MAPREDUCE, HAPDOOP, AND SPARK

Varun Jain 21963986, Kieron Ho 20500057

Assignment 2
CITS3402 High Performance Computing

Table of Contents

MapReduce

The MapReduce segment of the report is a summary of information found in the paper “MapReduce: Simplified data processing on large clusters” by J. Dean and S. Ghemawat, specifically paper sections 1, 2, and 3

1. Introduction

Hadoop MapReduce is a programming paradigm designed to process and then generate large volume datasets using parallelisation across clusters of commodity machines. The computation process is designed to break down large datasets and distribute these data blocks across multiple worker machines to concurrently run tasks on these data blocks at a greater completion speed. While these machines work to complete the same singular task, they do not interact with each other, instead receiving and sending all correspondence to a master or driver machine. The use of cheaper commodity machines linked in a cluster network allows large scale or complex problems to be solved efficiently and in a timely manner.

2. Programming Model

The MapReduce library features two predominant functions that make up its name: map and reduce function.

1. Mapping, which is applying the same function to all values in a list, is designed to separate out and prepare the full dataset for parallel operations. This process takes a set of <key, value> pairs, converts the dataset into a set of intermediate <key, value> pairs based on required operations, and runs the operations on them.
2. Reducing, which takes the pairs operated on from the above process and resolves a solution by sending the appropriate results to the driver machine. The results sent to the driver are by design significantly smaller than the datasets from which they are derived, hence the name Reduce. The reduce function will the take the associated value with each key and will merge together all the corresponding values for each key.

3. Example

One such problem which perfectly demonstrates the capability and purpose of MapReduce is the word count problem. In this problem, the number of times a distinct word occurs in a dataset of files must be found in a timely manner. The solution should return a single number, being the number of times the word is found to occur in the large collection of datasets. The workflow will look like the following:

(Mapping Stage)

Dataset

↓

Convert to pair <filename, contents of file>

↓

Intermediate pairs created <sought word, fraction of file contents>

↓

Operation run on fraction of file contents produces <sought word, 1> for each occurrence

↓

(Reduction Stage)

The nodes gather a list of occurrences <sought word, list<sought word, 1>>

↓

Occurrences are summed and sent to the driver machine, which calculates the full solution. In other words, reduce function will combine all the value for a distinct key and returns total number of times a distinct word appears in the dataset.

4. Other Examples

The following lists possible uses of MapReduce in computing solutions to problems that involve large datasets to be operated on.

- Distributed Grep: Grep is a command line utility that is designed to search a plain text dataset for a matching regular expression. Clearly here the second key that is passed to the worker machines will be the regular expression to search for, the value passed to each worker will be a fraction of the total plaintext dataset to search for, and the reduced value will be lines that match the desired regular expression

- Reverse Web-Link: This process is designed to find each source page that contains a link to the target page. The second key will be the URL to the target machine, the value passed will be a fraction of a list of URLs to examine, and the reduced value will be a list of all the URLs that contain links to the target URL.
- Count of URL Access Frequency: This process contains similarities to the word count problem as described above but is designed to count the number of times a website is requested based on web logs. The second key will be the web page of interest, the values will be fractions of the web logs, and the reduced value will be a count of the number of times a web page is requested.
- Turn-Vector per Host: This problem involves finding the most important (i.e. Most common) words in a document or set of documents. As opposed to the word count problem above, each word, or "term vector" has a frequency counter created for it and is matched to the document from which it was found in. The worker reduce function will send to the driver the most common term vector from each document.
- Inverted Index: This process contains similarities to the Reverse Web-Link computation. But instead, the goal for this application is to return all the documents which contain a distinct word. The map function would take the document ID as the input key. After computation, the function would parse through each document and produce <word, document ID > pairs. The MapReduce Library will automatically sort all the intermediate values (i.e. document IDs) together for a distinct word. The reduce function accepts these pairs and combines the corresponding document IDs for it.

5. Execution Overview

The MapReduce Library automatically partitions input files into pieces which are distributed across many commodity worker machines. Based on this framework each worker will run a copy of an operation on these distributed datasets concurrently. A master copy will be allocated, while any idle machines will be assigned either a Map task or a Reduce task. On receiving a Map task, a worker will read the contents of its input, parse into <input key, input value> pairs and pass it through the user-defined Map Function. The data generated by the Map function will be a set of <output key, output value> pairs which are subsequently buffered into memory. The master or driver will notify Reduce task allocated machines as to the location of intermediate Map results, which will be collected using remote procedure calls. These results are sorted, and the users reduce function will generate an intermediate set of <key, value> pairs, which will be appended to a final output to reduce partition. Once the task has completed, the master copy wakes the user program.

