M1. Actividad 5. Parallel Data Processing in Apache Spark (using Pyspark)

Mavericks

Gabriela Cortés Olvera - A01751655

José Ángel García Gómez - A01745865

Pablo González de la Parra - A01745096

José María Ibarra Peréz - A01706970

Erika Marlene García Sánchez - A01745158

Zaide Islas Montiel - A01751580

Ana Martínez Barbosa - A01382889

Elisa Sánchez Bahnsen - A01745371

Inteligencia artificial avanzada para la ciencia de datos II

Luis Trejo

Fecha de entrega:
9 de Noviembre de 2023

Tecnológico de Monterrey

Para la realización de la actividad se desarrollaron 2 funciones (Figura 1 y Figura 2) y un ciclo que va de 200 a 400 con pasos de 10 (Figura 3), lo anterior permitió que se pudiera correr los códigos de manera secuencial y paralelizada.

```python
# Create and start Spark session
app_name = 'PySpark Matrix Multiplication Example'
master = 'local'
spark = SparkSession.builder.appName(app_name).master(master).getOrCreate()
spark.sparkContext.setLogLevel("ERROR")

# Method to create a populated matrix of size N * N, with potential
# values ranging from (max_value - 1) through to max_value.
def create_matrix(size, max_value):
    return [[random.randint((max_value * -1), max_value) for i in range(size)] \
            for j in range(size)]

# Method to create a matrix populated with 0's of size N * N.
def create_empty_matrix(size):
    return [[0 for i in range(size)] for j in range(size)]

# Method to multiply two matrices of the same dimensions, i.e. N * N.
def matrix_multiply(A, B, C, size):
    for i in range(size):
        for j in range(size):
            total = 0 # Initialise total to 0
            for k in range(size):
                total += A[i][k] * B[k][j] # Perform matrix multiply
            C[i][j] = total

    # Return the result of the matrix multiplication
    return C


def getSequentialExecutionTime(N):

  # Initialise matrices
  A = create_matrix(N, 500)
  B = create_matrix(N, 500)
  C = create_empty_matrix(N)

  print('Performing standard matrix multiplication')

  # Perform and time matrix multiplication
  start = timer() #
  C = matrix_multiply(A, B, C, N) #
  end = timer() #

  return end - start
```

Figura 1. Ejecución secuencial.

```python
# Method to convert a Resilient Distributed Dataset (RDD) to a BlockMatrix object
def as_block_matrix(rdd, rows, columns):
    return IndexedRowMatrix(
        rdd.zipWithIndex().map(lambda i: IndexedRow(i[1], i[0]))
    ).toBlockMatrix(rows, columns)

# Method to convert an indexed row matrix to a local array using Scipy 'lil_matrix'
def indexedrowmatrix_to_array(matrix):
    # Create an empty array of the same dimensions as the matrix
    result = lil_matrix((matrix.numRows(), matrix.numCols()))

    # Iterate through each row and set values in the empty array
    for indexed_row in matrix.rows.collect():
        result[indexed_row.index] = indexed_row.vector

    # Return the local array
    return result

def getParallelExecutionTime(N):
  # Initialise matrices
  A = create_matrix(N, 500)
  B = create_matrix(N, 500)
  C = create_empty_matrix(N)

  # Convert arrays to RDDs
  A_rdd = spark.sparkContext.parallelize(A)
  B_rdd = spark.sparkContext.parallelize(B)

  # Perform and time matrix multiplication
  start = timer() #
  C_matrix = as_block_matrix(A_rdd, N, N).multiply(as_block_matrix(B_rdd, N, N)) #
  end = timer() #

  return end - start


# +=====================================================================+
# |                        End of File                                  |
# +=====================================================================+
```

Figura 2. Ejecución paralelizada.

```python
parallelExecutionTimes = []
sequentialExecutionTimes = []
n = []

for i in range (200, 2000, 200):
  print("N: ", i)
  sequentialTime = getSequentialExecutionTime(i)
  print("Sequential Time: ", sequentialTime)
  parallelTime = getParallelExecutionTime(i)
  print("Parallel Time: ", parallelTime)
  parallelExecutionTimes.append(parallelTime)
  sequentialExecutionTimes.append(sequentialTime)
  n.append(i)

print(parallelExecutionTimes)
print(sequentialExecutionTimes)
```

Figura 3. Ciclo “for”.

Para poder observar las diferentes ejecuciones de N se fueron almacenando los diferentes valores de corridas e incluso se graficaron los resultados para poder tener mejores insights de las corridas (Figura 4).

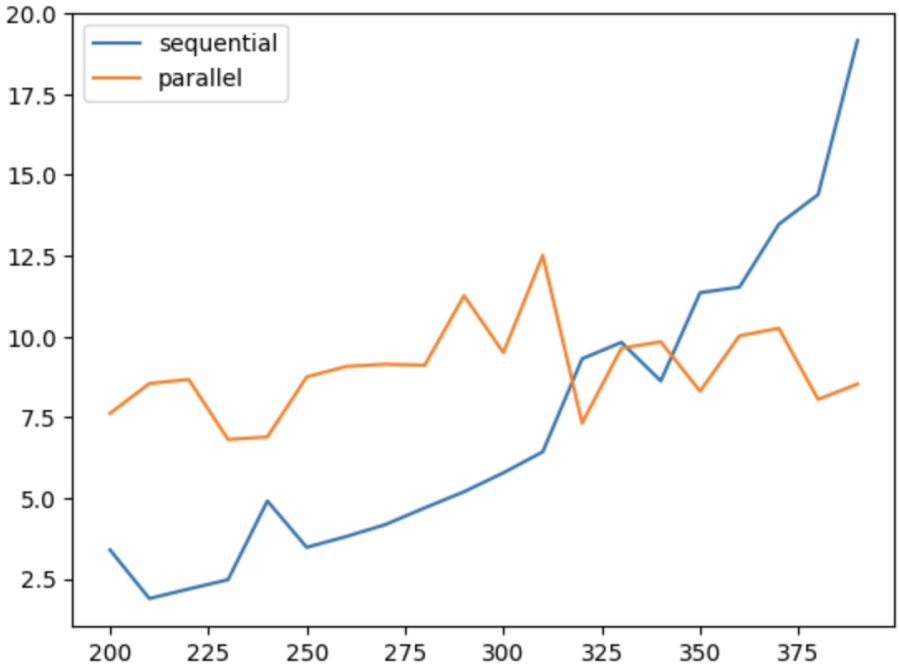

Figura 4. Ciclo “for”.

En la Figura 4 se puede observar que hay más de un punto donde se intersectan las gráficas, sin embargo, en la primera N donde se intersectan es aproximadamente en 320. De igual manera, mediante la gráfica se puede observar que contrario a como comienzan las ejecuciones a mayor N mejor performa las ejecuciones paralelizadas.

Por último, el N aproximado para una corrida de 5 minutos utilizando la ejecución paralelizada es de 2950. Lo anterior se corroboró mediante un get time en el código de python.