ASSIGNMENT 3

PROBLEM STATEMENT:

Implement Matrix Multiplication using Map-Reduce.

OBJECTIVE:

1. Implement Matrix Multiplication using the Map-Reduce programming model.
2. Understand the parallel processing capabilities of Map-Reduce for large-scale data.

Software Requirements

Hadoop (or any other Map-Reduce framework)

Hardware Requirements

A cluster of machines for distributed processing (number of nodes depends on the scale of data)

Data set

Input matrices for multiplication. This could be generated or sourced from real-world scenarios.

Libraries/modules used

Hadoop Map-Reduce libraries.

Theory

Matrix multiplication is a fundamental operation in linear algebra and is often used in various computational tasks. In the context of big data analysis, implementing matrix multiplication using MapReduce is an interesting and challenging task. MapReduce is a programming model commonly used for processing and generating large datasets in a distributed and parallel manner. Here's an overview of how you can implement matrix multiplication using MapReduce.

Matrix Multiplication using MapReduce

Input Data:

Assume you have two matrices, A and B, that you want to multiply.

Matrix A:

```
A = | a11 a12 |
    | a21 a22 |
```

Matrix B:

```
```

B = | b11 b12 |

| b21 b22 |

```

**MapReduce Steps**

1. Map Step: -

- Each Mapper takes a portion of Matrix A and Matrix B as input.

- The key is the row index for Matrix A and column index for Matrix B.

- The value is the corresponding element.

Mapper 1:

```

Input: (i, A, aij)

Output: [(j, ('A', i, aij))] # j is the column index of matrix B

```

Mapper 2:

```

Input: (j, B, bjk)

Output: [(i, ('B', j, bjk))] # i is the row index of matrix A

```

2. Shuffle and Sort:

- The framework groups the intermediate key-value pairs by key.

3. Reduce Step: - Each Reducer receives a group of key-value pairs with the same key.

- It multiplies the corresponding elements and sums up the products.

Reducer:

```

Input: (j, [('A', i, aij), ('B', j, bjk), ...])

Output: [(('A', i, aij), ('B', j, bjk)), ...]

```

4. Final Output:

- The final output is a set of key-value pairs representing the result matrix.

```

Output: (i, j, Cij) # Cij is the element at row i and column j of the result matrix C

```

Example: Let's say we want to multiply two 2x2 matrices: Matrix A:
```

```
| 1 2 |
| 3 4 |
```

Matrix B:

```
 | 5 6 | | 7 8 |
```

The MapReduce process would involve breaking down the matrices into key-value pairs, shuffling and sorting, and then performing the multiplication and summing in the Reducer. The final output would be the result matrix C:

Result Matrix C:

```
| 19 22 | | 43 50 |
```

This process showcases how MapReduce can be leveraged for distributed and parallel computation of matrix multiplication, making it suitable for handling large datasets in big data scenarios.

Algorithm of work –

• Divide the matrix multiplication into Map and Reduce phases.
• Map phase: Emit intermediate key-value pairs for each element in the input matrices.
• Shuffle and Sort phase: Group intermediate pairs by key (matrix element position).
• Reduce phase: Perform the actual multiplication and summing to get the result.

Applications

• Use cases include large-scale scientific computations, machine learning algorithms, and data processing tasks where matrices are involved.

Code

Mapper:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
```

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class MatrixMapper extends Mapper {
private Text outKey = new Text();
private Text outValue = new Text();
protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);
int i = Integer.parseInt(tokenizer.nextToken());
int j = Integer.parseInt(tokenizer.nextToken());
int valueIJ = Integer.parseInt(tokenizer.nextToken());
// Emit key-value pairs for multiplication in the reducer
for (int k = 0; k < context.getNumReduceTasks(); k++) {
outKey.set(j + "," + k); // Group elements by column index (j)
outValue.set(i + "," + valueIJ);
context.write(outKey, outValue);
} }
```

Reducer:

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class MatrixReducer extends Reducer {
private IntWritable result = new IntWritable();
protected void reduce(Text key, Iterable values, Context context) throws IOException, InterruptedException {
List iValues = new ArrayList<>();
List AValues = new ArrayList<>();
for (Text value : values)
```

