Is 'Distributed' worth it? Benchmarking Apache Spark with Mesos

N. Satra (ns532)

January 13, 2015

Abstract

A lot of research focus lately has been on building bigger distributed systems to handle 'Big Data' problems. This paper examines whether typical problems for web-scale companies really benefits from the parallelism offered by these systems, or can be handled by a single machine without synchronisation overheads. Repeated runs of a movie review sentiment analysis task in Apache Spark were carried out using Apache Mesos and Mesosphere, on Google Compute Engine clusters of various sizes. Only a marginal improvement in run-time was observed on a distributed system as opposed to a single node.

1 Introduction

Research in high performance computing and 'big data' has recently focussed on distributed computing. The reason is three-pronged:

- The rapidly decreasing cost of commodity machines, compared to the slower decline in prices of high performance or supercomputers.
- The availability of cloud environments like Amazon EC2 or Google Compute Engine that let users access large numbers of computers on-demand, without upfront investment.
- The development of frameworks and tools that provide easy-to-use idioms for distributed computing and managing such large clusters of machines.

Large corporations like Google or Yahoo are working on petabyte-scale problems which simply can't be handled by single computers [9]. However,

smaller companies and research teams with much more manageable sizes of data have jumped on the bandwagon, using the tools built by the larger companies, without always analysing the performance tradeoffs. It has reached the stage where researchers are suggesting using the same MapReduce idiom hammer on all problems, whether they are nails or not [7].

In this paper, we set out to better understand how one of these systems, Apache Spark, performs, and how it compares to a non-distributed implementation, for typical machine learning tasks.

2 Frameworks used

2.1 Apache Spark

I chose Apache Spark to implement this system because it abstracts away a lot of the implementation required to make a sequential program work in a distributed setting. Instead, you use functional programming paradigms where possible, as in the code sample below, and the Spark implementation is automatically able to parallelize this appropriately. Hence, when I employ Scala's map or filter idioms for example, the calculations are automatically distributed.

```
val modelAccuracies = folds.map { case (training, test) =>
val model = NaiveBayes.train(training, lambda = 1.0)
val predictionAndLabel = test.map(
        p => (model.predict(p.features), p.label)
)
1.0 * predictionAndLabel.filter(
        x => x._1 == x._2
        ).count() / test.count()
}
```

The core innovation that makes this possible is Spark's Resilient Distributed Datasets [12]. RDDs are read-only collections of objects that are calculated lazily when required, allowing the sequence of transformations and actions applied to them till that point to be optimized together. They also keep track of this lineage, allowing for recalculation in case of faults, rather than expensive storage of snapshots. RDDs are also automatically partitioned.

Behind the scenes, Spark handles the distribution of data and tasks using a Spark Context, which records configuration information on the various slaves in the cluster. This is contained in the Driver Program, which manages the entire running of the code, including synchronization of sub-tasks.

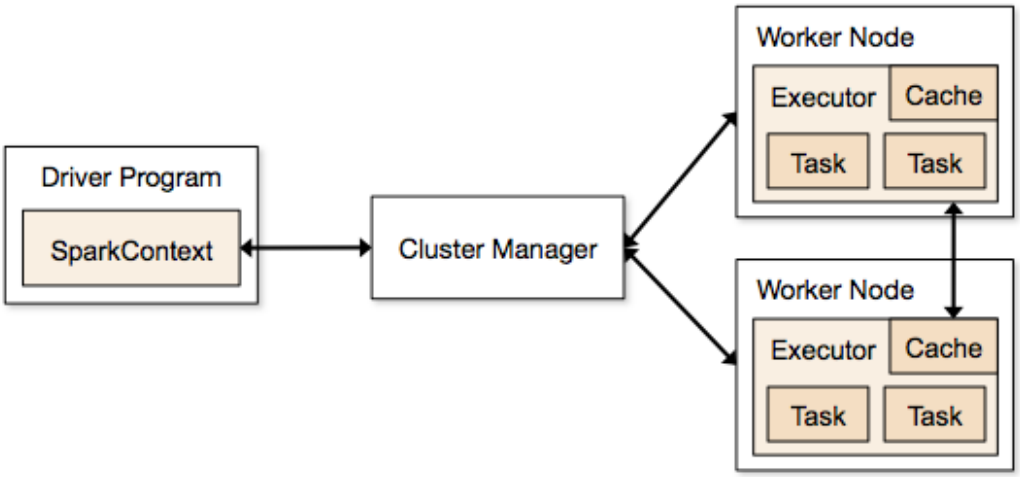

Figure 1: Central to Spark's architecture is the Driver Program, which distributes the work to the worker nodes as scheduled in the Spark Context, and monitors progress

