StratOS: A Big Data Framework for Scientific Computing

Nathaniel R. Stickley[1,*], Miguel A. Aragon-Calvo[1]

Abstract

We introduce StratOS, a Big Data platform for general computing that allows a datacenter to be treated as a single computer. With StratOS, the process of writing a massively parallel program for a datacenter is no more complicated than writing a Python script for a desktop computer. Users can run pre-existing analysis software on data distributed over thousands of machines with just a few keystrokes. This greatly reduces the time required to develop distributed data analysis pipelines. The platform is built upon industry-standard, open-source Big Data technologies, from which it inherits fast data throughput and fault tolerance. StratOS enhances these technologies by adding an intuitive user interface, automated task monitoring, and other usability features.

Keywords: Computing platforms, Cloud computing, Monitors

1. Introduction

In the last decade we have seen an exponential increase in the volume of data generated from sensors, experiments and simulations. Disciplines that once were data starved are now being flooded with terabytes, and soon petabytes of data. We are entering a new, data-driven, era of science in which discoveries will be made by analyzing data that is not only massive in size but heterogeneous and, in some cases, highly interconnected. The challenge is how to extract meaningful patterns from the sea of information. We now have access to massive datasets, yet we do not have standard methods to efficiently store, handle, and analyze such data. Different research groups, often facing common data analysis and management challenges, end up developing custom data pipelines, resulting in replication of efforts, wasted resources, and incompatibility among projects that might otherwise complement one another.

In industry, the need to perform large-scale data analysis has resulted in the development and adoption of Big Data frameworks, such as Apache Hadoop and Apache Spark. Largely driven by Internet and finance companies, these tools are most easily applied to Web and business data. While industry has greatly benefited from standardized Big Data technologies, we have not seen the same level of adoption in the astronomical community, partially because adapting existing Big Data tools for scientific data analysis is oftentimes not straightforward. Analogous solutions, designed specifically for large-scale scientific data analysis, management, and storage have not yet emerged, despite the increasing need. Projects such as the Sloan Digital Sky Survey (York et al., 2000) and the Bolshoi simulation (Klypin et al., 2011) have generated tens of terabytes of data, most of which remain largely inaccessible for full-scale analysis due to data throughput and storage limitations. Ongoing and future projects like the National Virtual Observatory and the Large Synoptic Survey Telescope (Ivezic et al., 2008) will produce petabytes of data. We need to develop efficient and scalable data analysis and management pipelines in order to face the coming challenges in the era of data-driven science.

1.1. I/O bottlenecks in conventional supercomputers

Most large-scale scientific data analysis is currently performed on conventional supercomputer architectures in which computing nodes are physically decoupled from data storage. Such architectures are well-suited for compute-intensive applications where CPU, memory, and inter-node communication speed are the limiting factors. However, they perform poorly when applied to data-intensive problems that require high data throughput, such as large-scale signal processing, or analyzing large ensembles of data. As the size of datasets approaches the petabyte scale, traditional supercomputing architectures quickly become I/O-bound, which limits their usefulness for data analysis.

1.2. The Big Data approach

A distributed computing architecture, known as the datacenter (Hoelzle and Barroso, 2009), has emerged as the natural architecture for analyzing the enormous quantities of data that have recently become available. Like most supercomputers, datacenters consist of many machines connected via a network. Unlike supercomputers, however, the data in datacenters is stored locally on the computing nodes, rather than in external storage servers. Each

*Corresponding author
Email addresses: nstic001@ucr.edu (Nathaniel R. Stickley), maragon@ucr.edu (Miguel A. Aragon-Calvo)
[1]Department of Physics and Astronomy, University of California, Riverside

node in a datacenter primarily analyzes data stored locally, rather than needing to first transfer data over a network before beginning the analysis.

Software frameworks designed for efficiently performing data analysis using the datacenter architecture are commonly referred to as "Big Data" frameworks. Such frameworks share three key features:

- Data placement awareness. Data analysis is performed preferentially on locally stored data, greatly reducing network traffic, increasing I/O throughput.
- Fault tolerance. By enforcing data replication over the datacenter, Big Data frameworks increase resistance to failure. Hardware malfunctions do not result in data and tasks loss. If a node fails its data is recovered from other nodes and tasks are rescheduled and performed on healthy nodes.
- Datacenter abstraction. Most Big Data frameworks provide some degree of abstraction over the datacenter, so typical users need not be aware of the details of the underlying architecture.

Big Data frameworks are designed to make use of parallel programming models that exploit the datacenter architecture. The prime example of this is the MapReduce model (Dean and Ghemawat, 2008), which allows the analysis of massive distributed datasets by automatically dividing the analysis into many independent tasks that run in parallel on the nodes of the datacenter (the map step). The results of the map step are then aggregated into a final result (the reduce step). For a more detailed discussion of popular Big Data tools, refer to Appendix B.

Most existing Big Data frameworks are primarily intended for text-based data. Analysis software is written in (or ported to) Java, the *de facto* standard language for enterprise and Big Data applications, or adapted to work with solutions such as Hadoop Streaming. In addition to this, reading and writing custom binary file formats, common in scientific research, is not straightforward; custom code must be written to handle each file format. The complexity involved in using Hadoop to analyze and manipulate binary files is illustrated in the astronomical image processing work of Wiley et al. (2011) and Chang et al. (2011).