<u>Spark</u>

The Spark segment of the report is a summary of the information found in the paper "Spark: Cluster computing with working sets" by M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica, specifically sections 1, 2, 3.1, 4 and 6 of the paper.

1. Introduction

MapReduce is a large-scale data processing method that uses parallelization to improve processing performance, however it has some limitations based on its design. For example, the MapReduce paradigm is acyclic, since it operates on processing one iteration, and receives the result marking the end of the task. This restricts it from being used in situations where a cyclic process is required to find the desired solution. This is where the Spark paradigm may be used. Spark functionality adds the re-use of datasets during parallel operations, while also having important features of MapReduce such as fault-tolerance and scalability.

2. MapReduce vs Spark Comparison

There are a few situations where Spark is a superior choice to MapReduce, including situations with:

- Iterative Jobs: These are situations where the results of calculations could be used as in input to the next operation as opposed to being discarded. One such example is for an operation that requires a refining or optimising process to produce a desired result. (Bekker, 2018)
- Interactive Analytics: These are situations where a user queries a dataset in search of a value. Using MapReduce, these queries must be loaded into worker memories before being searched, however since MapReduce uses the slower disk memory, these

queries may take significantly longer to process than required. Spark on the other hand uses RAM memory for operations, resulting in faster searching operations due to the faster speed of RAM compared to disk memory

Another benefit of Spark over MapReduce is the differing data flow paradigms, and how tasks are handled. MapReduce tasks are run with a batch-processing structure, meaning all datasets required in the calculation must be collected and identified prior to operations commencing. Results are only produced at the very end of the full dataset evaluation, resulting in a delay time between the initiation and evaluation of the data. Based on the size of this dataset the results could be a delay between seconds to hours. Spark uses a different data flow method that enables the "streaming" of data into the evaluation pipeline, allowing for real-time analysis results to be obtained. One such example is a monitoring of temperatures or climate conditions to indicate if certain conditions are met.

Spark is implemented using the Scala programming language. Scala allows the use of object-oriented programming such as that found in Java, while maintaining the functionality and ease of use found in a high-level language.

3. Programming Model

3.1 Resilient Distributed Dataset

Resilient Distributed Dataset (RDD) and parallel operations are two concepts fundamental to parallel programming with Apache Spark.

Resilient Distributed Dataset (RDD) is a collection of read-only objects which are distributed across a cluster of commodity machines. RDD elements are divided into logical partitions, spread across multiple computers. The datasets will be processed in parallel, with no communication between worker machines. RDDs are considered resilient because they are immutable and are fault tolerant – RDDs cannot be changed once they are created and also possess a system of self-recovery. This system allows RDDs to recover any missing or damaged partitions, for example in case there is a node failure. RDD's may be constructed in four ways, which will be listed below.

The first way a new Resilient Distributed Data's could be created is through Spark transformation. Spark Transformation takes the existing RDD as input and, depending on the type of transformation function that has been implemented by the user, produces one or more RDDs by the transformation operation. There are three different types of transformation: map, filter and flat-map.

1. The map() function reads the RDD line by line, and transforms its N length RDD into a new RDD with the same length (Team, D., 2016).
2. The filter() function returns new RDD containing elements that match the predicate (Data Flair, 2018),
3. The flatmap() function returns a new RDD with a length of M from the transformation operation of N length RDD. In other words, this function can produce 0, 1, or more elements and store it in a new RDD, depending on how the user implements the map() function (Team, D., 2016).

The remaining ways that RDDs can be made are:

- The alteration of the persistence of the existing RDD in memory through the following two actions: cache and save. The cache action keeps the RDD lazy, however persists the RDD in-memory, after it has been evaluated. This will allow it to be re-used in the future if required. The save action takes the result of the RDD, after evaluation and stores it to the distributed filesystem.
- By parallelizing the existing collection in the driver program. This means that the collection will be partitioned evenly across the multiple nodes.
- Through a file in a shared file system.

3.2 Parallel Operations

RDD can perform many different parallel computations through user-defined functions: reduce, collect and foreach.

- The reduce() function combines the elements contained inside the multiples datasets together using an associative function. The output produced is stored in the driver program.

- The collect() function returns the RDD contents to the driver program. (Data Flair, 2018)
- Finally, the foreach() function, it iterates through each element found in the RDD and manipulates the data depending on how the user implements the function.

3.3 Shared Variables

The worker node executes closures passed to the Spark operation. A closure refers to functions such as map, filter and reduce used in the Spark operations. Each worker will be given its own copy of all the variables, and no updates will propagate back to the remote machine. There are two limited type of shared variables provided by Spark: broadcast variables and accumulators.

