Case Study of Scientific Data Processing on a Cloud Using Hadoop

Chen Zhang[1], Hans De Sterck[2], Ashraf Aboulnaga[1], Haig Djambazian[3], and Rob Sladek[3]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Ontario, N2L 3G1, Canada
[2] Department of Applied Mathematics, University of Waterloo, Ontario, N2L 3G1, Canada
[3] McGill University and Genome Quebec Innovation Centre, Montreal, Quebec, H3A 1A4, Canada

Abstract. With the increasing popularity of cloud computing, Hadoop has become a widely used open source cloud computing framework for large scale data processing. However, few efforts have been made to demonstrate the applicability of Hadoop to various real-world application scenarios in fields other than server side computations such as web indexing, etc. In this paper, we use the Hadoop cloud computing framework to develop a user application that allows processing of scientific data on clouds. A simple extension to Hadoop's MapReduce is described which allows it to handle scientific data processing problems with arbitrary input formats and explicit control over how the input is split. This approach is used to develop a Hadoop-based cloud computing application that processes sequences of microscope images of live cells, and we test its performance. It is discussed how the approach can be generalized to more complicated scientific data processing problems.

1 Introduction

The concept of 'cloud computing' is currently receiving considerable attention, both in the research and commercial arenas. While cloud computing concepts are closely related to the general ideas and goals of grid computing, there are some specific characteristics that make cloud computing promising as a paradigm for transparently scalable distributed computing. In particular, two important properties that many cloud systems share are the following:

1. Clouds provide a homogeneous operating environment (for instance, identical operating system (OS) and libraries on all cloud nodes, possibly via virtualization).
2. Clouds provide full control over dedicated resources (in many cases, the cloud is set up such that the application has full control over exactly the right amount of dedicated resources, and more dedicated resources may be added as the needs of the application grow).

D.J.K. Mewhort et al. (Eds.): HPCS 2009, LNCS 5976, pp. 400–415, 2010.

While these two properties lead to systems that are less general than what is normally considered in the grid computing context, they significantly simplify the technical implementation of cloud computing solutions, possibly to the level where feasible, easily deployable technical solutions can be worked out. The fact that cloud computing solutions, after only a short time, have already become commercially viable would point in that direction. Indeed, the first property above removes the complexity of dealing with versions of application code that can be executed in a large variety of software operating environments, and the second property removes the complexity of dealing with resource discovery, queuing systems, reservations, etc., which are the characteristics of shared environments with heterogeneous resources.

Cloud computing is thus a promising paradigm for transparently scalable distributed computing. It was pioneered in large-scale web application processing: a well-known and highly successful example is Google's web processing and hosting system. Cloud computing is now also being used and explored extensively for enterprise applications. This paper is concerned with the application of cloud computing to large-scale scientific data processing, an area which is relatively unexplored so far.

It is useful to distinguish three software layers in clouds:

1. The OS (possibly virtualized).
2. A cloud computing framework, often including an execution environment, a storage system, and a database-like capacity.
3. A user application built on top of layers 1 and 2.

Google's system is a well-known cloud computing framework, and several commercial cloud computing frameworks have recently appeared on the market, including products from GigaSpaces [19], Elastra [16], etc. A well-known commercial cloud provider is Amazon [3].

In this paper, we focus on cloud computing environments similar to Google's system, which was designed for scalable and reliable processing and hosting of the very large data sets that are provided by the world's largest 'information organization' company. Three major components in Google's web processing system are:

1. MapReduce, a scalable and reliable programming model and execution environment for processing and generating large data sets.
2. Google File System (GFS), a scalable and reliable distributed file system for large data sets.
3. BigTable, a scalable and reliable distributed storage system for sparse structured data.

Google's system is tailored to specific applications. GFS can deal efficiently with large input files that are normally written once and read many times. MapReduce handles large processing jobs that can be parallelized easily: the input normally consists of a very long sequence of atomic input records that can be processed independently, at least in the first phase. Results can then

be collected (reduced) in a second processing phase, with file-based key-value pair communication between the two phases. MapReduce features a simple but expressive programming paradigm, which hides parallelism and fault-tolerance. The large input files can be split automatically into smaller files that are processed on different cloud nodes, normally by splitting files at the boundaries of blocks that are stored on different cloud nodes. These split files are normally distributed over the cloud nodes, and MapReduce attempts to move computation to the nodes where the data records reside. Scalability is obtained by the ability to use more resources as demand increases, and reliability is obtained by fault-tolerance mechanisms based on replication and redundant execution.