2. Simplified datacenter computing: StratOS

In the remainder of the paper, we describe StratOS[1], a Big Data platform designed to abstract a datacenter as a regular desktop computer, as illustrated in Figure 1. StratOS allows users to launch any pre-existing command-line driven program or script on a datacenter; programs do not need to be made aware of StratOS in order to work. A

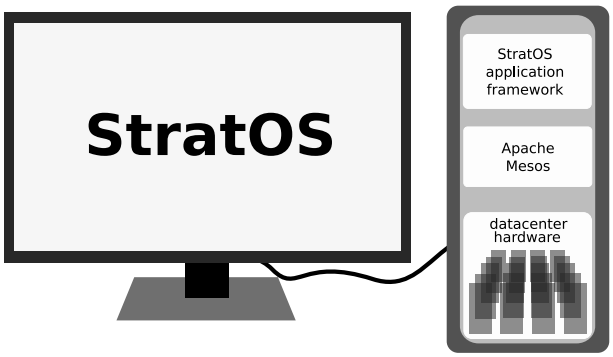

Figure 1: The StratOS platform treats datacenter hardware the same way a desktop operating system treats local CPUs, disks, and RAM. The user is presented with an interface that hides the details of the hardware.

Python module is provided for scripting and interactively launching distributed data analysis tasks, and a C++ library is available for more advanced applications. Static data analysis pipelines can be built with just a few lines of code and the process of building more flexible analysis systems, similar to Astro-WISE (Valentijn et al., 2007; Vriend et al., 2012), is greatly simplified due to the level of abstraction that StratOS provides.

StratOS is built using industry-standard open source tools, which allows it to benefit from the efforts of a large community of developers. This also means that system builders and administrators who are already familiar with existing Big Data tools do not need to learn anything new in order to install and maintain StratOS.

The StratOS platform consists of three main components: (i) a distributed operating system kernel, (ii) a locality-aware distributed file system, and (iii) the StratOS application framework. The kernel and file system currently used by StratOS are open source projects, managed by the Apache Software Foundation: Apache Mesos (Hindman et al., 2011) and the Hadoop Distributed File System (HDFS) (Shvachko et al., 2010). The HDFS is mounted on each node of the system, using FUSE-DFS, so that applications can access the HDFS as though it is a local file system. The application framework, which is the platform's defining component, provides a simple, intuitive interface for launching and managing programs on the distributed system. A schematic overview of the platform architecture is shown in Figure 2.

StratOS users only need to be familiar with the basic usage of a Python interpreter. No experience with distributed computing or multithreading is necessary. The framework automatically schedules tasks so that, whenever possible, execution occurs on CPUs that have local access to the data being used. This minimizes network traffic and maximizes the rate at which data can be read. Tasks that are lost due to hardware failures are automatically rescheduled on other machines. The StratOS application framework also provides a means of automatically monitoring each task that runs on the distributed system;

[1] `http://bitbucket.org/stratos-project/`

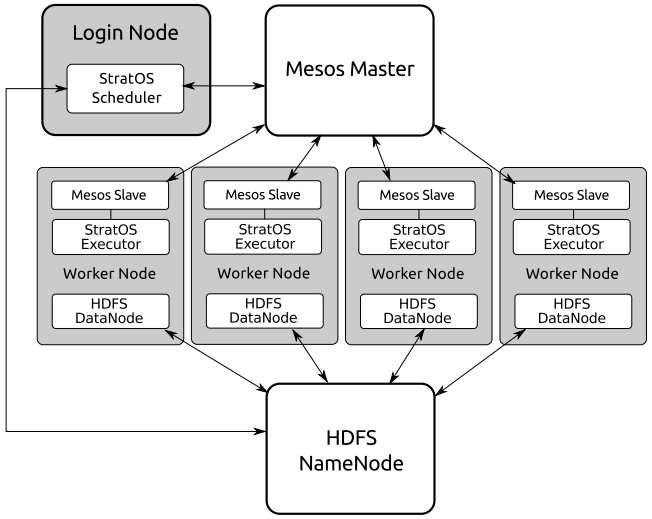

Figure 2: An overview of the StratOS architecture. Each worker node runs a Mesos Slave daemon, an HDFS DataNode daemon, and a StratOS executor, which is part of the StratOS application framework. The Mesos master daemon and the HDFS NameNode daemon each run on a dedicated node. The scheduler of the StratOS application framework runs on a login node.

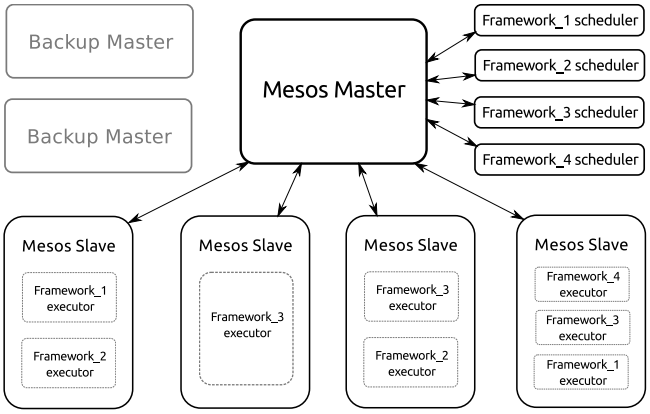

Figure 3: Mesos architecture overview. The Mesos slave daemon runs on each worker node and communicates directly with framework executors. Note that multiple framework executors can share a single slave node. Each slave daemon communicates with the Mesos master node via a network. Framework schedulers communicate with the Mesos master. Optionally, the system can be configured so that backup master nodes can take over for the active master node, in the case of hardware failure.

a user-defined task monitoring script can be executed at regular intervals to examine the detailed behavior of each task.