- Broadcast variables work like global variables. These are read-only datasets, which are distributed and processed simultaneously across a large number of computers. These variables are immutable and fit-in within the memory of one machine. Every worker will have a copy of the variables, in other words, the variables can be accessed by any worker.
- Accumulator variables are used for global counter variables and add operation, (with a starting value of 0), using an associative operation. These variables can only be read by the driver. So, a simple example for this would be to keep track certain log type records identified through a count implementation.

4. Text Search Example

This example demonstrates the use of Spark in the search of a key word “ERROR” in a large text-based data file.

Data Flow:

Hadoop Distributed File System

↓

Filter for and collect lines that contain word “ERRORS”

↓

Map resulting dataset to reduction values (and possibly store in cache)

↓

Send to driver to be reduced appropriately

5. Implementation

5.1 Iterative and caching capabilities

After being processed, datasets can be cached as an RDD to be used further. In the above text search example, the operation searched for and stored the lines that contained the word “ERROR”. This information can be stored as a “chain of objects” that capture the lineage (or object and its predecessors) of each resulting RDD. This means that any object recorded contains its information on the transformations that lead to its current state. This can be used in case of errors to recover lost values.

Another feature is the operations that are implemented on the RDDs. These are:

1. getPartitions which returns a list of partition IDs
2. getIterator (over the partitions) which returns an iterator for a partition
3. getPrefferredLocations (for each partition) which is used for task scheduling to achieve data locality

When a task is initiated on Spark, file blocks are distributed with a priority based on their “preferredLocation” value.
Once a task has been assigned and launched, the “getIterator” operation is called to start reading the partitions.

5.2 Resilient Distributed Dataset Types

There are three main types of RDDs to look at during the implementation of Spark:

1. HdfsTextFile partitions based on block ID, applies preferred locations based on the lock locations, and the getIterator operation opens a data stream to the block
2. MappedDataSet partitions and sets preferred locations based on the parent RDD, and the iterator applies a map function to the parent elements
3. CachedDataSet is the cached data resulting from a previous operation. Its preferred location is initially the parent location, where it can subsequently be updated to whichever node it was cached on. If a worker node is lost, the RDD can be recalculated based on the parent it was transformed from.

5.3 Closures

Sending “closures” is easy using the implemented programming language Scala, since the whole of the closure can be sent as a serialized object. The only downside to this implementation is that there are rare cases where object references found in the outer scope of a closure is not actually used in the body of the closure.

5.4 Shared Variables

1. Broadcast variables are sent to a file in the shared system. When a worker requires the variable’s value, it is searched for first on a local cache, and if not found is subsequently searched for on the shared file system
2. Accumulator variables are distributed to each worker with unique serial values. After each task completion the accumulator value is sent to the driver program, consequently reset to zero. The driver ensures that each worker partition sends its results only once to prevent duplication reception of results

6. Related Function Implementations

6.1 Distributed Shared Memory

DSMs (Distributed Shared Memory) are similar implementations of the memory distribution paradigm to RDDs (Resilient Distributed Datasets), with some key differences. For an initial comparison RDD can be viewed as an abstraction of DSM. RDD also features a greater restriction on its programming model, but which results in the capability to rebuild data lost through node failure.

While DSM achieves a level of fault tolerance through a check pointing system, RDDs use a lineage system based on the object nature of RDDs, to only require reconstruction of the specific dataset that was lost.

Another key difference is that RDDs distribute data much the same as MapReduce, instead of DSM’s method of allowing nodes to access some global address space

6.2 Language Integration

Spark's chosen language is Scala, which features similarities to DryadLINQ that utilises .NET's support for "integrated queries" to run queries on a cluster. However, unlike DryanLINQ, Spark allows for operation results to be cached and reused across parallel operations. Spark also enables the use of shared variables (broadcast type and accumulator type). Scala allows the serialisation of closure objects which improves the ability to distribute the closures to the connected workers.

6.3 Lineage

The object serialisation feature of Scala and Spark allows the lineage details of datasets to be captured. This is useful for full analytical workflow explanations, data replication, and data recovery.

References

Bekker, A. (2018). *Spark vs. Hadoop MapReduce: Which big data framework to choose*. [online] Scnsoft.com. Available at: https://www.scnsoft.com/blog/spark-vs-hadoop-mapreduce

Data Flair. (2018). *Spark RDD Operations-Transformation & Action with Example*. [online] Available at: https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Dean, J. and Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. [online] Available at: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

Team, D. (2016). *Apache Spark Map vs FlatMap Operation – DataFlair*. [online] Data-flair.training. Available at: https://data-flair.training/blogs/apache-spark-map-vs-flatmap/

Zaharia, M. (2018). *Introduction to MapReduce and Hadoop*.

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S. and Stoica, I. (2018). *Spark: Cluster Computing with Working Sets*. [online] Usenix.org. Available at: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf