International Journal of Scientific and Research Publications, Volume 2, Issue 11, November 2012
ISSN 2250-3153

Implementation of Spark Cluster Technique with SCALA

Tarun Kumawat[1], Pradeep Kumar Sharma[2], Deepak Verma[3], Komal Joshi[4], Vijeta Kumawat[5]

[1] Tarun Kumawat [CSE],JECRC UDML College of Engineering, Kukas, Jaipur, Rajasthan, India
[2] Pradeep Kumar Sharma [CSE], JECRC UDML College of Engineering, Kukas, Jaipur, Rajasthan, India
[3] Deepak Verma [CSE], Jaipur National University, Jagatpura, Jaipur Rajasthan, India
[4] Komal Joshi [CSE], Rajasthan Technical University, Kota, Rajasthan, India
[5] Jyoti Vidyapeet Woman's University, Jaipur, Rajasthan,India

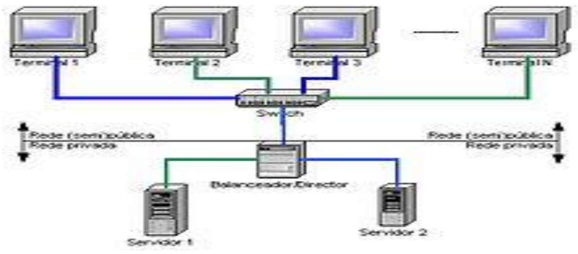
Abstract- Spark is an open source cluster computing system that aims to make data analytics fast — both fast to run and fast to write. To run programs faster, Spark provides primitives for in-memory cluster computing: your job can load data into memory and query it repeatedly much quicker than with disk-based systems like Hadoop Map Reduce. To make programming faster, Spark integrates into the Scala language, letting you manipulate distributed datasets like local collections. You can also use Spark interactively to query big data from the Scala interpreter.

Index Terms- Cluster Spark, Scala, RDD, MapReduce , Hadoop

I. Introduction

The formal *engineering* basis of cluster computing as a means of doing parallel work of any sort was arguably invented by Gene Amdahl of IBM, who in 1967 published what has come to be regarded as the seminal paper on parallel processing: Amdahl's Law. Amdahl's Law describes mathematically the speedup one can expect from parallelizing any given otherwise serially performed task on a parallel architecture. This article defined the engineering basis for both multiprocessor computing and cluster computing, where the primary differentiator is whether or not the inter-processor communications are supported "inside" the computer (on for example a customized internal communications bus or network) or "outside" the computer on a *commodity* network.

Computer clusters may be configured for different purposes ranging from general purpose business needs such as web-service support, to computation-intensive scientific calculations. In either case, the cluster may use a high-availability approach. Note that the attributes described below are not exclusive and a "compute cluster" may also use a high-availability approach, etc.



A load balancing cluster with two servers and 4 user stations

"Load-balancing" clusters are configurations in which cluster-nodes share computational workload to provide better overall performance. For example, a web server cluster may assign different queries to different nodes, so the overall response time will be optimized.[8] However, approaches to load-balancing may significantly differ among applications, e.g. a high-performance cluster used for scientific computations would balance load with different algorithms from a web-server cluster which may just use a simple round-robin method by assigning each new request to a different node.

"Computer clusters" are used for computation-intensive purposes, rather than handling IO-oriented operations such as web service or databases.[9] For instance, a computer cluster might support computational simulations of weather or vehicle crashes. Very tightly coupled computer clusters are designed for work that may approach "supercomputing".

"High-availability clusters" (also known as failover clusters, or HA clusters) improve the availability of the cluster approach. They operate by having redundant nodes, which are then used to provide service when system components fail. HA cluster implementations attempt to use redundancy of cluster components to eliminate single points of failure. There are commercial implementations of High-Availability clusters for many operating systems. The Linux-HA project is one commonly used free software HA package for the Linux operating system.

II. Cluster Management

One of the challenges in the use of a computer cluster is the cost of administrating it which can at times be as high as the cost of administrating N independent machines, if the cluster has N nodes. In some cases this provides an advantage to shared memory architectures with lower administration costs. This has also made virtual machines popular, due to the ease of administration.