3. Base technologies

3.1. Distributed resource management with Apache Mesos

In the StratOS platform, the distributed resource manager is crucial because it is responsible for efficiently allocating computational resources. Among the available options, Mesos provided the best match to our requirements because of its scalability and granularity.

Mesos is a distributed operating system kernel that manages resources on a cluster of networked computing/storage nodes (e.g., a datacenter). Applications developed to run on top of Mesos, called *frameworks*, are offered available resources by Mesos in order to perform data analysis tasks. The framework is then responsible for selecting tasks to run on the resources offered by the kernel. A framework may also decline resource offers that are not desired, in which case, the resources may be offered to another framework. Mesos scales to tens of thousands of nodes and allocates resources to frameworks with a high degree of granularity; the smallest allocatable resource unit is a single CPU thread. This makes it possible for tasks from multiple frameworks to share a single machine, allowing the efficient use of the datacenter, since there is no need to partition the datacenter into application-specific sections.

3.1.1. Mesos architecture

Mesos consists of two components: a *master daemon* and a *slave daemon*. The master runs on the head node of the system and an instance of the slave runs on each worker node. The master is responsible for global resource management and system monitoring, while the slaves are responsible for managing resources on individual nodes.

Each Mesos framework consists of two components, corresponding to the master-slave pair: a scheduler and an executor. The scheduler handles resource offers and other information provided by the master, such as task status update messages. The executors are responsible for performing the tasks assigned to them by the scheduler and providing their local slave daemon with task status updates. Refer to Figure 3 for a schematic overview of the Mesos architecture. For a discussion of Mesos' fault tolerance features, refer to Appendix A.

3.2. Data locality awareness with HDFS

StratOS stores data across the datacenter using the HDFS, which is a robust, distributed file system, inspired by the Google File System architecture (Ghemawat et al., 2003). Files stored in HDFS are broken into blocks, which are then replicated on multiple machines, so that the failure of any individual hard drive or host machine does not result in data loss.

Suppose a user specifies that the block size for a particular file is 128 MB and that the block replication factor for the file is three. If the file contains 500 MB of data, it would be broken into four blocks and the HDFS would contain three copies of each block. The file could be distributed across as many as 12 separate machines. When the file is later accessed, as many as 12 machines could potentially send data to the machine that is accessing the file. If StratOS is used to launch a program that reads this particular file, the program would automatically be launched on the machine that contains the largest fraction of the file's data.

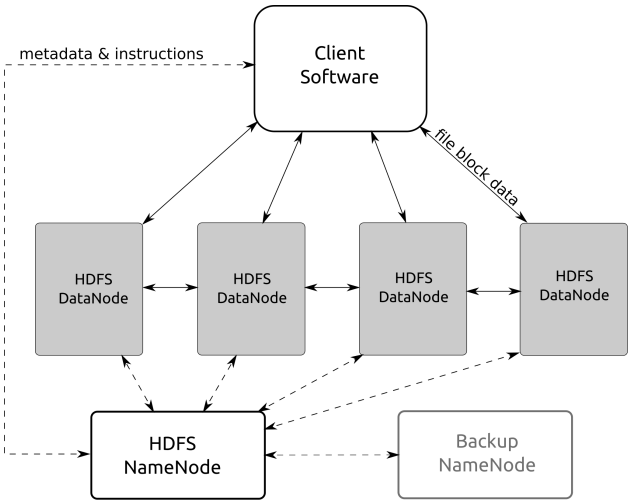

Figure 4: HDFS architecture overview. Dashed lines indicate the transfer of metadata and instructions, while solid lines indicate block data transfer. Client software sends instructions to, and obtains metadata from, the NameNode. Each DataNode receives instructions and metadata from the NameNode. File block data is transferred among DataNodes and between client software and DataNodes. Optionally, a backup NameNode can be configured so that the system can survive a NameNode failure.

3.2.1. HDFS architecture

HDFS consists of two components: a NameNode daemon and a DataNode daemon. These are, in many ways, analogous to Mesos' master and slave daemons. The NameNode stores the directory tree of the file system, tracks the physical location of each block, and maps filenames to file blocks. It also ensures that each block is replicated as many times as specified by the user. Each DataNode daemon manages blocks of data, which are stored on the local file system of the host machine. DataNodes are responsible for performing actions requested by the NameNode. Data can be transferred between DataNodes (in order to create copies) and directly between a DataNode and client software, as shown in Figure 4.

To read data from the HDFS, client software first communicates with the NameNode, which provides the identities of the DataNodes containing the blocks of interest. The client can then retrieve the blocks of data directly from the relevant DataNodes. Transferring data to the HDFS proceeds similarly; the client communicates with the NameNode to determine which DataNodes will contain blocks of the file. The data is then transferred directly to the selected DataNode that is nearest to the client, in terms of network distance. The nearest DataNode forwards packets of data to the second-closest DataNode that was selected to contain a replica of the current block. This process continues until the packet has been sent to all of the selected DataNodes.

When data is added to the HDFS from an external source (i.e., from a computer that is not running the DataNode daemon), the new blocks of data are distributed across the nodes of the system uniformly. Reading files that are larger than the HDFS block size usually requires a portion of the file to be transferred over the network. On the other hand, when data is added by a program running on an HDFS node, one copy of each data block is stored locally on that node. This data can subsequently be read entirely from the local disk.