In this paper, we use Hadoop [4], which is an open source implementation of a subset of the Google system described above. The three Hadoop components that are analogous to Google's components described above are:

1. Hadoop's MapReduce.
2. Hadoop's Distributed File System (DFS).
3. The HBase storage system for sparse structured data.

Hadoop is used by companies like Yahoo and Facebook, and is also widely used as an instruction tool for cloud computing education.

In this paper, we use the Hadoop cloud computing framework to develop a simple user application that allows processing of scientific data of a certain type on clouds. Our paper is mainly an application paper, exploring the use of Hadoop-based cloud computing for a scientific data processing problem. At the same time, we describe how we developed a small extension to Hadoop's MapReduce which allows it to handle scientific data processing applications, and we report on our experience with performance studies.

Our paper focuses on a real scientific data processing case study: we develop a cloud application for a specific project that requires large-scale biological image processing. Large amounts of data are generated that need to be organized systematically in the cloud, and various types of data processing have to be performed (most algorithms use MATLAB). The application, built on top of Hadoop, allows users to submit data processing jobs to the cloud. At expected full production levels, the scale of the problem calls for the type of scalable and reliable solution platform that cloud computing can provide.

We find that we can use Hadoop without much modification: the only required change is to MapReduce input formats and the splitting of the input. Hadoop's MapReduce input is normally file-based, and we extend this to more general types of inputs, like file folders or database (DB) tables. More importantly, Hadoop normally splits files in parts that have equal binary size, irrespective of the boundaries between atomic input records. We modify this such that the user can easily provide a method that splits the input along boundaries between atomic input records. For example, in our application the atomic input records are folders with image files, and we allow the user to specify the number of groups these input folders have to be split in. In another example the atomic input records could be the rows of a DB table, and the user would be allowed to specify how many groups of rows the table needs to be split in. Note that

these are small modifications of a practical nature, especially since Google's MapReduce allows for such user-defined splitting methods as well. Nevertheless, Hadoop does not currently allow for this type of splitting in an easy way, and we describe an easy solution to this problem.

The scientific data processing problem considered in this paper is simple: the workload is divisible, without the need for communication between tasks. In this simple example there is only one processing phase, and thus there is no need to use the 'reduce' phase of Hadoop's MapReduce. Nevertheless, our cloud solution relies on many of the other features offered by the cloud concept and Hadoop, including scalability, reliability, fault-tolerance, easy deployability, etc.

As discussed in more detail at the end of this paper, our approach can be generalized easily to more complicated scientific data processing jobs (such as jobs with input data stored in a relational database, jobs that require a reduce phase after the map phase, scientific workflows, etc.). In this paper we focus on a simple real-world test case to demonstrate the applicability of cloud computing with Hadoop to scientific data processing in principle, and to relate our experiences in attempting to do this, and we leave the more complicated scientific computing applications for future work.

The rest of this paper is organized as follows. In the next section, we describe the biomedical image processing problem for which we develop a cloud application in this paper. The design of our system and the modifications to Hadoop's MapReduce are described in Section III, followed by performance analysis and system refinements in Section IV. This is followed by a discussion section (Section V) and an overview of related work in Section VI. Section VII formulates conclusions and describes future work.

2 Case Study

In this paper, we develop a Hadoop-based cloud computing application that processes sequences of microscope images of live cells. This project is a collaboration between groups at Genome Quebec/McGill University in Montreal, and at the University of Waterloo. We first describe the general context of the project, followed by a detailed description of the image processing tasks that will be considered later on in the paper.

2.1 General Description of Biomedical Image Processing Problem

Our goal is to study the complex molecular interactions that regulate biological systems. To achieve this we are developing an imaging platform to acquire and analyze live cell data at single cell resolution from populations of cells studied under different experimental conditions. The key feature of our acquisition system is its capability to record data in high throughput, both in the number of images that can be captured for a single experimental condition and the number of different experimental conditions that can be studied simultaneously. This is achieved by using an automated bright field and epifluorescence microscope in

