}").result())
            for k in range(block_size):
                for l in range(block_size):
                    row = i * block_size + k
                    col = j * block_size + l
                    result_matrix[row][col] = block[k][l]

    return result_matrix
```

2.2.5 Distributing Matrix Blocks

The distribute_blocks function is responsible for distributing the matrix multiplication tasks across the Hazelcast cluster. Each block of matrix A and matrix B is extracted and multiplied in parallel using Python's ThreadPoolExecutor. The results are stored in the Hazelcast map (resultMap) for later assembly.

```
def distribute_blocks(client, matrix_a, matrix_b, block_size, sub_block_size):
    executor = client.get_executor("blockExecutor")
    result_map = client.get_map("resultMap")
    result_map.clear()

    matrix_size = matrix_a.shape[0]
    blocks = matrix_size // block_size

    with ThreadPoolExecutor() as pool:
        futures = []

        for i in range(blocks):
            for j in range(blocks):
                block_a = extract_block(matrix_a, i * block_size, j * block_size
                block_b = extract_block(matrix_b, i * block_size, j * block_size

                future = pool.submit(multiply_blocks, block_a, block_b)
                result_map.put(f"{i}_{j}", future.result())

    result_matrix = assemble_matrix(result_map, matrix_size, block_size)
    return result_matrix
```

2.2.6 Main Function

The main function connects to the Hazelcast cluster, generates two random matrices, and then initiates the distributed multiplication using distribute_blocks. The execution time is measured and the first 10 elements of the result matrix are printed for verification.

```
def main():
    client = connect_hazelcast()

    if not client.lifecycle.is_running():
        print("Could not connect to the Hazelcast cluster. Check the configurati
        return

    matrix_size = 50
    block_size = 10
    sub_block_size = 5

    matrix_a = generate_matrix(matrix_size, matrix_size)
    matrix_b = generate_matrix(matrix_size, matrix_size)

    start_time = time.time()
    result = distribute_blocks(client, matrix_a, matrix_b, block_size, sub_block
    end_time = time.time()

    print("Matrix multiplication completed!")
    print("Result (first 10 values):", result.flatten()[:10])
    print(f"Total time: {end_time - start_time:.2f} seconds")

    client.shutdown()

main()
```

2.3 Java Experiment

The Java implementation of the distributed matrix multiplication used Hazelcast's Java client to distribute tasks across nodes. However, since the experiment was conducted on a single machine, the number of available nodes was limited. The matrix multiplication was performed using a block-based approach, where each block of the matrix was processed in parallel using 'Callable' tasks.

The Java implementation utilized the Strassen algorithm for matrix multiplication, which is more efficient than the naive approach for larger matrices. We tested matrix sizes from 10000x10000 to 25000x25000. However, due to memory limitations, the experiment encountered an 'OutOfMemoryError' for larger matrix sizes.

Java Code Repository: The full code of these experiments can be found in the following repository: `https://github.com/JoseLuisPerdomo/BD_Stage4`.

2.4 Distributed Matrix Multiplication Implementation with Hazelcast

In this experiment, a distributed solution for matrix multiplication is implemented using Hazelcast. The algorithm is designed to divide the task of multiplying two large matrices into smaller blocks, which are then distributed and processed in parallel, maximizing efficiency in multi-node environments.

2.4.1 Initial Hazelcast Configuration

The Hazelcast instance is configured by setting up a custom serialization for matrix block multiplication tasks (represented by `MatrixBlockTask`), using a specialized serializer (`MatrixBlockTaskSerializer`). Additionally, the network is configured to enable TCP connections between cluster nodes, specifying the addresses of the cluster members.

```
HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance(configuration);
```

2.4.2 Matrix Generation and Block Division

Two large random matrices (30,000 x 30,000) are generated, and then divided into smaller blocks of size 1500 x 1500. Each block of the matrices will be processed independently for the multiplication.