```
{
String[] parts = value.toString().split(",");
int i = Integer.parseInt(parts[0]);
int Aij = Integer.parseInt(parts[1]);
iValues.add(i); AValues.add(Aij);
}
// Perform multiplication and aggregation
for (int i = 0; i < iValues.size(); i++)
{
int product = AValues.get(i); // Assuming corresponding Bjk values are available in the reducer context
result.set(product);
context.write(new Text(iValues.get(i) + "," + key.toString().split(",")[0]), result); // Emit final (i,j) result
} } }
```

Driver:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class MatrixMultiplyDriver
{
public static void main(String[] args) throws Exception
{
if (args.length != 3)
{
System.err.println("Usage: MatrixMultiplyDriver <input path matrix A> <input path matrix B>
<output path>”);
System.exit(-1);
}
Configuration conf = new Configuration();
```

```
Job job = Job.getInstance(conf, "Matrix Multiplication");
job.setJarByClass(MatrixMultiplyDriver.class);
job.setMapperClass(MatrixMapper.class);
job.setReducerClass(MatrixReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0])); // Input path for matrix A // Assuming matrix B is accessible within the reducer context
FileOutputFormat.setOutputPath(job, new Path(args[2])); // Output path for result matrix
System.exit(job.waitForCompletion(true) ? 0 : 1);
} }
```

Steps to run the code

Here are the steps to run the matrix multiplication MapReduce program using the provided code:

1. Prerequisites:

• Hadoop environment: Ensure you have a Hadoop environment set up and configured on your system. This includes setting up HDFS, Yarn, and necessary environment variables.

• Java compiler: You'll need a Java compiler like javac to compile the provided code.

• Input files: Prepare two text files containing your matrices (matrix A and matrix B) in the format described earlier (rows separated by newlines, elements within rows separated by spaces).

2. Compile the code:

Open a terminal in your Hadoop environment directory and compile the Java files for the mapper, reducer, and driver: javac MatrixMapper.java MatrixReducer.java MatrixMultiplyDriver.java

3. Submit the MapReduce job:

Run the driver class with the path to your input files (matrix A and matrix B) and the desired output directory:

hadoop jar <your_jar _file name>.jar MatrixMultiplyDriver <input_path_matrix A> <input_path_matrix B> <output_path>

Replace <your_jar _file name>.jar with the actual name of the JAR file created after compiling the code.

Replace <input_path_matrix A>, <input_path_matrix B> and <output_path>with the actual paths to your input and output directories.

4. Monitoring and Output:

The job will be submitted to the Hadoop cluster and start processing. You can monitor the progress using the Hadoop web interface or command-line tools. Once the job completes successfully, check the output directory for the resulting matrix with elements (i, j) calculated by the program.

Running Example:

Assume you have two text files matrix_a.txt and matrix_b.txt containing your matrices and an output directory named result. you compile the code and run the driver with the following command: hadoop jar matrix_multiply.jar MatrixMultiplyDriver matrix_a.txt matrix_b.txt result.

If the job finishes successfully, you'll find the resulting product matrix with elements (i, j) in the result directory.

Note: This is a basic example. You might need to adapt the code and steps based on your specific matrices, Hadoop configuration, and desired output format. Remember to modify the reducer logic if matrix B isn't directly accessible within the reducer context.

CONCLUSION:

In this way we have Implemented Matrix Multiplication using Map-Reduce.

ORAL QUESTION:

1. Can you explain how MapReduce can be used to parallelize Matrix Multiplication?
2. What are the key components of a MapReduce implementation for Matrix Multiplication?
3. What challenges might arise when dealing with very large matrices in MapReduce?
4. What optimizations could you implement to improve the performance of Matrix Multiplication in MapReduce?

```
import multiprocessing

def matrix_multiply_mapper(row, col):
    result = 0
    for i in range(len(row)):
        result += row[i] * col[i]
    return result

def matrix_multiply_worker(args):
    row_index, row, columns = args
    return [(row_index, col_index, matrix_multiply_mapper(row, col))
            for col_index, col in enumerate(columns)]

def matrix_multiply_reduce(results):
    final_result = {}
    for row_index, col_index, value in results:
        if row_index not in final_result:
            final_result[row_index] = {}
        final_result[row_index][col_index] = value

    return final_result

def map_reduce_matrix_multiply(matrix1, matrix2):
    num_workers = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes=num_workers)

    args = [(i, matrix1[i], matrix2) for i in range(len(matrix1))]
    intermediate_results = pool.map(matrix_multiply_worker, args)

    pool.close()
    pool.join()

    final_result = matrix_multiply_reduce(
        [item for sublist in intermediate_results for item in sublist])

    return final_result

if __name__ == "__main__":
    matrix1 = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]

    matrix2 = [
        [9, 8, 7],
        [6, 5, 4],
        [3, 2, 1]
    ]

    result = map_reduce_matrix_multiply(matrix1, matrix2)
    for row_index, row in result.items():
        print(row)
```

```
{0: 46, 1: 28, 2: 10}
{0: 118, 1: 73, 2: 28}
{0: 190, 1: 118, 2: 46}
```