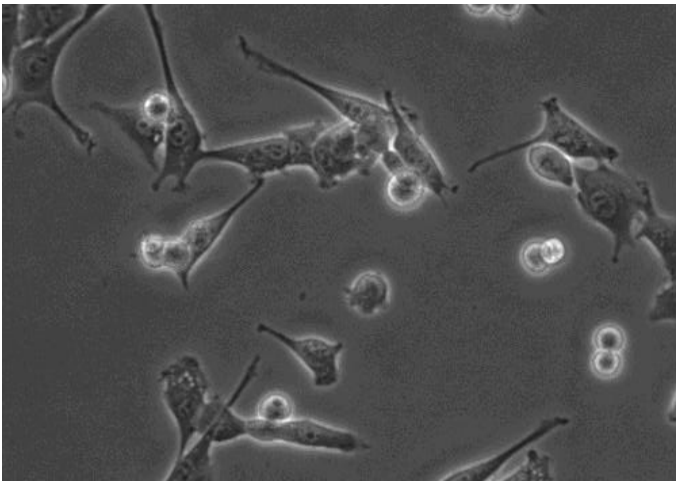

Fig. 1. Single microscope image with about two dozen cells on a grey background. Some interior structure can be discerned in every cell (including the cell membrane, the dark grey cytoplasm, and the lighter cell nucleus with dark nucleoli inside). Cells that are close to division appear as bright, nearly circular objects. In a typical experiment images are captured concurrently for 600 of these 'fields'. For each field we acquire about 900 images over a total duration of 48 hours, resulting in 260 GB of acquired data per experiment. The data processing task consists of segmenting each image and tracking all cells individually in time. The cloud application is designed to handle concurrent processing of many of these experiments and storing all input and output data in a structured way.

combination with miniaturized printed live cell assays. The acquisition system has a data rate of 1.5 MBps, and a typical 48 hour experiment can generate more than 260 GB of images, recorded as hundreds of multichannel videos each corresponding to a different treatment (Figure 1). Newer systems that we are currently evaluating can produce ten times more data in the same time.

The data analysis task for this platform is daunting: thousands of cells in the videos need to be tracked and characterized individually. The output consists of precise motion, morphological and gene expression data of each cell at many different timepoints. One of the particular systems we are studying is C2C12 myoblast differentiation - a process that results in the formation of muscle fibers from cultured cells. Cell differentiation, such as adipogenesis and myogenesis is mediated by gene transcription networks, which can be monitored by fluorescent probes. In a typical experiment, we need to measure the activity of several hundred different probes, called reporters, each of which records the activity of a gene regulatory circuit. For these experiments, the intensity of the fluorescent reporters is measured from the videos. While image analysis is the current bottleneck in our data processing pipeline, it happens to be a good candidate step for parallelization. The data processing can be broken up into hundreds of

independent video analysis tasks. The image analysis task uses computationally intensive code written in MATLAB to analyze both the data and generate result files. The analysis method we are currently developing solves the segmentation and tracking problem by first running a watershed segmentation algorithm. We then perform tracking by matching the segmented areas through time by using information about the cell shape intensity and position. As a final step we detect cell division events. The output data of the analysis is represented as a set of binary trees, each representing a separate cell lineage, where each node stores detailed information about a single cell at all time points.

To date we have used a local eight core server for data processing. A 600 video dataset takes up to 12h to process. This is the time required for one analysis of one experiment. Once the system will be fully operational, we will be acquiring large amounts of data (hundreds to thousands of GB per 48 hour experiment). We thus consider the development of a scalable and reliable cloud computing system for processing the data generated by our experiments of critical importance for our project.

2.2 Description of Specific Data Processing Problem

For each experiment (a specific set of parameters for the live cells under study), we may perform several data acquisitions. Typically, each acquisition generates 600 folders (one per field, see Figure 1), in which 900 acquired images are stored. Each image has a resolution of 512 x 512 16-bit pixels (512 KB), resulting in a total data size of 260 GB per acquisition. Different types of analysis (or data processing) jobs may be performed on the data gathered in each acquisition. Analysis jobs are normally performed using MATLAB programs, and the analysis can be parallelized easily, since each field can be processed independently. In the next Section we describe the design of a Hadoop-based cloud computing system for processing the data gathered in our live cell experiments.

3 System Design

3.1 Hadoop Components Used

As mentioned in Section I, we use Hadoop as our cloud computing framework. We use three Hadoop components: the MapReduce programming and execution environment, the reliable distributed file sytem called DFS, and a BigTable-like storage system for sparsely structured data called HBase. Each of the above components organizes cloud nodes into a master-slave structure. Figure 2 shows how the Hadoop components are used in our system. The DFS master node must be visible to all other components because they rely on DFS to store their data.Hadoop's MapReduce provides an API to write MapReduce programs as well as an environment to run those programs. Its master program is called "JobTracker", and slave programs are called "TaskTrackers". JobTracker accepts submitted user jobs, consisting of many map and reduce tasks with file-based

inputs. The large input files are split automatically into chunks that can be processed by the map workers (slaves) independently. The MapReduce framework distributes tasks with these input chunks to TaskTrackers to do the first-phase map computation. The results of the map phase are gathered and processed by the reduce phase, which generates final outputs. It is also acceptable to have map-only jobs. In this case, the reduce phase simply does nothing.

