Università di Messina

Dipartimento di scienze matematiche e informatiche,
scienze fisiche e della terra

Corso di Laurea Triennale in Informatica

Distributed computation of linear algebra operations

Author:
Gabriele Aloisio *(503264)*

Supervisor:
prof. Roberto Marino

19 settembre 2023

Indice

1 Abstract

2 Introduction

Matrix operations are fundamental in numerous scientific and computational domains, serving as the building blocks for various applications. However, traditional sequential computation on a single machine often becomes a bottleneck, limiting the speed and scalability of matrix calculations. To address this problem, we are going to use an open-source distributed computing framework called **Ray**.

2.1 Overview of distributed systems and networks

A distributed system consists of multiple interconnected computers that collaborate and coordinate their activities to achieve a common goal. These systems are designed to tackle tasks that cannot be efficiently computed by a single machine. Networks serve as the backbone of distributed systems, enabling communication among the connected nodes. Network infrastructures enable coordination, data sharing, and synchronization across distributed systems, regardless of their physical locations.

2.2 The Ray library

The Ray library is an open-source distributed computing framework primarily designed for building scalable and high-performance applications. It was developed by the company Anyscale, which aimed to simplify the development of distributed and parallel computing applications. When it comes to matrix operation computation, using Ray for distributed computation offers several advantages over performing the operations on a single machine:

- **Faster execution**: With Ray you can use multiple machines and CPUs to compute operations. This significantly reduces the computation time compared to a single machine. Each machine can work on a subset of the matrix data, completing the calculations in parallel. This distributed approach can lead to substantial speedups.
- **Scalability**: As the size of the matrices grows, a single machine may struggle to handle the computational demands due to memory limitations or processing power constraints. Ray allows you to scale horizontally by adding more machines to the distributed setup.
- **Fault tolerance**: Ray offers fault tolerance mechanisms that ensure the continuity of computation even in the presence of failures. If a machine participating in the distributed computation fails, Ray can automatically redistribute the workload to other available machines.
- **Resource utilization**: With Ray, each machine contributes its processing power and memory capacity to the overall computation. This efficient utilization of resources allows you to make the most of the available hardware infrastructure, compared to a single machine that may be underutilized.

3 Case study

4 Implementation

4.1 Python

Python is a versatile and high-level programming language known for its simplicity and readability. It is widely used for a variety of applications, including web development, data analysis, artificial intelligence, scientific computing, and automation. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming, and its large and active community contributes to a vast ecosystem of libraries and frameworks, making it a powerful tool for a wide range of tasks. It is one of the two languages supported by Ray, alongside Java.

4.2 Ray

To parallelize a function with Ray, import Ray and and initialize it with *ray.init()*. Then decorate the function with *@ray.remote* to declare that you want to run this function remotely. Lastly, call the function with *.remote()* instead of calling it normally. This remote call yields a future, a Ray object reference, that you can then fetch with *ray.get*:

```
import ray
ray.init()

@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]

```

4.3 Ray clusters

4.4 The Matrix class

Before implementing the parallelized algorithms for Ray, we write the code for the serial execution in the **Matrix** class (most of the functions defined in the class are auxiliary therefore were excluded for practical purposes):

```
class Matrix:
    def __init__(self, data):
        if isinstance(data, list):
            self.data = [[round(val, 8) for val in row]
                                        for row in data]
        else:
            raise ValueError("Input must be a list")

    [...]

    def dot(A, B): [...]
    def det(self): [...]
    def rank(self): [...]
    def inv(self): [...]
```

Which we modify later to achieve parallelization with Ray.

4.4.1 Dot product

Matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix **product**, has the number of rows of the first and the number of columns of the second matrix.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{bmatrix}$$

The matrix multiplication of A and B is denoted as $C = A \cdot B$. The resulting matrix C will have dimensions $m \times p$.