Spark contains the mllib library, which provides optimized-for-spark implementations of machine learning functionality. This was a natural choice for implementing the machine learning sections of the task, rather than hand-coding them, since it allowed quick swapping in and out of various types of classifiers. It is still incomplete, not containing things like k-fold validation. These improvements are expected in the new ml library newer versions of Spark will ship with.

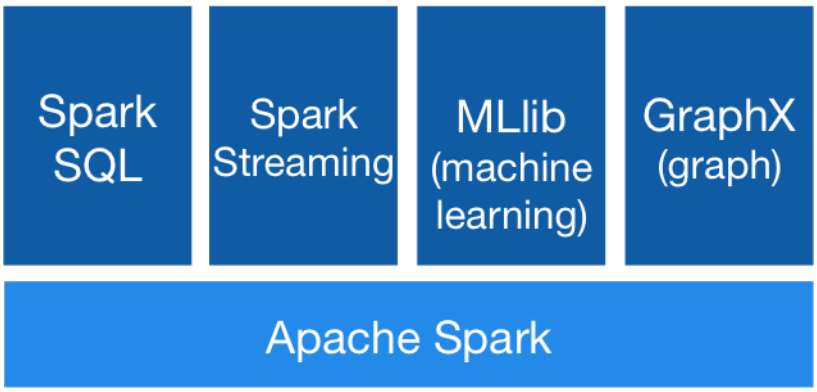

Figure 2: Spark is general purpose, but ships with libraries for specific domains of tasks, such as graph analysis and machine learning

2.2 Apache Mesos

Mesos [3] is a platform that aggregates all the resources (CPU, memory, storage) in a cluster of nodes or data center, and allocates them appropriately to multiple software frameworks the data center might be running, such as Spark and Hadoop as in this case.

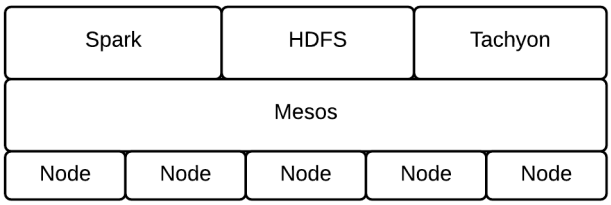

Figure 3: Mesos abstracts away the cluster, and allows multiple frameworks to just work as if they have a pool of resources available, without having to worry about which individual node is available

A Mesos cluster consists of a master daemon (with optional standby masters that were deemed unnecessary for this project), and slave daemons running on each node in the cluster. The master makes offers to each framework of resources it has available in the cluster. The frameworks on their part, provide a scheduler registered with the master that accepts or rejects resource offers, and an executor process that can run the framework's tasks on the slave nodes. Issues such as managing data locality are left up to the framework by allowing to selectively accept resource offers, since Mesos isn't aware of what data the framework is storing where.