Hadoop's DFS is a flat-structure distributed file system. Its master node is called "namenode", and slave nodes are called "datanodes". Namenode is visible to all cloud nodes and provides a uniform global view for file paths in a traditional hierarchical structure. File contents are not stored hierarchically, but are divided into low level data chunks and stored in datanodes with replication. Data chunk pointers for files are linked to their corresponding locations by namenode.

HBase is a BigTable-like data store. It also employs a master-slave topology, where its master maintains a table-like view for users. Tables are split for distributed storage into row-wise "regions". The regions are stored on slave machines, using HBase's "region servers". Both the master and region servers rely on DFS to store data. Table I (See Figure 3) shows an example HBase table, which stores the web pages that link to web page www.cnn.com, along with the anchor text used in the link. Note that the order of the URL component strings is reversed in the row key, in an attempt to place links to pages that reside on the same web server in close-by rows, and thus making it likely that they are stored on the same datanode. The data stored in HBase are sorted key-value pairs logically organized as sparse tables indexed by row keys with corresponding column family values. Each column family represents one or more nested key-value pairs that are grouped together with the same column family as key prefix. The HBase table in Table I contains one row with key "com.cnn.www" and column family value "anchor:", which then contains two nested key-value pairs, with keys (web pages) "anchor:cnnsi.com" and "anchor:my.look.ca", and values (anchor text) "CNN" and "CNN.com".

3.2 System Overview

As shown in Figure 2, our system puts all essential functionality inside a cloud, while leaving only a simple Client at the experiment side for user interaction. In the cloud, we use DFS to store data, we use two HBase tables, called "Data" (Table II, Figure 3) and "Analysis" (Table III, Figure 3), to store metadata for data and for analysis jobs, respectively, and we use MapReduce to do computation. The Client can issue three types of simple requests to the cloud application (via the Cloud Frontend): a request for transfering experiment data (an acquisition) into the cloud's DFS, a request for performing an analysis job on a certain acquisition using a certain analysis program, and a request for querying/viewing analysis results. The Cloud Frontend processes Client requests. When it receives a data-transfer request, it starts an scp connection to the Client's local storage, and transfers data to its Temporary Staging Storage. It then puts the staged data from the Temporary Staging Storage into the DFS. It also updates the "Data" table in HBase to record the metadata that describes the acquisition (including

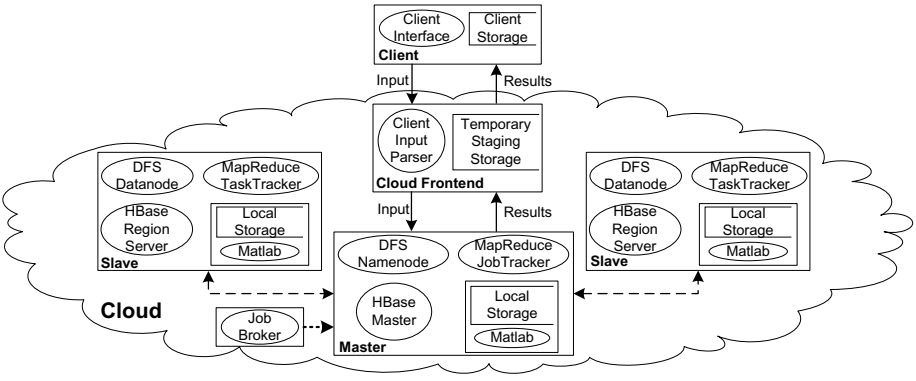

Fig. 2. System design diagram, showing the Hadoop components and cloud application components

TABLE I

EXAMPLE HBASE TABLE

Row Key	Column Family "anchor:"	
com.cnn.www	anchor:cnnsi.com	CNN
	anchor:my.look.ca	CNN.com

TABLE II

DATA TABLE IN HBASE

Row Key	Type	User	Acq ID	Exp. ID	Column Family: Raw:		Column Family: Result:	
DataKey1	Raw	X	1	A	Raw: FieldRange	1-10		
					Raw: FileParentDir	Data/expA/acq1/raw/		
DataKey2	Result	Y	1	A			Result:FieldRange	1-2
							Result:FileParentDir	Data/expA/acq1/Result/round1
							Result:RoundNum	1