We note that the HDFS is not a fully POSIX-compliant file system. For instance, once data has been written to a file, it cannot be modified. Files can, however, be appended with new data and they can be deleted and replaced with a new files with the same names as the old files.

3.2.2. Standard file access with FUSE-DFS

The HDFS must be accessed using a program that is aware of the HDFS interface. In order to allow pre-existing software to access the HDFS without being modified, we use the FUSE-DFS utility, which is part of the Hadoop software project. FUSE-DFS is used to mount HDFS as a local file system on each node of the cluster. Any software can then access the HDFS as though it were an ordinary directory on the local file system. Thus, the user does not need to modify their software in order to take advantage of the features offered by StratOS. However, FUSE-DFS imposes constraints on the usage patterns. Most importantly, there is no support for appending data to a file. The user is not presented with an error message when trying to append data to a file. Thus, the user must be careful to not confuse the mounted HDFS with a regular file system.

4. The StratOS application framework

The StratOS application framework is a custom Mesos framework that allows users to launch arbitrary software on a cluster. The user simply specifies the commands that they wish to execute on the cluster and the framework takes care of the details. A Python module and C++ library are available so that the framework can be used interactively, via a script, or via a C++ application. The framework inherits the fault tolerance offered by Mesos and HDFS and adds an extra layer of fault tolerance, in the form of task monitoring scripts.

4.1. The StratOS scheduler

The primary responsibility of the scheduler is to assign commands to appropriate host machines, based on HDFS data placement. Each command assigned to the StratOS scheduler is first inspected for the presence of HDFS filenames. The scheduler then identifies which host machines contain data used by each command. When the Mesos master offers resources to the scheduler, the scheduler checks to see which not-yet-launched tasks involve data on each host listed in the offer. Tasks are then assigned to the appropriate hosts. If the scheduler is offered resources on a host that does not contain any blocks of relevant data, the offer is declined. Resource offers involving non-optimal

hosts are only accepted if the scheduler has requested resources on a more suitable host unsuccessfully. This can happen if the appropriate hosts are being used by another framework.

Whenever the status of a task changes, the Mesos master informs the scheduler of the change by sending a task status update message. The scheduler then makes a note of the change and takes appropriate actions. For instance, if a task status message indicates that a task has been lost, the scheduler assigns the task to a new host. If the status message indicates that a task has completed, the content of the message is parsed for extra details. For instance, the message may contain results of a computation performed by the task. The task status message may also indicate that a particular task was killed by the executor. In this case, the message may contain a list of new commands that should be launched on the cluster to replace the task that was killed. Refer to Figure 5 for a graphical summary of the communication within the framework.

4.2. The StratOS executor

The executor is responsible for launching individual tasks on its host machine. It also sends status update messages to the Mesos slave daemon, which then forwards the messages to the master. Like a Unix shell, the executor allows the standard output and error streams of each child task to be redirected. For instance, the standard output can be saved to a file or stored in a status update message. Note that the latter option is only practical for tasks that send a small amount of output to the standard output stream. This is because status update messages are transferred to the master; sending large amounts of data to the master would result in a communication bottleneck.

4.3. Task monitoring

The executor can also execute a user-defined task-monitoring script at regular intervals. The monitoring script is provided with the content of the standard output and error streams and the process identification number of the task that it is monitoring. If the script detects that the task is behaving in an undesirable way, it can instruct the executor to terminate the task. Instructions for re-starting terminated tasks can also be provided by defining a re-launch script. For example, if the user knows that the phrase "using defaults instead" in a particular application's standard error stream indicates that a non-fatal error has occurred, the user could write a script which searches the standard error stream for that particular phrase. When this phrase is encountered, the script can instruct the framework to terminate the task. If the user has defined a re-launch script, the executor then runs this script immediately after terminating the task. The re-launch script can be used to construct one or more new commands, which are then sent to the scheduler. For instance, if the "using defaults" message depends on the input parameters of the code of interest, the re-launch script could be designed to slightly alter the input parameters of the task that was terminated.

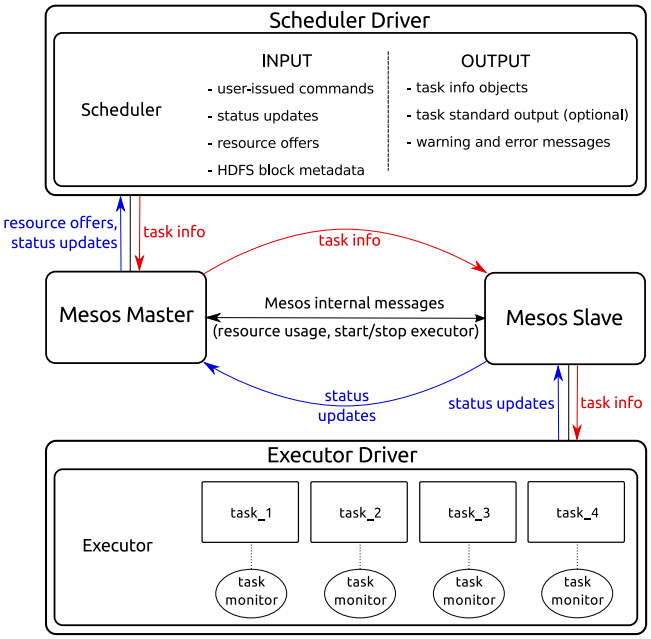

Figure 5: An overview of the StratOS application framework architecture. The Mesos Master communicates with the Mesos Slave daemon, running on each worker node. The StratOS scheduler interacts with the Mesos Master daemon, using the Mesos scheduler driver. Similarly, each StratOS executor interacts with its local Mesos Slave, using the Mesos executor driver. The executors run tasks assigned by the scheduler and send status updates to the scheduler by way of the Mesos Slave and Master daemons. Optionally, the executor can attach a task monitor to each task.