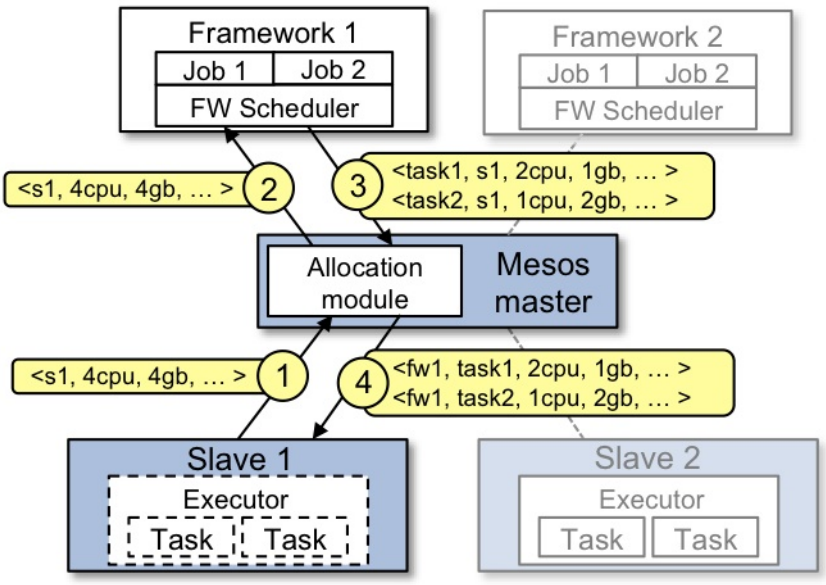

Figure 4: Mesos receives resource availability from slave nodes and offers them to the framework's scheduler, which then tells it which task to run on each of the resources

2.3 Tachyon and Hadoop File System

For storage of the input file and various binaries required to run the task, Hadoop Distributed File System [10] was employed. This allowed us to simply upload the file once and make it available to the entire cluster at the same URI. The alternative would have been to manually upload the files to each of the nodes in the cluster at the exact same path, so that the same executor running on various nodes would be able to access the files locally at the same path.

Intermediate data that Spark generated and used while carrying out the computation, on the other hand, was stored in-memory, as RDDs in Tachyon [6]. This is a storage system that offers higher data throughput compared to HDFS, at nearly memory speeds. Since Spark's fault tolerance is based on regeneration of data rather than persisted snapshots, Tachyon was able to eliminate the overheads of persisting data to disk unnecessarily. Tachyon uses a master-slave architecture, with lineage-based fault tolerance and a checkpointing algorithm for data recovery.

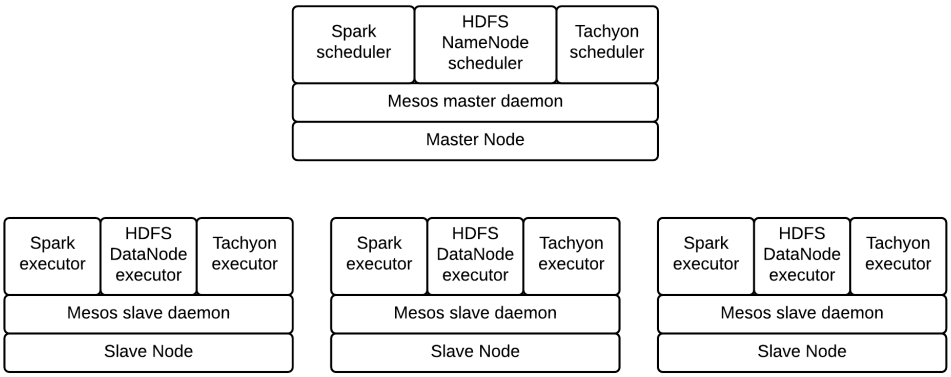

Figure 5: Architecture of the system

2.4 Resulting architecture

As a result of using these systems, we designed a cluster architecture with one master and multiple slaves. The master ran the following:

- Mesos master daemon that makes offers for slave resources according to a resource allocation schedule. It makes offers to the schedulers for Spark, HDFS and Tachyon.
- Spark scheduler that takes the Scala program and determines which of these resources to accept, keeping data locality in mind.
- HDFS NameNode that stores a directory listing in memory of the files available, and answers file requests from any machine in the cluster with the address of the DataNode to find them on.
- Tachyon master daemon that keeps track of lineage information and where each dataset is persisted.
- A ZooKeeper [4] master that provides various cluster services like naming, synchronization and configuration management.

Each of the slaves ran the following:

- Mesos slave daemon that sends the master messages when resources become available
- Spark executor that receives tasks from the master and runs them locally

- HDFS DataNode that uses local hard disk space to provide distributed storage.
- Tachyon slave daemon that uses local memory for temporary storage and hard disk for checkpoint storage.
- A ZooKeeper slave daemon that provides coordination with the master and thus the other slaves.