TABLE III

ANALYSIS TABLE IN HBASE

Row Key	User	Input Data Row Key	Output Data Row Key	Round Number	Column Family: Program:		Status
AnalysisKey1	Y	DataKey1	DataKey2	1	Program:Name	CellLineage	new
					Program:Version	1	
					Program:Option	4	
					Program:InputFieldRange	1-2	

Fig. 3. HBase tables

the range of fields recorded in the acquisition, and the DFS path to the folder containing the fields). In the "Data" table, we distinguish records by type. The "Raw" type denotes raw experiment data, and the properties of a raw data acquisition are stored in the column family "Raw". The "Result" type denotes result data, which has different parameters than raw data, which are thus stored in a

different column family. In HBase's sparse data implementation, empty column families are not actually stored and do not take up resources; they are shown as empty in the logical table view maintained by the HBase Master. If the request is job submission or query, it inserts a record into the "Analysis" table in HBase or queries the "Analysis" table for the required information. The Job Broker shown in the bottom left corner of Figure 2 polls the "Analysis" table through regular "heart-beat" intervals to discover newly inserted unprocessed jobs, and submits the MapReduce jobs to the Master node Job Tracker to get processing started. The Job Broker can be run on any node of the Cloud. The Master node contains the DFS namenode, the MapReduce Job Tracker and the HBase master. We put the master processes of all three Hadoop components on one single Master node for simplicity, while they may also be put on separate nodes. All other nodes run as slaves to do both data storage and task processing. In our case, MATLAB is a native program that cannot use DFS files directly but requires its input and output files to reside on the local file system, we need to get files out of the DFS and copy them to local storage before MATLAB can start processing. Local storage, together with the MATLAB process started by the MapReduce application program, is shown as an isolated box inside each node in Figure 2. After MATLAB processing completes, the results are put back into DFS, and, when all Map tasks have been completed, the "Analysis" table in HBase is updated accordingly to reflect the change of status from "new" to "complete".

3.3 Programming MapReduce

In order to use Hadoop for our problem, it is necessary to extend the default way how Hadoop handles input data formats, how it handles the way input data are split into parts for processing by the map workers, and how it handles the extraction of atomic data records from the split data. Hadoop normally processes a very large data file containing a long sequence of atomic input records that each can be processed independently. The file is split automatically, normally along DFS block boundaries, and, due to the granularity of the atomic input records, the input splits usually each contain many atomic data records. For example, the atomic input record may be a line of text in a file separated by carriage returns. The file can be split at arbitrary locations, and record boundaries can be recovered easily. This type of automatic splitting is efficient and convenient for this type of problems.

For our scientific data processing application, however, the situation is different. In our case, an atomic data record is a folder of images corresponding to one field (total data size 512KB x number of images in that folder). The granularity is much coarser, and when we split the input, we may require just a few atomic input records per split (i.e., per map worker). In this case, it is more convenient and efficient to let the user control the number of splits, and to perform the split exactly at the boundary of the atomic input records. Also, it is more natural to provide the input indirectly, via paths to the DFS folders that contain the image

files, rather than providing the complete data set serialized into a single large file for all map tasks.

We have implemented this approach by writing new classes that implement the Hadoop interfaces for handling input and input splits. We have implemented the following classes:

1. StringArrayInputFormat.java (implements Hadoop's InputFormat interface).
2. CommaSeparatedStringInputSplit.java (implements Hadoop's InputSplit interface).
3. CommaSeparatedStringInputSplitRecordReader.java (implements Hadoop's RecordReader interface).

StringArrayInputFormat accepts as input a string array that contains the DFS paths to all the field data folders, and gets the number of splits that the user application desires from job configuration. It splits the input into smaller string array objects defined by CommaSeparatedStringInputSplit.java. When a map task is executed on a slave node, it is passed one of these small string arrays, specifying the input folders for the split to be processed. It then calls CommaSeparatedStringRecordReader to extract atomic input records (which each point to one field folder). The reader can download simplified versions of these classes from our project website [27].

In the next section we use this implementation for our cell data processing case study and evaluate its performance. Note that this approach can also be used to make Hadoop suitable for many other types of problems that do not directly fit within Hadoop's default context, namely, processing very long sequences of small atomic input records serialized into a single big file that is split automatically. In the discussion section below we give the example of processing records directly from a database table that is accessed remotely.