III. Task Scheduling

When a large multi-user cluster needs to access very large amounts of data, task scheduling becomes a challenge. The MapReduce approach was suggested by Google in 2004 and other algorithms such as Hadoop have been implemented.

However, given that in a complex application environment the performance of each job depends on the characteristics of the underlying cluster, mapping tasks onto CPU cores and GPU devices provides significant challenges. This is an area of ongoing research and algorithms that combine and extend MapReduce and Hadoop have been proposed and studied.

IV. NODE FAILURE MANAGEMENT

When a node in a cluster fails, strategies such as "fencing" may be employed to keep the rest of the system operational. Fencing is the process of isolating a node or protecting shared resources when a node appears to be malfunctioning. There are two classes of fencing methods; one disables a node itself, and the other disallows access to resources such as shared disks.

The STONITH method stands for "Shoot the Other Node in the Head", meaning that the suspected node is disabled or powered off. For instance, *power fencing* uses a power controller to turn off an inoperable node.[18]

The *resources fencing* approach disallows access to resources without powering off the node. This may include *persistent reservation fencing* via the SCSI3, fibre Channel fencing to disable the fibre channel port or global network block device (GNBD) fencing to disable access to the GNBD server.

V. INTRODUCTION OF SPARK

Spark is an open source cluster computing system that aims to make data analytics *fast* — both fast to run and fast to write. To run programs faster, Spark provides primitives for in-memory cluster computing: your job can load data into memory and query it repeatedly much quicker than with disk-based systems like Hadoop MapReduce. To make programming faster, Spark integrates into the Scala language, letting you manipulate distributed datasets like local collections. You can also use Spark interactively to query big data from the Scala interpreter.
Spark is implemented in Scala, a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster. Although our implementation of Spark is at an early stage, experience with the system is encouraging. Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 40 GB dataset with sub-second latency. Several machine learning researchers in our lab have used Spark implement a number of algorithms, including logistic regression, expectation maximization, and bootstrap sampling.
Spark was initially developed for two applications where keeping data in memory helps: *iterative* algorithms, which are common in machine learning, and *interactive* data mining. In both cases, Spark can outperform Hadoop MapReduce by **30x**. However, you can use Spark for general data processing too. Spark runs on the Apache Mesos cluster manager, letting it coexist with Hadoop. It can also read any data source supported by Hadoop.
Spark was developed in the UC Berkeley AMP Lab. It's used by several groups of researchers at Berkeley to run large-scale applications such as spam filtering, natural language processing and road traffic prediction. It's also used to accelerate data analytics at Conviva, Klout, and Quantifind, and other companies.

VI. IMPLEMENTATION OF SPARK

Spark is built on top of Mesos, a "cluster operating system that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. This allows Spark to run alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI, and share data with them. In addition, building on Mesos greatly reduced the programming effort that had to go into Spark. The core of Spark is the implementation of resilient distributed datasets. As an example, suppose that we define a cached dataset called cachedErrs representing error messages in a log file, and that we count its elements using map and reduce,

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

These datasets will be stored as a chain of objects capturing the lineage of each RDD, shown in Figure 1. Each dataset object contains a pointer to its parent and information about how the parent was transformed. Internally, each RDD object implements the same simple interface, which consists of three operations:

• getPartitions, which returns a list of partition IDs.
• getIterator(partition), which iterates over a partition.
•getPreferredLocations(partition), which is used for task scheduling to achieve data locality.

VII. SPARK EXAMPLES

Spark is built around *distributed datasets* that support types of parallel operations: transformations, which are lazy and yield another distributed dataset (e.g., map, filter, and join), and actions, which force the computation of a dataset and return a result (e.g., count). The following examples show off some of the available operations and features.

7.1 Text Search

In this example, we search through the error messages in a log file:

```
val file = spark.textFile("hdfs://...")
val errors = file.filter(line => line.contains("ERROR"))
// Count all the errors

errors.count()
// Count errors mentioning MySQL
```

```
errors.filter(line => line.contains("MySQL")).count(
// Fetch the MySQL errors as an array of strings
errors.filter(line => line.contains("MySQL")).collect()
```

The red code fragments are Scala function literals (closures) that get passed automatically to the cluster. The blue ones are Spark operations.