5. StratOS use cases

In this section, we present three illustrative cases of the ability of StratOS to run massive analysis on distributed data using a few commands. All examples make use of the StratOS Python interface.

Example I: Batch processing of images

The most basic way to use StratOS is as a distributed batch processor. Batch processing is ideal for any embarrassingly parallel pipeline step that involves analyzing large datasets distributed on a datacenter. Examples of such applications include large-scale image or video processing, analysis of simulation data, and exploration of parameter space in an ensemble dataset.

Suppose a large number of images, stored in a datacenter, need to be processed using a program called `img_proc`, which requires two arguments: an output directory and an input filename. The `img_proc` program processes its input image, saves the resulting image to the specified output directory, and writes statistics about the operation to the standard output stream. In order to use `img_proc` with StratOS to process a directory full of images, the

user places the `img_proc` program in the HDFS file system, opens a Python interpreter, and types the following commands:

```
from stratos import Processor

job = Processor()

job.glob_batch("/dfs/img_proc /dfs/out/ %f%",
               "/dfs/images/*")

results = job.get_results()
```

The first statement loads the StratOS `Processor` class into the current namespace. This class provides an interface to the StratOS scheduler. Since the import statement is always present, it will be omitted from subsequent examples.

The second statement creates a `Processor` object, named `job`, using default parameters to set up a batch job. On the third line, the `Processor`'s `glob_batch()` method is used to construct a list of commands, which are then submitted to the StratOS scheduler. The second argument of `glob_batch()` is a filename pattern, containing one or more wildcard characters (asterisks). All files within the `/dfs/images/` directory will be matched. The first argument, which specifies the command to be launched, contains a filename placeholder, `%f%`. Suppose the directory, `/dfs/images/` contains files named `img_00000`, `img_00001`, `img_00002`, ..., `img_50000`. The `glob_batch()` method would submit the following commands to the scheduler:

```
/dfs/img_proc /dfs/out/ img_00000
/dfs/img_proc /dfs/out/ img_00001
/dfs/img_proc /dfs/out/ img_00002
                ⋮
/dfs/img_proc /dfs/out/ img_50000
```

By default, the standard output of each task is sent to the scheduler and is made accessible using `get_results()`. In this example, `results` is a Python list containing the standard output of each task.

Example II: Interactive data analysis

The `glob_batch()` method, used in Example I, causes the current thread to be blocked; the user cannot work interactively with the batch job until all tasks have completed. In order to enable interactive data analysis and task management, a streaming mode of operation is available. In this mode, the user can inspect the status of each task, retrieve the output of completed tasks, add new tasks to the scheduler, and cancel specific tasks before the entire job has finished. This type of interactive usage is demonstrated in Example II (Figure 6).

Example II begins by providing the `Processor`'s constructor with a non-default parameter, specifying that each task will be allocated four CPU threads. The `glob_stream()` method on the second line works the same way

```
job = Processor(threads_per_task=4)
# add some tasks:
job.glob_stream("/dfs/img_proc2 /dfs/out/ %f%",
                "/dfs/images/*")
# how many tasks have not completed?:
job.count_unfinished_tasks()
# get results from completed tasks:
results = job.get_results()
# add more tasks:
job.glob_stream("/dfs/img_proc2 %f%",
                "/dfs/more_images/*")
# block the thread until all tasks have completed:
job.wait()
# append remaining results to the list:
results += job.get_results()
```

Figure 6: Example II

as the `glob_batch()` method, discussed previously, except that `glob_stream()` does not block the thread. This makes it possible to retrieve a partial list of results, add more tasks to the job, and perform various other operations while tasks are still running on the cluster. The third command requests the number of tasks that have not yet finished executing—a useful indicator of the job's progress. With the fourth command, the standard output of each completed task is stored in a Python list, called `results`; the standard output of the tasks can be examined at this point. The fifth command assigns more tasks to the job. Calling `job.wait()` causes the thread to be blocked until all tasks have completed. Finally, we obtain the remaining results by calling `get_results()` once more. Note that individual results are only returned *once* by `get_results()`. Thus, the second invocation of `get_results()` only returns the output from tasks that completed after the the first invocation of `get_results()`.

Example III: Custom MapReduce implementation

In streaming mode, multiple StratOS schedulers can operate simultaneously and it is possible for the schedulers to interact. In this example, a version of MapReduce is implemented using two interacting schedulers.

Suppose a program called `map` analyzes a single input file and saves the results of its analysis in a new file whose filename is written to the standard output stream. Another program, called `reduce`, reads two files containing the output of the `map` program. It then summarizes the contents of the input files and saves the summary to a file whose name is written to the standard output. These `map` and `reduce` programs are used together in Example III (Figure 7) to obtain a single file which summarizes the contents of a directory full of input files.

