Distributed computation of linear algebra operations

Distributed systems and networks laboratory

Gabriele Aloisio [503264]

2023

Università degli studi di Messina

Table of contents

Introduction

Matrix operations

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.

Matrix operations

To address this problem, we are going to use an open-source distributed computing framework called Ray.

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.

Issues with distributed systems

Distributed systems also present unique challenges:

- Data consistency
- Fault tolerance
- Load balancing
- Network latency

Solving challenges with Ray

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
- Scalability
- Fault tolerance
- Resource utilization

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.

Implementation in Python

The Matrix classe

Before implementing the parallelized algorithms for Ray, we wrote the code for the serial execution in the Matrix class:

The Matrix class

```
aboveskip
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): [...]
```

(Most of the functions defined in the class are auxiliary therefore were excluded for practical purposes)

The Matrix class

We modify the Matrix class in a later time to parallelize its operations

Operations

List of operation

We will showcase the following operations:

- Multiplication
- Determinant
- Inverse
- Rank

Multiplication

Multiplication

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.

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

Multiplication

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:

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

Multiplication with Ray (Serial implementation)

```
aboveskip
@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.shape()[0])]
            for i in range(A.shape()[0]):
                for j in range(b_cols):
                    for k in range(a_cols):
                        result[i][j] += A.data[i][k] * B.data[k][j]
            return Matrix(result)
        else:
            raise ValueError(
                "Matrix dimensions do not match for dot product")
    else:
        raise ValueError("Dot product requires a Matrix object")

```

Multiplication with Ray (dot_calc)

To parallelize this function, we simply delegate the calculation to an auxiliary function *dot_calc*:

aboveskip

```python
@staticmethod
@ray.remote
def task_multiply(a, b, i, j, k):
    return a[i][k] * b[k][j]

```

Multiplication with Ray (Parallel)

```
aboveskip
@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):
                        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")
```

Determinant

Determinant

The determinant is a scalar value that is a function of the entries of a square matrix. It characterizes some properties of the matrix and the linear map represented by the matrix. In particular, the determinant is nonzero if and only if the matrix is **invertible**.

Determinant - Laplace method

To compute the determinant of matrices with order greater than 2 we are going to use the Laplace method.

Let A be a square matrix of size $n \times n$. The Laplace expansion of the determinant along the ith row is given by:

$$\det(A) = a_{i1}C_{i1} + a_{i2}C_{i2} + \ldots + a_{in}C_{in}$$

where C_{ij} denotes the cofactor of the element a_{ij}.

Determinant - Cofactor

The cofactor C_{ij} is calculated as follows:

$$C_{ij} = (-1)^{i+j} \cdot \det(M_{ij})$$

where $\det(M_{ij})$ represents the determinant of the submatrix obtained by deleting the ith row and jth column from matrix A.

Determinant - Laplace method

Using the Laplace method, the determinant can be calculated recursively by expanding along any row or column until a 2×2 matrix is reached, for which the determinant can be directly computed.

The Laplace method provides an alternative approach for determining the determinant of a matrix and can be particularly useful for matrices of larger sizes.

It's an essential operation since it's used in the other operations of this study.

Determinant function (Serial)

```
aboveskip
def det(self):
if self.is_square():
    size = self.size()["rows"]
    a = self.get()

    if size == 1:
        return a[0][0]

    elif size == 2:
        return (a[0][0] * a[1][1]) - (a[0][1] * a[1][0])

    else:
        sum = 0

        for i in range(1, size):
            print(i)

            futures = self.task_det.remote(self=self, elements=a, i=
    i)
            print(ray.get(futures))

            sum += ray.get(futures)

        return sum

else:
    raise ValueError("Cannot compute determinant of a non-square
    matrix")


```

Determinant function (minor)

The *minor* function

```
aboveskip
    def minor(self, i, j):
    '''
    Extract a minor matrix by removing the ith row and jth column
    '''

    data = self.get()
    minor_data = [row[:j] + row[j + 1:]
                  for row_idx, row in enumerate(data) if row_idx
     != i]

    return Matrix(minor_data)
```

Then becomes the parallelized function *dist_minor*:

Determinant function (dist_minor)

```
aboveskip
@ray.remote
def dist_minor(A, i, j):
    '''
    Extract a minor matrix by removing the ith row and jth column
    '''

    data = A.get()
    minor_data = [row[:j] + row[j + 1:]
                  for row_idx, row in enumerate(data) if row_idx
     != i]

    return Matrix(minor_data)