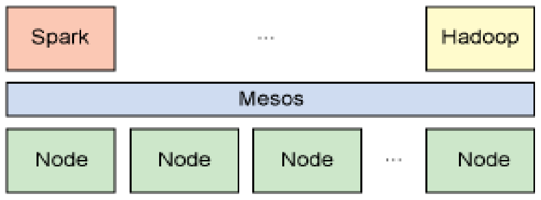
Spark cluster computing architecture

Although Spark has similarities to Hadoop, it represents a new cluster computing framework with useful differences. First, Spark was designed for a specific type of workload in cluster computing—namely, those that reuse a working set of data across parallel operations (such as machine learning algorithms). To optimize for these types of workloads, Spark introduces the concept of in-memory cluster computing, where datasets can be cached in memory to reduce their latency of access.

Spark also introduces an abstraction called *resilient distributed datasets* (RDDs). An RDD is a read-only collection of objects distributed across a set of nodes. These collections are resilient, because they can be rebuilt if a portion of the dataset is lost. The process of rebuilding a portion of the dataset relies on a fault-tolerance mechanism that maintains *lineage* (or information that allows the portion of the dataset to be re-created based on the process from which the data was derived). An RDD is represented as a Scala object and can be created from a file; as a parallelized slice (spread across nodes); as a transformation of another RDD; and finally through changing the persistence of an existing RDD, such as requesting that it be cached in memory.

Applications in Spark are called *drivers*, and these drivers implement the operations performed either on a single node or in parallel across a set of nodes. Like Hadoop, Spark supports a single-node cluster or a multi-node cluster. For multi-node operation, Spark relies on the Mesos cluster manager. Mesos provides an efficient platform for resource sharing and isolation for distributed applications (see Figure 1). This setup allows Spark to coexist with Hadoop in a single shared pool of nodes.

Figure 1. Spark relies on the Mesos cluster manager for resource sharing and isolation.



Although Hadoop captures the most attention for distributed data analytics, there are alternatives that provide some interesting advantages to the typical Hadoop platform. Spark is a scalable data analytics platform that incorporates primitives for in-memory computing and therefore exercises some performance advantages over Hadoop's cluster storage approach. Spark is implemented in and exploits the Scala language, which provides a unique environment for data processing. Get to know the Spark approach for cluster computing and its differences from Hadoop.

VIII. INTRODUCTION SCALA

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application.

Many existing companies who depend on Java for business critical applications are turning to Scala to boost their development productivity, applications scalability and overall reliability.

For example, at Twitter, the social networking service, Robey Pointer moved their core message queue from Ruby to Scala. This change was driven by the company's need to reliably scale their operation to meet fast growing Tweet rates, already reaching 5000 per minute during the Obama Inauguration. Robeys thinking behind the Twitter Kestrel project is explained in the developers live journal. His concise 1500 lines of Scala code can be seen as he has generously made them available as an open source project.

IX. FEATURES OF SCALA

9.1 Scala is object-oriented

Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits. Classes are extended by subclassing and a flexible mixin-based composition mechanism as a clean replacement for multiple inheritance.

9.2 Scala is functional

Scala is also a functional language in the sense that every function is a value. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages.

Furthermore, Scala's notion of pattern matching naturally extends to the processing of XML data with the help of right-ignoring sequence patterns. In this context, sequence comprehensions are useful for formulating queries. These features make Scala ideal for developing applications like web services.

9.3 Scala is statically typed

Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports:

- generic classes,
- variance annotations,
- upper and lower type bounds,
- inner classes and abstract types as object members,
- compound types,
- explicitly typed self references,
- views, and
- polymorphic methods.

A local type inference mechanism takes care that the user is not required to annotate the program with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

Scala is extensible

In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries:

- any method may be used as an infix or postfix operator, and
- closures are constructed automatically depending on the expected type (target typing).