3 Related Work

The original Spark paper [11] focusses on comparing its performance to Hadoop, which it clearly outperforms. However, there is little work studying how Spark performs on clusters of various sizes, including one multi-core machine. [5] studies how graph analysis problems can perform on a single machine, and compares itself to distributed systems.

4 Method

To understand how a distributed system performs compared to a single-machine system, we ran the same task on both. The task we chose to run handled sentiment analysis on movie review texts using the Naive Bayes classifier implemented in Apache Spark's machine learning library (mllib). Specifically, we'll be training the model on unigram presence in the reviews, so the system approximately associates each word in a review with the likelihood that it would appear in a positive or negative review.

4.1 The Task

The dataset consists of 2000 movie reviews extracted from the Rotten Tomatoes website and tagged as positive or negative according to their associated rating. This dataset was first introduced in [8], with the text already split into word tokens.

The data was then pre-processed in Weka to generate bag of word feature vectors from the free text. This was done by first generating a vocabulary of all the unigrams in the text, and then creating sparse vectors for each document that indicated whether each attribute was present or not in the document. This vocabulary contained 54,604 words, including proper nouns, numbers and some punctuation symbols, because no stop words or stemmer were used. This pre-processing could have been done within Spark too, but

we decided instead to do the pre-processing once, using tools built for the job, and using the results directly in the Spark application, which could then be run quickly multiple times.

The results of the pre-processing were the input to the Spark application. These took the form of a single file of 2000 lines. Each line contained the class of the review (positive or negative), and the feature vector indicating which words were present in the review (using IDs for each word rather than the word itself as the index).

In the Spark application, I load this file from the HDFS, and parse it into datatypes Spark can understand. It is then run through 3-fold cross validation, wherein the data is divided into three stratified folds(each fold has roughly the same proportion of positive and negative reviews). The classifier is then trained on two folds and tested on the third. The accuracy for each fold is averaged out to get the overall accuracy of this model. I attained an accuracy of 82.7%. We could have tested different types of classifiers here to try and improve the accuracy, but this was a good enough baseline to understand the distributed architecture underlying it.

Code was inserted around the entire Spark application to measure how long it took to train and test the model three times, but exclude the time taken to start up the JVM.

4.2 The Infrastructure

Since our dataset was rather small, we designed our cluster to contain 1 master and 3 slaves, whereas the single-computer control experiment contain just 1 machine. Each machine had 2 CPUs and 7.5 GB of RAM. These were standard Google Compute Engine 'n1-standard-2' instances, located in the 'asia-east1-a' zone.

These instances were spun up using Mesosphere, which set them up with installations of Mesos, HDFS and ZooKeeper. It also configured them to act in a master-slave architecture, and to accept my public key for logging on to them. I uploaded the Spark code (packaged as a jar), Spark libraries and dataset onto the server and then placed them on HDFS. After installing Spark and configuring my application to use the cluster, the experiment was run 5 times using the entire cluster, and 5 times using only the master instance.

4.3 The Results

The running times for the experiment are shown below.

Run Number	Cluster time (ns)	Single Instance time (ns)
1	15400280742	12400530773
2	14990859514	12623000232
3	15461173880	11893738207
4	15529786773	12593034890
5	15145971146	13004830580
Average	1.531×10^{10}	1.25×10^{10}

Table 1: Time taken for 3-fold cross validation of sentiment analysis model

Thus, the cluster and single instance finished the experiment on average in 15.31 and 12.5 seconds.

5 Conclusion

We can see that both systems have execution times on the same order of magnitude, with a speedup of 2.81 seconds or 18%. However, the cluster requires 4 times the number of machines as the single instance setup, and network bandwidth too.

Perhaps, while continuing progress on distributed systems, we should also dedicate some research time towards improving single-computer systems to handle bigger loads. GraphChi [5], for example, was able to optimise single-computer hard-disk-based processing of large graphs. It attained run-times on graphs with 42 million vertices and 1.5 billion edges that were on the same order of magnitude (760 seconds) as Spark (487 seconds), using two orders of magnitude fewer CPUs (2 vs 100).