```

So finally the distributed determinant function will be:

Determinant function (Parallel)

```
aboveskip
    def det(self):
        if self.is_square():
            data = self.get()
            _, cols = self.shape()

            if cols == 1:
                return data[0][0]

            elif cols == 2:
                return (data[0][0] * data[1][1]) - (data[0][1] * data[1][0])

            else:
                det_value = 0


                minors_futures = [Matrix.dist_minor.remote(self, 0, j) for j in range(cols)]
                minors = ray.get(minors_futures)

                for minor, j in zip(minors, range(cols)):
                    # minor = self.minor(0, j)
                    det_value += ((-1) ** j) * data[0][j] * minor.det()

                return det_value

```

Inverse

Inverse of a matrix (1)

The inverse of a matrix is a fundamental concept in linear algebra. For a square matrix A, if an inverse exists, it is denoted as A^{-1}. A matrix is invertible (or non-singular) if and only if its determinant is non-zero.

The inverse of matrix A satisfies the following condition:

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

where I represents the identity matrix.

Inverse of a matrix (2)

To find the inverse of a matrix, one common method is to use the formula:

$$A^{-1} = \frac{1}{\det(A)} \cdot \operatorname{adj}(A)$$

where $\det(A)$ denotes the determinant of matrix A and $\operatorname{adj}(A)$ represents the adjugate (or adjoint) of matrix A.

The adjugate of matrix A is obtained by taking the transpose of the matrix of cofactors of A. The cofactor C_{ij} is calculated as:

$$C_{ij} = (-1)^{i+j} \cdot \det(M_{ij})$$

where $\det(M_{ij})$ represents the determinant of the submatrix obtained by deleting the ith row and jth column from matrix A.

Inverse function (Serial) (1)

aboveskip

```
def inv(self):
if not self.is_square():
    raise Exception("Matrix must be square")

elif self.det() == 0:
    raise Exception("Matrix is not invertible")

else:
    rows, cols = self.shape()
    data = self.get()

    det = self.det()

    # special case for 2x2 matrix:
    if rows == cols == 2:
        return [[data[1][1] / det, -1 * data[0][1] / det],
                [-1 * data[1][0] / det, data[0][0] / det]]

...

```

Inverse function (Serial) (2)

aboveskip

```
...
    # find matrix of cofactors
    cof_matrix = []

    for row in range(rows):
        cof_row = []

        for column in range(cols):
            minor = self.minor(row, column)

            cof_row.append(((-1)**(row + column)) * minor.det())

        cof_matrix.append(cof_row)

    cof_matrix = Matrix(cof_matrix).transpose()

    cof_rows, cof_cols = cof_matrix.shape()
    cof_data = cof_matrix.get()

    for row in range(cof_rows):
        for column in range(cof_cols):
            cof_data[row][column] = cof_data[row][column] / det

    return cof_matrix

```

Distributing the inverse function

To distribute the inverse function we will need two auxiliary functions: *inv_cof_matrix* and *inv_calc*

inv_cof_matrix

Find all the cofactor matrices of a given matrix

aboveskip

```python
@ray.remote
def inv_cof_matrix(A, row, cols):
    from matrix import Matrix

    cof_row = []
    minor_futures = [Matrix.dist_minor.remote(A, row, col) for col
     in range(cols)]
    minors = ray.get(minor_futures)

    for col,minor in zip(range(cols), minors):
        cof_row.append(((-1)**(row + col)) * minor.det())

    return cof_row

```

inv_calc

Calculation step for the main function

aboveskip

```
@ray.remote
def inv_calc(row, cof_cols, cof_data, det):
    for col in range(cof_cols):
        return (row, col, (cof_data[row][col] / det))

```

Inverse function (Parallel) (1)

aboveskip

```
def inv(self):
if not self.is_square():
    raise Exception("Matrix must be square")

elif self.det() == 0:
    raise Exception("Matrix is not invertible")

else:
    rows, cols = self.shape()
    data = self.get()
    det = self.det()

    # special case for 2x2 matrix:
    if rows == cols == 2:
        m = [[data[1][1] / det, -1 * data[0][1] / det],
                [-1 * data[1][0] / det, data[0][0] / det]]

        return Matrix([[round(i, 8) for i in j] for j in m])
...