A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

Related Work

Distributed Shared Memory: Spark's resilient distributed datasets can be viewed as an abstraction for distributed shared memory (DSM), which has been studied extensively [19]. RDDs differ from DSM interfaces in two ways. First, RDDs provide a much more restricted programming model, but one that lets datasets be rebuilt efficiently if cluster nodes fail. While some DSM systems achieve fault tolerance through check pointing, Spark reconstructs lost partitions of RDDs using lineage information captured in the RDD objects. This means that only the lost partitions need to be recomputed, and that they can be recomputed in parallel on different nodes, without requiring the program to revert to a checkpoint. In addition, there is no overhead if no nodes fail. Second, RDDs push computation to the data as in MapReduce, rather than letting arbitrary nodes access a global address space. Other systems have also restricted the DSM programming model to improve performance, reliability and programmability.

The need to extend MapReduce to support iterative jobs was also recognized by Twister, a MapReduce framework that allows long-lived map tasks to keep static data in memory between jobs. However, Twister does not currently implement fault tolerance. Spark's abstraction of resilient distributed datasets is both fault-tolerant and more general than iterative MapReduce. A Spark program can define multiple RDDs and alternate between running operations on them, whereas a Twister program has only one map function and one reduce function. This also makes Spark useful for interactive data analysis, where a user can define several datasets and then query them. Spark's broadcast variables provide a similar facility to Hadoop's distributed cache [2], which can disseminate a file to all nodes running a particular job. However, broadcast variables can be reused across parallel operations.

X. Discussion and Future Work

Spark provides three simple data abstractions for programming clusters: resilient distributed datasets (RDDs), and two restricted types of shared variables: broadcast variables and accumulators. While these abstractions are limited, we have found that they are powerful enough to express several applications that pose challenges for existing cluster computing frameworks, including iterative and interactive computations. Furthermore, we believe that the core idea behind RDDs, of a dataset handle that has enough information to (re)construct the dataset from data available in reliable storage, may prove useful in developing other abstractions for programming clusters.

In future work, we plan to focus on four areas:

1. Formally characterize the properties of RDDs and Spark's other abstractions, and their suitability for various classes of applications and workloads.
2. Enhance the RDD abstraction to allow programmers to trade between storage cost and re-construction cost.
3. Define new operations to transform RDDs, including a "shuffle" operation that repartitions an RDD by a given key. Such an operation would allow us to implement group-bys and joins.
4. Provide higher-level interactive interfaces on top of the Spark interpreter, such as SQL and R- shells.

XI. Summary

Although Hadoop captures the most attention for distributed data analytics, there are alternatives that provide some interesting advantages to the typical Hadoop platform. Spark is a scalable data analytics platform that incorporates primitives for in-memory computing and therefore exercises some performance advantages over Hadoop's cluster storage approach. Spark is implemented in and exploits the Scala language, which provides a unique environment for data processing. Get to know the Spark approach for cluster computing and its differences from Hadoop.

References

[1] http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf
[2] http://www.scala-lang.org/node/25
[3] Bader, David; Robert Pennington (June 1996). "Cluster Computing: Applications". Georgia Tech College of Computing. Retrieved 2007-07-13.
[4] Grid and Cluster Computing by Prabhu 2008 8120334280 pages 109-112
[5] William W. Hargrove and Forrest M. Hoffman (1999). "Cluster Computing: Linux Taken to the Extreme". Linux magazine. Retrieved October 18, 2011.
[6] http://en.wikipedia.org/wiki/History_of_computer_clusters
[7] http://www.mosharaf.com/wp-content/uploads/spark-abstract-osdi10.pdf
[8] www.cs.berkeley.edu/~franklin/Papers/hotcloud.pdf
[9] http://www.ibm.com/developerworks/library/os-spark/

Authors

First Author – Tarun Kumawat [CSE],JECRC UDML College of Engineering, Kukas, Jaipur, Rajasthan, India, Email: tarun.kumawat04@gmail.com
Second Author – Pradeep Kumar Sharma [CSE], JECRC UDML College of Engineering, Kukas, Jaipur, Rajasthan, India, Email: Sharma_4203@yahoo.com

Third Author – Deepak Verma [CSE], Jaipur National University, Jagatpura, Jaipur Rajasthan, India, Email: vermad83@gmail.com
Fourth Author – Komal Joshi [CSE], Rajasthan Technical University, Kota, Rajasthan, India, Email: komaljoshi1988@gmail.com
Fifth Author – Vijeta Kumawat [CSE], Jyoti Vidyapeet Woman's University, Jaipur, Rajasthan, India, Email: vijetakumawat@gmail.com