The entry c_{ij} of the resulting matrix C is computed as follows:

$$c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \cdots + a_{in} \cdot b_{nj} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The resulting matrix C can be expressed as:

$$C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mp} \end{bmatrix}$$

The definition above can be described as the following code:

```
1  @staticmethod
2  def dot(A, B):
3      if A.is_matrix() and B.is_matrix():
4          a_rows, a_cols = A.shape()
5          b_rows, b_cols = B.shape()
6
7          if a_cols == b_rows:
8              result = [[0] * b_cols for _ in range(A.shape()[0])]
9              for i in range(A.shape()[0]):
10                 for j in range(b_cols):
11                     for k in range(a_cols):
12                         result[i][j] += A.data[i][k] * B.data[k][j]
13             return Matrix(result)
14         else:
15             raise ValueError(
16                 "Matrix dimensions do not match for dot product")
17     else:
18         raise ValueError("Dot product requires a Matrix object")
```

Listing 1: Dot function

To parallelize this function, we simply delegate the calculation made in line 12 to an auxiliary function *dot_calc* defined in the *tasks.py* file:

```python
@ray.remote
def dot_calc(A, B, i, j, k):
    return A.data[i][k] * B.data[k][j]
```

Listing 2: dot_calc

```python
@staticmethod
def dot(A, B):
    if A.is_matrix() and B.is_matrix():
        a_rows, a_cols = A.shape()
        b_rows, b_cols = B.shape()

        if a_cols == b_rows:
            result = [[0] * b_cols for _ in range(a_rows)]
            futures = []

            for i in range(a_rows):
                for j in range(b_cols):
                    for k in range(a_cols):
                        # result[i][j] += A.data[i][k] * B.data[k][j]
                        futures.append(((i, j), t.dot_calc.remote(A, B, i, j,
    k)))

            for future in futures:
                i, j = future[0]
                result[i][j] += ray.get(future[1])

            return Matrix(result)
        else:
            raise ValueError(
                "Matrix dimensions do not match for dot product")
    else:
        raise ValueError("Dot product requires a Matrix object")
```

Listing 3: Parallelized dot function

4.4.2 Determinant

4.4.3 Rank

The **rank** of a matrix A, denoted as rank(A), is defined as the maximum number of linearly independent rows or columns in the matrix. We chose to determine the rank of a matrix by using the minor criterion, also known as the **criterion of minors**. According to this criterion, the rank of a matrix is equal to the largest order of a non-zero determinant of any square submatrix within the given matrix.

Let A be a matrix of size $m \times n$, and let j be the order of the largest non-zero determinant among all square submatrices of A. Then the rank of A, denoted as rank(A), is equal to j.

- To apply the minor criterion, we calculate the determinants of all possible square submatrices of A, ranging from 1×1 to $\min(m, n) \times \min(m, n)$. The order of the largest non-zero determinant among these submatrices gives us the rank of A.
- If there is a non-zero determinant of order j, but all determinants of order $k + 1$ or higher are zero, then the rank of A is j.

This is then brought to code as the following:

```
def rank(self):
    rows, cols = self.shape()
    j1 = min(rows, cols)

    for i in range(j1, 1, -1):
        if self.get_square_submatrices(i) != -1:
            return i
```

Listing 4: Rank function

The function *get_square_submatrices(i)* is an auxiliary function used to calculate all the possible square submatrices of a matrix of order i:

```
def get_square_submatrices(self, order):
    data = self.get()
    rows, cols = self.shape()
    submatrices = []

    for start_row in range(rows - order + 1):
        for start_col in range(cols - order + 1):
            submatrix = []

            for row in range(order):
                submatrix.append(
                    data[start_row + row][start_col:start_col + order])

            submatrices.append(Matrix(submatrix))

    return submatrices
```

Listing 5: get_square_submatrices

This is then parallelized by distributing the computation of *get_square_submatrices*, creating a further **remote** auxiliary function, *get_submatrix_task*:

```
def get_square_submatrices(self, order):
    data = self.get()
    rows, cols = self.shape()
    futures = []

    for start_row in range(rows - order + 1):
        for start_col in range(cols - order + 1):
            futures.append(t.get_submatrix_task.remote(start_row, start_col,
    order, data))

    return ray.get(futures)
```

Listing 6: get_square_submatrices

```
@ray.remote
def get_submatrix_task(start_row, start_col, order, data):
        from matrix import Matrix
        submatrix = []

        for row in range(order):
            submatrix.append(
                data[start_row + row][start_col:start_col + order])

        return submatrix
```

Listing 7: get_submatrix_task

5 Result of tests

6 Conclusion