```

Inverse function (Parallel) (2)

aboveskip

```python
...
    # find matrix of cofactors
    cof_rows_futures = [t.inv_cof_matrix.remote(self, row, cols) for row in range(rows)]
    cof_matrix = ray.get(cof_rows_futures)
    cof_matrix = Matrix(cof_matrix).transpose()

    cof_rows, cof_cols = cof_matrix.shape()
    cof_data = cof_matrix.get()

    data_futures = [t.inv_calc.remote(row, cof_cols, cof_data, det) for row in range(cof_rows)]

    for data in ray.get(data_futures):
        for row, col, value in [data]:
            cof_data[row][col] = value

    return Matrix([[round(i, 8) for i in j] for j in cof_matrix])

```

Rank

Rank (1)

The rank of a matrix A, denoted as $\mathrm{rank}(A)$, is defined as the maximum number of linearly independent rows or columns in the matrix.

The rank of a matrix can be determined 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.

Criterion of minors

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 $\text{rank}(A)$, 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.

Rank function (Serial)

```
aboveskip
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

```

get_square_submatrices (Serial)

The main protagonist here is the *get_square_submatrices* function:

aboveskip

```python
def get_square_submatrices(self, order):
'''
Auxiliary function for the rank() function
'''
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

```

get_submatrix_task

We will distribute *get_square_submatrices* by defining an auxiliary function *get_submatrix_task*:

aboveskip

```
@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

```

get_square_submatrices (Parallel)

```
aboveskip
def get_square_submatrices(self, order):
'''
Auxiliary function for the rank() function
'''
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)

```

Results of tests

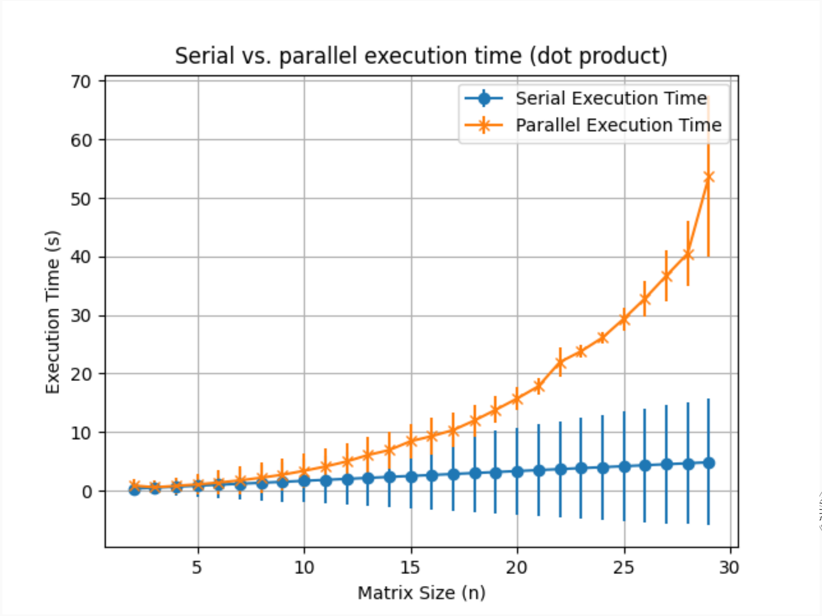
Description of tests

For our experiments, we considered the following parameters:

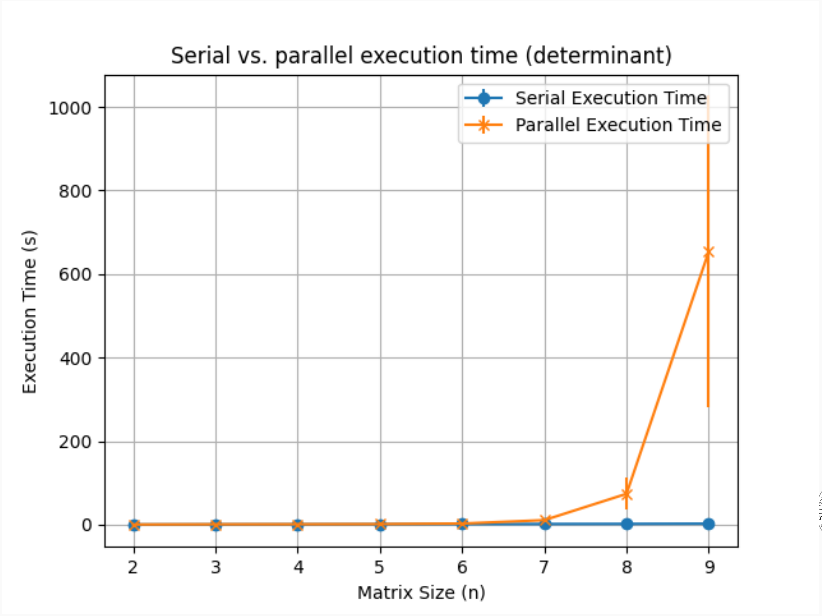
- **Determinant calculation:** $n = 2 \ldots 10$, each n executed three times.
- **Dot product:** $n = 2 \ldots 30$, each n executed five times.
- **Inverse:** $n = 2 \ldots 8$, each n executed three times.
- **Rank:** $n = 2 \ldots 500$, each n executed five times.

For each type of operation, the code iterates through different matrix sizes, executing the operation multiple times for each size to capture variations in performance

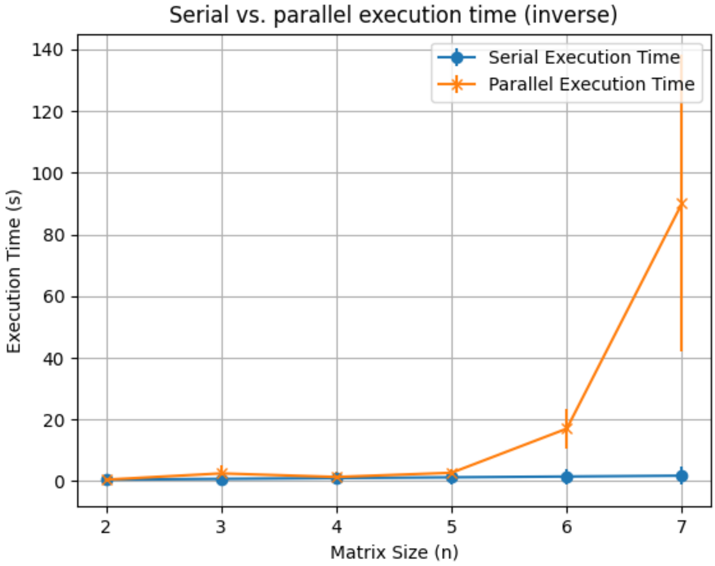
Results of tests (Multiplication)



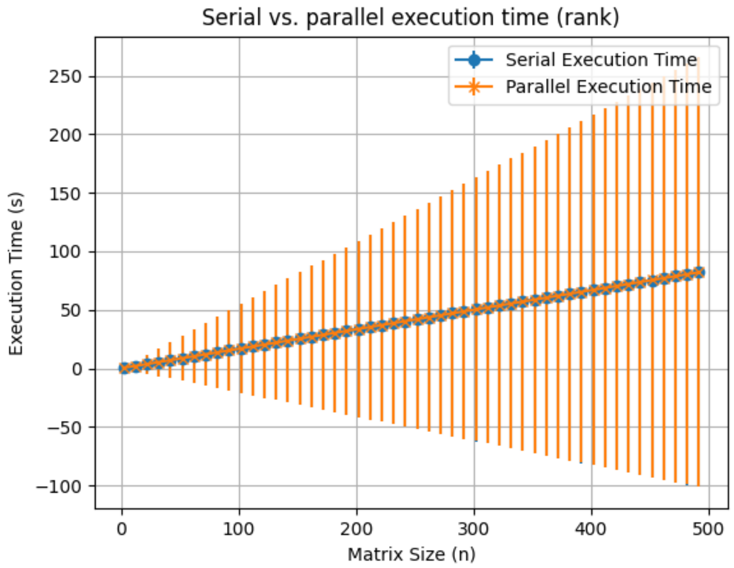
Results of tests (Determinant)



Results of tests (Inverse)



Results of tests (Rank)



Conclusion

Conclusion

Serial execution always outperforms parallel execution, with both serial execution times and standard deviations sensitive to matrix size. This reveals that the advantages of parallelization are not universally applicable, with the performance gains being countered by overhead in some scenarios.

Thanks for reading

☺