```
int[][] matrixA = MatrixGenerator.createMatrix(size, size);
int[][] matrixB = MatrixGenerator.createMatrix(size, size);
```

2.4.3 Executing Concurrent Tasks

The `executeTasks` method is responsible for creating and executing the matrix block multiplication tasks in parallel. For each block in matrices A and B, a `MatrixBlockTask` is created and executed using Hazelcast's distributed execution service (`ExecutorService`).

```
ExecutorService executor = hazelcast.getExecutorService("taskExecutor");
for (int x = 0; x < totalBlocks; x++) {
    for (int y = 0; y < totalBlocks; y++) {
        Callable<int[][]> task = new MatrixBlockTask(blockA, blockB, chunkSize,
        Future<int[][]> future = executor.submit(task);
        partialResults.put(x + ":" + y, future.get());
        latch.countDown();
    }
}
```

2.4.4 Collecting Results

Once all the tasks are completed, the `buildFinalMatrix` method reconstructs the final result matrix by combining the processed blocks. The partial results are stored in a distributed map (`IMap`) in Hazelcast, allowing concurrent access from different nodes.

```
int[][] finalMatrix = new int[size][size];
for (int x = 0; x < totalBlocks; x++) {
    for (int y = 0; y < totalBlocks; y++) {
        int[][] block = partialResults.get(x + ":" + y);
        // Copy blocks into the final matrix
    }
}
```

2.4.5 Matrix Multiplication Algorithm: Naive and Strassen

The processing of each matrix block is performed using two multiplication algorithms:

- **Naive**: A traditional multiplication algorithm based on three nested loops.
- **Strassen**: A more efficient algorithm that divides the matrix into submatrices and reduces the number of multiplications required.

The selected algorithm is defined when creating the `MatrixBlockTask` object, and the choice is made using the `algorithm` parameter.

```
if ("Strassen".equalsIgnoreCase(algorithm)) {
    return calculateUsingStrassen(blockA, blockB);
} else {
    return calculateUsingNaive(blockA, blockB);
}
```

2.4.6 Task Serialization

The serialization and deserialization of matrix multiplication tasks are crucial for their distribution across the Hazelcast cluster. The `MatrixBlockTaskSerializer` class is responsible for converting the tasks into a format that can be transmitted over the network and then reconstructed on the receiving nodes.

```
@Override
public void write(ObjectDataOutput out, MatrixBlockTask task) throws IOException
    // Serialize matrix blocks and chunk size
}

@Override
public MatrixBlockTask read(ObjectDataInput in) throws IOException {
    // Deserialize matrix blocks and chunk size
    return new MatrixBlockTask(blockA, blockB, chunkSize, "Strassen");
}
```

2.5 Experiment Results

The experiments showed the following results:

- **Experiment 1:** Matrix size = 10000, Block size = 500, Chunk size = 100.
  Time: 22904 ms
- **Experiment 2:** Matrix size = 20000, Block size = 1000, Chunk size = 200.
  Time: 157350 ms
- **Experiment 3:** Matrix size = 25000, Block size = 1200, Chunk size = 250.
  Result: The experiment resulted in an `OutOfMemoryError` due to the limitations of hazelcast heap memory usage.

Due to the lack of access to a cluster, all tests were conducted on a single machine. The heap size used by Hazelcast was limited to 7.9 GB, despite the machine having 28 GB available, preventing larger matrices from being processed.

3 Conclusion

In this work, we explored distributed matrix multiplication using Hazelcast, implemented in both Python and Java. We observed that Hazelcast provides an efficient framework for handling matrix multiplication on a distributed system, although the experiments were limited by memory constraints, as they were conducted on a single machine.

The results show that the distributed approach works well for moderate-sized matrices, but larger matrices exceeding the available heap size led to memory errors. Despite these limitations, the experiment demonstrated the potential of distributed computing to handle large-scale matrix operations.

For future work, access to a distributed cluster would allow for testing with much larger matrices, enabling a better understanding of Hazelcast's scalability and performance in a true distributed environment.