The example begins with the creation of two `Processor` objects, named `mapper` and `reducer`, which will be used to run the `map` and `reduce` programs, respectively. The `mapper` tasks are allocated twice as many threads as the `reducer` tasks because the `map` program requires more computational power than the reduce program. We have

```
mapper = Processor(threads_per_task=4, name="mapper")

reducer = Processor(threads_per_task=2, name="reducer")

mapper.glob_stream("/dfs/map %f%", "/dfs/input_data/*")

mapped_files = []

while (mapper.count_unfinished_tasks() or
       reducer.count_unfinished_tasks() or
       len(mapped_files) > 1):
    mapped_files += mapper.get_results()
    if len(mapped_files) >= 2:
        reduction_inputs = [[mapped_files.pop(0)
                             for i in [0,1]]]
        reducer.template_stream("/dfs/reduce %f% %f%",
                                reduction_inputs)
    mapped_files += reducer.get_results()
```

Figure 7: Example III

introduced the `name` parameter in the `Processor` constructor, which allows us to easily distinguish tasks belonging to different jobs. This allows us to easily distinguish between the `mapper` and `reducer` if we need to inspect the Mesos logs.

The `glob_stream()` method is then used to assign tasks to the `mapper`. As tasks from the `mapper` are completed, the results are used to assign new tasks to the `reducer`. Results from the `reducer` are recursively combined until only one output filename remains in the `mapped_files` list.

The `template_stream()` method, used by the `reducer`, constructs commands by substituting instances of the parameter placeholders in its first argument with entries of the Python list in its second argument. Multiple parameters can be specified by using a nested list as the second parameter. Suppose the second argument is the following nested list:

```
[["/dfs/temp/file_1", "/dfs/temp/file_2"],
 ["/dfs/temp/file_3", "/dfs/temp/file_4"],
 ["/dfs/temp/file_5", "/dfs/temp/file_6"]]
```

The `template_stream()` method would submit the following commands to the scheduler:

```
/dfs/reduce /dfs/temp/file_1 /dfs/temp/file_2
/dfs/reduce /dfs/temp/file_3 /dfs/temp/file_4
/dfs/reduce /dfs/temp/file_5 /dfs/temp/file_6
```

Although it is not the most efficient implementation, this example hints at the ease with which MapReduce can be implemented using StratOS. Note that the `mapper` and `reducer` streams are executed simultaneously and, because of the granularity offered by Mesos, tasks belonging to the `mapper` stream can be executed on the same host as tasks from the `reducer` stream.

6. Discussion

StratOS makes it possible to treat a cluster of machines as single machine. In achieving this, it combines the strengths of Big Data tools and classic batch processors. Like a classic batch processor, StratOS can launch arbitrary software a cluster. In contrast, most software used by popular Big Data frameworks must be made aware of the framework in some way. The tasks launched by StratOS can operate on arbitrary data formats with no extra effort from the user, whereas most Big Data tools require extra effort for each specific data format that is used. Unlike classic batch schedulers, StratOS tasks are scheduled in a data locality-aware manner in order to improve data throughput. New tasks can be incrementally added to a batch job, results can be accessed programmatically while the job is running, and multiple batch jobs can be used in a single analysis routine—either in parallel or sequentially. Additionally, StratOS provides an automated way to monitor the behavior of each task and take actions, based upon the observed behavior. This is a feature that no other tool offers.

The StratOS application framework can be used via a Python module or a C++ library, whereas most Big Data frameworks are heavily dependent upon Java. This makes it more accessible to scientists, since scientists are more likely to be familiar with Python or C++, than Java. The Python module also allows the framework to be used interactively.

Since StratOS is based upon industry-standard tools, it benefits from the efforts of a large community of engineers. This also means that the platform is compatible with many existing tools. For instance, Apache Hadoop Mapreduce, Apache Spark, Apache Hama, Apache Storm, TORQUE, and MPICH can all run on the StratOS platform alongside the StratOS application framework because all of these tools are compatible with the Apache Mesos kernel. It is even possible to use the StratOS application framework and Apache Spark together in the same Python script.

StratOS as a cloud service

Cloud computing services allow users to pay for computing resources as needed, rather than building their own cluster. Since the core components of StratOS are already available on several cloud computing services, such as the Amazon Elastic Compute Cloud (EC2), installing StratOS on these services is straightforward.

Future work

Although it is useful to treat HDFS as a local filesystem, as is done in StratOS, the user must be aware of certain complications that can arise. In particular, since FUSE-DFS does not support appending data to files, programs that depend upon the ability to append will not work properly. This can cause errors that are difficult to diagnose. Also, storing a large number of small files (much smaller than the HDFS block size) unnecessarily burdens

the HDFS NameNode. This problem can be alleviated by combining small files into larger files, but this requires extra effort. Dealing with very large files (requiring tens of blocks) is also inefficient because the entire file must be read by each task that uses it. When possible, splitting large files into smaller, independent pieces, can help to resolve this problem. However, this also requires extra effort by the user. For maximal throughput, programs sometimes need to directly access the HDFS using the HDFS library.

When implementing an algorithm that requires the same data to be accessed repeatedly by subsequent tasks, it is beneficial to cache the repeatedly-used data by saving it on a local hard drive or in a RAM disk. This improves performance because there is no need to repeatedly read the same files from the HDFS. In its current form, StratOS does not provide an automatic method of caching data. The user can launch tasks which store data in a specific location on the local machine, however the user is also responsible for deleting such files when they are no longer needed. Furthermore, there is no easy way to ensure that subsequent tasks are launched on machines containing cached data.

StratOS offers no automatic way to facilitate inter-task communication. In order for tasks to communicate, the user must either save files to the HDFS or include network communication capabilities in the task software. This requires extra thought and effort on the part of the user.