Besides the performance issues, there are also other factors affecting the choice of architecture. Doing distributed computation is hard. Suddenly, one needs to worry about network latency, network jitter on cloud computers, concurrency and partitioning. The developer has to learn a lot more skills, such as managing a cluster and debugging distributed systems. Pricing becomes more complicated, as one has to now predict things like network usage [2]. Factors that were important for performance on a single computer, such as data locality, also have a much bigger impact on performance and costs [2] when the data is on another machine rather than on another cache line.

However, there are pros related to the deployment of cluster computing in enterprises too, as outlined in [1]. Compute capacity can be gradually scaled out as budget increases, rather than scaling up as a step function when buying big, expensive machines. Failure is easier to deal with, since

distributed systems can recover much more quickly from a single CPU failing than single-machine systems, and commodity components are cheaper to replace. Compute capacity can also be distributed much more flexibly between multiple tasks and frameworks.

Ultimately, we can only conclude that while distributed computing has its advantages, it should be deployed in appropriate situations rather than being considered the silver bullet for all problems.

6 Evaluation and Future Work

The chief issue with this experiment was that the dataset was simply too small and the computation too quick, even with 54000 features across 2000 documents. It was chosen because sentiment analysis is a common machine learning task and hence has been benchmarked on numerous systems. I am now running similar experiments with collaborative filtering tasks to do product recommendations, with much bigger datasets, to get a more accurate view of how distributed systems help performance. Such datasets might also allow experimentation with various sizes of clusters, and how they perform at a range of dataset sizes.

I am also carrying out similar timing on other machine learning tasks that may be more computation-intensive or data-intensive, since the aspect that dominates the time requirement depends on the nature of the task. Hence, I am studying matrix operations (which require a lot of data exchange) and logistic regression (which require very little data but quite a bit of computation).

A bulk of the time spent on this project was spent getting the Mesos cluster set up, and interfacing correctly with Spark, rather than programming the actual task. This is in line with the problems described with distributed computing - developers who aren't necessarily systems administrators need to learn how to set up and manage clusters. This was partially due to missing or outdated documentation on Spark, Mesos and Mesosphere. Hence, I might structure my personal notes well and contribute them back to the open source projects as up-to-date documentation on how to get Spark working with Mesosphere.

References

[1] Mark Baker and Rajkumar Buyya. Cluster computing: the commodity supercomputer. *Software: Practice and Experience*, 29

(6):551–576, 1999. ISSN 1097-024X. doi: 10.1002/(SICI)1097-024X(199905)29:6¡551::AID-SPE248¿3.0.CO;2-C. URL http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1097-024X(199905)29:6<551::

[2] Jim Gray. Distributed computing economics. *Queue*, 6(3):63–68, May 2008. ISSN 1542-7730. doi: 10.1145/1394127.1394131. URL http://doi.acm.org/10.1145/1394127.1394131.

[3] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1972457.1972488.

[4] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[5] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387884.

[6] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670985. URL http://doi.acm.org/10.1145/2670979.2670985.

[7] Jimmy Lin. Mapreduce is good enough?if all you have is a hammer, throw away everything that's not a nail! *Big Data*, 1(1):28–37, November 2012. ISSN 2167-6461. doi: 10.1089/big.2012.1501. URL http://online.liebertpub.com/doi/abs/10.1089/big.2012.1501.

[8] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the ACL*, 2004.

[9] Jawwad Shamsi, Muhammad Ali Khojaye, and Mohammad Ali Qasmi. Data-intensive cloud computing: Requirements, expectations, challenges, and solutions. *Journal of Grid Computing*, 11(2):281–310, June 2013. ISSN 1570-7873, 1572-9184. doi: 10.1007/s10723-013-9255-6. URL http://link.springer.com/article/10.1007/s10723-013-9255-6.

[10] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010. doi: 10.1109/MSST.2010.5496972.

[11] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

[12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zah