4 Performance Analysis and Optimizations

4.1 Environment Setup

We use a homogeneous cluster to do initial system development and proof-of-concept tests. The cluster is comprised of 21 SunFire X4100 servers with two dual-core AMD Opteron 280 CPUs interconnected by Gigabit Ethernet. The tests described below are performed on a smaller set of data, in which each 'field' folder contains 300 compressed images of size 300KB, for a total data size of 90MB per folder. The cloud is composed of 20 slave nodes for all tests.

4.2 MapReduce Performance

In this section we describe some simple runtime tests with our Hadoop-based cloud computing framework, mainly to show the functionality of our solution, and to give some insight in its performance. In fact, the use of hadoop allows to speed up calculations by a factor that equals the number of worker nodes,

except for startup effects, which are relatively small when the execution time of individual tasks is large enough.

Figure 4 shows a test in which we run an increasing number of tasks with one folder per task. It can be seen that the total time increases, which means that there is an overhead for scheduling.

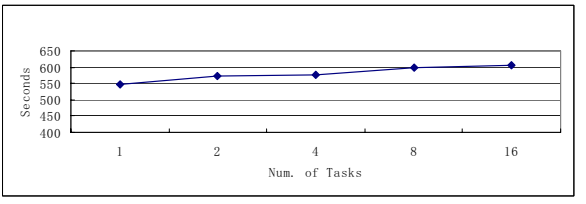

Fig. 4. Test for scheduling overhead

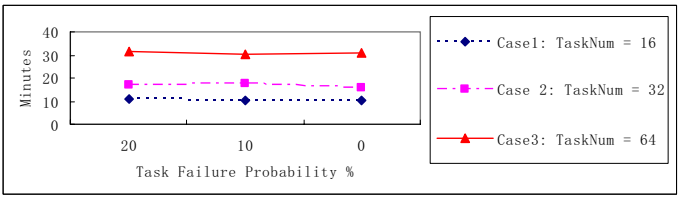

Fig. 5. Test for the degree of fault tolerance

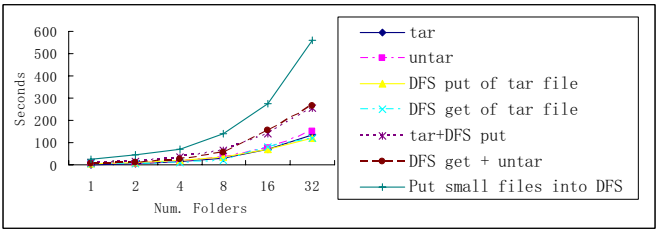

Fig. 6. Tar and DFS Performance

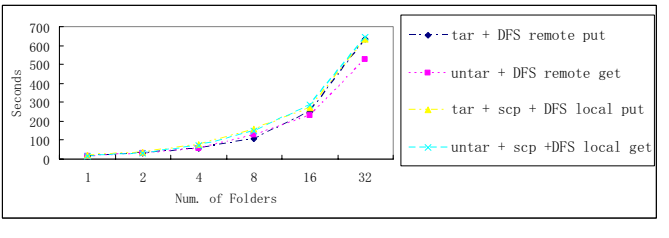

Fig. 7. DFS as remote staging tool

Figure 5 shows how the system is resilient against failures. We again consider jobs with one folder per task. Each task has a failure rate of 0%, 10% or 20%. (We artificially stop some of the tasks right after they start, with the probabilities specified.) Figure 5 shows that Hadoop is able to recover from these failures as expected. (It restarts tasks that have failed automatically.) The total time does not increase markedly when tasks fail, due to the fact that we let tasks fail immediately after they start. This shows that there is not much overhead associated with the fault-tolerance mechanism. The figure also shows that 64 tasks take about twice the time of 32 tasks, as expected. Interestingly, 32 tasks take less than double the time of 16 tasks (on 20 cloud nodes), which is likely due to an overhead cost associated with starting and completing a job.

4.3 Remote Data Staging and DFS Performance

In this section, we discuss optimizations in data transfer. It turns out that using tar files per directory results in a much more efficient system, both for transfer of data to and from the cloud, and for putting data from temporary storage into DFS and back.

In our system, data circulate through the following cycle:

1. Remote staging-in of raw experiment data: From Client to Cloud Front End.
2. Put raw data into DFS: From Cloud Front End to DFS.
3. Get raw data from DFS for MATLAB to process at each node: From DFS to Local Storage.
4. Put result data back to DFS after processing at each node: From Local Storage to DFS.
5. Get result data from DFS: From DFS to Cloud Front End.
6. Remote staging-out of result data: From Cloud Front End to Client.