We plan to make the StratOS application framework even easier to use by providing automated methods for handling additional usage scenarios. We will also work to address the weaknesses of the StratOS application framework, discussed above. In particular, we plan to add an easy, automated method for persistently storing certain data on local disks (including RAM disks) for use by subsequent tasks. Cached data will automatically be cleaned up so that manual intervention is unnecessary. The StratOS application framework will rely less heavily upon FUSE-DFS and there will be facilities for automatically improving the performance of tasks that use large numbers of small files as well as some tasks that make use of very large files. We will also eventually provide an automated means of enabling inter-task communication so that a broader variety of algorithms can be implemented easily.

7. Acknowledgements

This research was funded by a Big Data seed grant from the Vice chancellor for research and development, UC Riverside.

References

Apache Hadoop, URL: http://hadoop.apache.org/. accessed: 2015-2-22.

Apache Hama, URL: http://hama.apache.org/. accessed: 2015-2-22.

Apache Mesos, URL: http://mesos.apache.org/. accessed: Accessed: 2015-2-22.

Apache Spark, URL: http://spark.apache.org/. accessed: 2015-2-22..

Apache Storm, URL: http://storm.apache.org/. accessed: 2015-2-22

Chang, G., Malhotra, S., Wolgast, P., 2011. Leveraging the cloud for robust and efficient lunar image processing. IEEE Aerospace Conference 0, 1–8. doi:http://doi.ieeecomputersociety.org/10.1109/AERO.2011.5747553.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Communications of the ACM 51, 107–113.

EC2, . Amazon Elastic Compute Cloud. URL: http://aws.amazon.com/ec2/. accessed: 2015-2-22.

Ghemawat, S., Gobioff, H., Leung, S.T., 2003. The google file system. SIGOPS Oper. Syst. Rev. 37, 29–43. URL: http://doi.acm.org/10.1145/1165389.945450, doi:10.1145/1165389.945450.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I., 2011. 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, USENIX Association, Berkeley, CA, USA. pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

Hoelzle, U., Barroso, L.A., 2009. The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines. 1st ed., Morgan and Claypool Publishers.

Ivezic, Z., et al., 2008. LSST: from Science Drivers to Reference Design and Anticipated Data Products. ArXiv e-prints arXiv:0805.2366.

Klypin, A.A., Trujillo-Gomez, S., Primack, J., 2011. Dark Matter Halos in the Standard Cosmological Model: Results from the Bolshoi Simulation. ApJ 740, 102. doi:10.1088/0004-637X/740/2/102, arXiv:1002.3660.

MapR, URL: http://www.mapr.com/. accessed: 2015-2-22.

MPICH, URL: http://www.mpich.org/. accessed: 2015-2-22.

National Virtual Observatory, . National Virtual Observatory. URL: http://www.usvao.org/. accessed: 2015-2-22.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The hadoop distributed file system, in: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE Computer Society, Washington, DC, USA. pp. 1–10. URL: http://dx.doi.org/10.1109/MSST.2010.5496972, doi:10.1109/MSST.2010.5496972.

TORQUE, URL: http://www.adaptivecomputing.com/products/open-source/torque/. accessed: 2015-2-22.

Valentijn, E.A., et al., 2007. Astro-WISE: Chaining to the Universe, in: Shaw, R.A., Hill, F., Bell, D.J. (Eds.), Astronomical Data Analysis Software and Systems XVI, p. 491. arXiv:astro-ph/0702189.

Valiant, L.G., 1990. A bridging model for parallel computation. Commun. ACM 33, 103–111. URL: http://doi.acm.org/10.1145/79173.79181, doi:10.1145/79173.79181.

Vriend, W.J., Valentijn, E.A., Belikov, A., Verdoes Kleijn, G.A., 2012. Astro-WISE Information System, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), Astronomical Data Analysis Software and Systems XXI, p. 719. arXiv:1111.6465.

Wiley, K., Connolly, A., Gardner, J., Krughoff, S., Balazinska, M., Howe, B., Kwon, Y., Bu, Y., 2011. Astronomy in the Cloud: Using MapReduce for Image Co-Addition. PASP 123, 366–380. doi:10.1086/658877, arXiv:1010.1015.

York, D.G.,et al., 2000. The Sloan Digital Sky Survey: Technical Summary. The Astronomical Journal 120, 1579–1587. doi:10.1086/301513, arXiv:astro-ph/0006396.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I., 2012. 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), USENIX, San Jose, CA. pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/

technical-sessions/presentation/zaharia.

Appendix A. Fault tolerance

In this appendix, we describe the fault tolerance features of Mesos and HDFS in greater detail.

Mesos

If the Mesos master daemon detects that a slave has become unreachable (for instance, due to hardware failure), it notifies the relevant framework schedulers that the tasks running on the unreachable machine have been lost. The schedulers can then assign the lost tasks to other hosts. Thus, the system seamlessly handles node failures. Additionally, it is possible to configure redundant Mesos masters, so that the system is resilient to master node failure.

HDFS

The HDFS DataNode daemon sends status messages, called "heartbeats," to the NameNode at regular intervals to inform the NameNode that it is still alive and reachable. If the NameNode stops receiving heartbeats from a particular DataNode, that DataNode is assumed to be dead. The NameNode then instructs the remaining DataNodes to make additional copies of the data that was stored on the dead node so that the required replication factor of each block is maintained. Additionally, each time a DataNode reads a block of data, it computes a checksum. If the checksum does not match the original checksum, stored in the block's associated metadata file, the NameNode is informed that the block has been corrupted. A new copy is then created from the uncorrupted copies on other machines. As in the case of the Mesos master daemon, it is possible to configure redundant, backup NameNodes so that the file system remains intact when the machine hosting the primary NameNode experiences a failure.

Appendix B. Summary of existing tools

In this appendix, we briefly summarize the features of several popular tools: Apache Hadoop MapReduce, Apache Hama, Apache Spark, MapR, and TORQUE. We also comment on aspects of these tools that limit their usefulness for large-scale scientific data analysis.

Hadoop MapReduce

The Hadoop MapReduce framework is an implementation of the MapReduce programming model. The user provides a mapper function and a reducer function, typically in the form of Java class methods. The user then provides a set of key-value pairs (for instance, filenames and file contents) for the framework to operate upon. The mapper transforms the initial set of key-value pairs into a second set of key-value pairs. The intermediate key-value pairs are then globally sorted by key and transformed into a third set of key-value pairs by the reducer.

General computing applicability

The mapper and reducer almost always need to be designed with Hadoop in mind. A feature called Hadoop Streaming makes it possible to use pre-existing executable files as the mapper and reducer. However, the executables need to be able to read and write key-value pairs via the standard input and output streams. In many situations, using pre-existing software with Streaming requires the software to be invoked from a script which formats the input and output streams appropriately. In other situations, the mapper and reducer programs have to be modified in order to to work properly.

Handling non-trivial data formats efficiently requires special care. If the data format being used with MapReduce is more complicated than a text file that can be split into single-line records, then a custom file reader must be defined in order for the MapReduce framework to properly read the data. Reading binary files produced by scientific instruments or simulations is even more cumbersome than reading formatted text.

Using Hadoop MapReduce with languages other than Java requires a bit more work, since the framework is primarily intended to be used by Java programmers. In order to use Hadoop MapReduce with languages other than Java, one can use Hadoop Pipes, which makes it possible to write mappers and reducers in C++. The C++ code can be extended to make use of other languages, such as Python.

Hama

Apache Hama is a framework for Big Data analytics which uses the Bulk Synchronous Parallel (BSP) computing model (Valiant, 1990) in which a distributed computation proceeds in a series of super-steps consisting of three stages: (1) concurrent, independent computation on worker nodes, (2) communication among processes running on worker nodes, and finally (3) barrier synchronization.

Individual processes stay alive for multiple super-steps. Thus, data can easily be stored in RAM between steps. This allows Hama to perform very well on iterative computations that repeatedly access the same data. Hama can outperform Hadoop MapReduce by two orders of magnitude on such tasks.

General computing applicability

Like Apache MapReduce, Hama is primarily intended to be used with Java, but it is possible to write programs in C++, using Hapa Pipes. Hama handles data formats in exactly the same way as Hadoop MapReduce. Thus, working with raw scientific data is not straightforward in most cases.

Spark

Apache Spark is framework based upon the concept of Resilient Distributed Datasets (RDDs) (Zaharia et al.,

2012). As the name suggests, RDDs are distributed across a cluster of machines. For added performance, the contents of an RDD can be stored in memory. Their resiliency lies to the fact that only the initial content of the RDDs and transformations performed on them need to be stored in a distributed file system; the memory-resident version of an RDD can be automatically recreated upon node failure by repeating transformations on the initial data.

Compared with Hadoop MapReduce, Spark offers more flexibility. There is no need to use a particular programming model; it is possible to implement MapReduce, BSP, and other models with Spark. Spark is also more interactive. Once an RDD is created, operations can easily be performed on the data by issuing commands from an interpreter. Spark works natively with three programming languages: Scala, Java, and Python.

General computing applicability

Any executable file can be invoked with the RDD `pipe()` transformation, which sends data to the executable via a Unix pipe and then stores the standard output of the executable in an RDD. However, in order to be useful, the program must be able to read data from its input stream and send output data to the standard output stream. Programs that do not behave in this way need to first be modified in order to be compatible with Spark.

Using data that is more complicated than plain text requires the user to define a file format reader. Thus, working directly with raw scientific data formats is not straightforward.

MapR

MapR is a proprietary Big Data platform, based on a distributed file system, called MapR-FS, which is more flexible than HDFS. It can also achieve higher performance than HDFS, in some cases. Unlike HDFS, MapR-FS is POSIX compliant. Notably, file editing is not limited to merely appending data to the end of a file. Many existing frameworks that use HDFS and Hadoop's Yarn cluster manager run on the MapR platform.

General computing applicability

Whereas most other tools and platforms are free, MapR is not. Although there are several price tiers, the full benefits of the MapR-FS are only realized with the most expensive option. Due to licensing, a MapR cluster must contact a licence server in order to start running. Another potential drawback is that disks must be formatted at a low level in order to use MapR-FS. Once a disk is formatted with MapR-FS, it can only be accessed by MapR. Most of the complications inherent in using Hadoop MapReduce and Spark remain problematic with MapR.

TORQUE

TORQUE is a distributed resource manager, designed for submitting and managing batch jobs on a cluster. With TORQUE, it is possible to launch arbitrary programs and operate on arbitrary data. Batch jobs are launched by first writing a batch submission script and then submitting the script to the scheduler. It is possible to monitor the status of a batch job and individual tasks while the job is running.

General computing applicability

Unlike the Big Data tools, discussed above, TORQUE is not aware of data placement. Thus, data throughput depends heavily upon the speed of the network. It is also not straightforward to create data analysis routines consisting of multiple batch jobs. TORQUE is primarily intended for manually launching batch jobs, rather than creating jobs programmatically.