Figure 6 shows that it takes approximately the same time to tar and untar a data folder, as it takes to put the tared file into DFS or take it out. Putting the untared folder into DFS takes unacceptably long. Therefore, we use tared files in our implementation.

Figure 7 shows that DFS can also be used efficiently as a remote data-staging tool: rather than first performing an scp transfer from the client to the cloud, followed by putting the tared folder from the cloud temporary storage into DFS, one can also run a Hadoop DFS datanode on the client machine outside the cloud, and put the data directly into DFS. Figure 7 shows that these two methods have almost the same efficiency. However, due to firewall constraints this may not be easy to do across organizations.

5 Discussion

5.1 Advantages and Disadvantages of Using HBase

One could consider using a regular relational database instead of HBase. Compared to a database, one major drawback of HBase is its inability to handle

complex queries. This is because HBase organizes data in sets containing key-value pairs, rather than relations. The query model used in relational databases can thus not be applied to HBase. We don't use a relational database in our project because of the following reasons:

1. It is complex and expensive to install and deploy large-scale scalable database management systems.
2. For our application, we don't need complex database-like queries as supported by database systems, and HBase performs well for the tasks we need it for.
3. DFS provides reliable storage, and current database systems cannot make use of DFS directly. Therefore, HBase may have a better degree of fault tolerance for large-scale data management in some sense.
4. Most importantly, we find it natural and convenient to model our data using sparse tables because we have varying numbers of fields with different metadata, and the table organization can change significantly as the usage of the system is extended to new analysis programs and new types of input or output data.

Because of these reasons, we believe that HBase is a good choice to do metadata management for problems like ours.

5.2 Extending Our Approach to Other Types of Data Processing

It is easy to extend our approach to other types of data processing based on our classes. For example, one can use a database table as input for doing simple distributed query processing using Hadoop's default approach, by retrieving all the table contents, putting it into one serialized file as input, and implementing a record reader that can detect atomic input record boundaries correctly. However, it is much more natural to let the Map workers read their own set of records from the database table directly. To this end, users can write a new class which extends our StringArrayInputFormat.java class, and override our getSplits method in it. It is easy to pass in the database jdbc url and the number of splits desired. The getSplits method would then first determine the total number of rows in the table, and then split the table into row ranges according to the specified number of splits. Each row range is then written in our CommaSeperatedStringInputSplit format as a split. Then the user can implement a record reader class such that, in the Mapper program, the database table will be accessed by the jdbc url and the row range specified in each split will be retrieved by the record reader and processed by the Mapper. Example code for remote database access can be downloaded from our website [27].

6 Related Work

Cloud computing and its usage are still in their infancy and not much literature is available in the forum of academic publications. As we've mentioned,

virtualization is used to construct a homogeneous environment for clouds while homogeneous clusters can also be easily configured for this purpose. Considering virtualization, [21] studies a negotiation and leasing framework for Cluster on Demand built from both Xen [7] virtual machines and physical machines. They built a Grid hosting environment as in [26] while a similar platform with simple unified scheduling among homogeneous machines can make this a Cloud environment. The authors of [28] study automatic virtual machine configuration for database workloads, in which several database management system instances, each running in a virtual machine, are sharing a common pool of physical computing resources. The performance optimization they made by controlling the configurations of the virtual machines in which they run may be extended to cloud environments for automatic configuration. Google File System [18] with its open source implemetation Hadoop DFS is currently a popular choice for a cloud file system. BigTable [11] and its open source implementation HBase act as data organization tools on top of the underlying distributed file system in the form of distributed column-oriented datastores. Other scalable data hosting and organization solutions include Amazon's Dynamo, Yahoo's PNUTS, Sinfonia, etc. Various data processing languages to facilitate usage of distributed data structures are also investigated in Sawzall [25], Pig Latin [24], and SCOPE [10]. Google's MapReduce [13], Microsoft's Dryad [22], and Clustera [15] investigate distributed programming and execution frameworks. MapReduce aims at simplicity with limited generality while Dryad provides generality but is complex to write programs with. Clustera is similar to Dryad but uses a different scheduling mechanism. MapReduce has been used for a variety of applications [9,17,23].

7 Conclusions and Future Work

In this paper, we have evaluated Hadoop by doing a case study on a scientific data processing problem with MATLAB. In the case study, we extend the capability of Hadoop's MapReduce in accepting arbitrary input formats with explicit split control. Our system design and implementation can be reused for similar application scenarios and extended to arbitrary applications with divisible inputs. We have discussed performance evaluation runs and optimizations in terms of data transfer mechanisms.

The scientific data-processing problem considered in this paper is simple: the workload is divisible, without the need for communication between tasks. Nevertheless, our cloud solution is attractive because it takes advantage of many of the desirable features offered by the cloud concept and Hadoop, including scalability, reliability, fault-tolerance, easy deployability, etc. Our approach can be generalized easily to more complicated scientific data processing jobs than the examples given above. For instance, we have already designed and implemented a computational workflow system based on Hadoop that supports workflows built from MapReduce programs as well as legacy programs (existing programs not built by using the MapReduce API) [30]. We also plan to investigate how we can process jobs with intertask communications. Building on the experience

presented in this paper, we will address Hadoop-based cloud computing for more complicated scientific computing applications in future work.

Acknowledgements

We want to give thanks to Genome Quebec, the University of Waterloo Faculty of Mathematics and David R. Cheriton School of Computer Science, and SHARCNET for providing the computing infrastructure for this work.

References

1. Aguilera, M.K., Merchant, A., Shah, M.A., Veitch, A.C., Karamanolis, C.T.: Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In: SOSP 2007 (2007)
2. Aguilera, M., Golab, W., Shah, M.: A Practical Scalable Distributed B-Tree. In: VLDB 2008 (2008)
3. Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2/` (retrieved date: September 27, 2009)
4. Apache Hadoop, `http://hadoop.apache.org/` (retrieved date: September 27, 2009)
5. Apache HBase, `http://hadoop.apache.org/hbase/` (retrieved date: September 27, 2009)
6. Apache Hama, `http://incubator.apache.org/hama/` (retrieved date: September 27, 2000)
7. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauery, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: SOSP 2003 (2003)
8. Brantner, M., Florescu, D., Graf, D.A., Kossmann, D., Kraska, T.: Building a Database on S3. In: SIGMOD 2008 (2008)
9. Catanzaro, B., Sundaram, N., Keutzer, K.: A MapReduce framework for programming graphics processors. In: Workshop on Software Tools for MultiCore Systems (2008)
10. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In: VLDB 2008 (2008)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: BigTable: A Distributed Storage System for Structured Data. In: OSDI 2006 (2006)
12. Cooper, B., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform. In: VLDB 2008 (2008)
13. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI 2004 (2004)
14. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-Value Store. In: SOSP 2007 (2007)
15. DeWitt, D.J., Robinson, E., Shankar, S., Paulson, E., Naughton, J., Krioukov, A., Royalty, J.: Clustera: An Integrated Computation and Data Management System. In: VLDB 2008 (2008)

16. ELASTRA, http://www.elastra.com/ (retrieved date: Sepember 27, 2009)
17. Elsayed, T., Lin, J., Oard, D.: Pairwise Document Similarity in Large Collections with MapReduce. In: Proc. Annual Meeting of the Association for Computational Linguistics (2008)
18. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. In: SOSP 2003 (2003)
19. GigaSpaces, http://www.gigaspaces.com/ (retrieved date: September 27, 2009)
20. Google and IBM Announce University Initiative, http://www.ibm.com/ibm/ideasfromibm/us/google/index.shtml (retrieved date: September 27, 2009)
21. Irwin, D.E., Chase, J.S., Grit, L.E., Yumerefendi, A.R., Becker, D., Yocum, K.: Sharing Networked Resources with Brokered Leases. In: USENIX Annual Conference 2006 (2006)
22. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: EuroSys 2007 (2007)
23. McNabb, A.W., Monson, C.K., Seppi, K.D.: MRPSO: MapReduce Particle Swarm Optimization. In: Genetic and Evolutionary Computation Conference (2007)
24. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: SIGMOD 2008 (2008)
25. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall. Scientific Programming 13(4) (2005)
26. Ramakrishnan, L., Irwin, D.E., Grit, L.E., Yumerefendi, A.R., Iamnitchi, A., Chase, J.S.: Toward a Doctrine of Containment: Grid Hosting with Adaptive Resource Control. In: SC 2006 (2006)
27. Scalable Scientific Computing Group, University of Waterloo, http://www.math.uwaterloo.ca/groups/SSC/software/cloud (retrieved date: September 27, 2009)
28. Soror, A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosielis, P., Kamath, S.: Automatic Virtual Machine Configuration for Database Workloads. In: SIGMOD 2008 (2008)
29. Yang, H.C., Dasdan, A., Hsiao, R.-L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD 2007 (2007)
30. Zhang, C., De Sterck, H.: CloudWF: A Computational Work ow System for Clouds Based on Hadoop. In: The First International Conference on Cloud Computing, Beijing, China (2009)