On-the-fly Task Execution for Speeding Up Pipelined MapReduce

Diana Moise, Gabriel Antoniu, Luc Bougé

On-the-fly Task Execution for Speeding Up Pipelined MapReduce

Diana Moise[1], Gabriel Antoniu[1], and Luc Bougé[2]

[1] INRIA Rennes - Bretagne Atlantique / IRISA, France
[2] ENS Cachan - Brittany / IRISA, France

Abstract. The MapReduce programming model is widely acclaimed as a key solution to designing data-intensive applications. However, many of the computations that fit this model cannot be expressed as a single MapReduce execution, but require a more complex design. Such applications consisting of multiple jobs chained into a long-running execution are called pipeline MapReduce applications. Standard MapReduce frameworks are not optimized for the specific requirements of pipeline applications, yielding performance issues. In order to optimize the execution on pipelined MapReduce, we propose a mechanism for creating map tasks along the pipeline, as soon as their input data becomes available. We implemented our approach in the Hadoop MapReduce framework. The benefits of our dynamic task scheduling are twofold: reducing job-completion time and increasing cluster utilization by involving more resources in the computation. Experimental evaluation performed on the Grid'5000 testbed, shows that our approach delivers performance gains between 9% and 32%.

Keywords: MapReduce, pipeline MapReduce applications, intermediate data management, task scheduling, Hadoop, HDFS

1 Introduction

The MapReduce abstraction has revolutionized the data-intensive community and has rapidly spread to various research and production areas. Google introduced MapReduce [8] as a solution to the need to process datasets up to multiple terabytes in size on a daily basis. The goal of the MapReduce programming model is to provide an abstraction that enables users to perform computations on large amounts of data.

The MapReduce abstraction is inspired by the "map" and "reduce" primitives commonly used in functional programming. When designing an application using the MapReduce paradigm, the user has to specify two functions: *map* and *reduce* that are executed in parallel on multiple machines. Applications that can be modeled by the means of MapReduce, mostly consist of two computations: the "map" step, that applies a filter on the input data, selecting only the data that satisfies a given condition, and the "reduce" step, that collects and aggregates all the data produced by the first phase. The MapReduce model exposes a simple interface, that can be easily manipulated by users without any experience with parallel and distributed systems. However, the interface is versatile enough so that it can be employed to suit a wide range of data-intensive applications. These are the main reasons for which MapReduce has known an increasing popularity ever since it was introduced.

An open-source implementation of Google's abstraction was provided by Yahoo! through the Hadoop [5] project. This framework is considered the reference MapReduce implementation and is currently heavily used for various purposes and on several infrastructures. The MapReduce paradigm has also been adopted by the cloud computing community as a support to those cloud-based applications that are data-intensive. Cloud providers support MapReduce computations so as to take advantage of the huge processing and storage capabilities the cloud holds, but at the same time, to provide the user with a clean and easy-to-use interface. Amazon released ElasticMapReduce [2], a web service that enables users to easily and cost-effectively process large amounts of data. The service consists of a hosted Hadoop framework running on Amazon's Elastic Compute Cloud (EC2) [1]. Amazon's Simple Storage Service (S3) [3] serves as storage layer for Hadoop. AzureMapReduce [9] is an implementation of the MapReduce programming model, based on the infrastructure services the Azure cloud [6] offers. Azure's infrastructure services are built to provide scalability, high throughput and data availability. These features are used by the AzureMapReduce runtime as mechanisms for achieving fault tolerance and efficient data processing at large scale.

MapReduce is used to model a wide variety of applications, belonging to numerous domains such as analytics (data processing), image processing, machine learning, bioinformatics, astrophysics, etc. There are many scenarios in which designing an application with MapReduce requires the users to employ several MapReduce processing. These applications that consist of multiple MapReduce jobs chained into a long-running execution, are called *pipeline MapReduce applications*. In this paper, we study the characteristics of pipeline MapReduce applications, and we focus on optimizing their execution. Existing MapReduce frameworks manage pipeline MapReduce applications as a sequence of MapReduce jobs. Whether they are employed directly by users or through higher-level tools, MapReduce frameworks are not optimized for executing pipeline applications. A major drawback comes from the fact that the jobs in the pipeline have to be executed sequentially: a job cannot start until all the input data it processes has been generated by the previous job in the pipeline.

In order to speed up the execution of pipelined MapReduce, we propose a new mechanism for creating "map" tasks along the pipeline, as soon as their input data becomes available. Our approach allows successive jobs in the pipeline to overlap the execution of "reduce" tasks with that of "map" tasks. In this manner, by dynamically creating and scheduling tasks, the framework is able to complete the execution of the pipeline faster. In addition, our approach ensures a more efficient cluster utilization, with respect to the amount of resources that are involved in the computation. We implemented the proposed mechanisms in the Hadoop MapReduce framework [5] and evaluated the benefits of our approach through extensive experimental evaluation.

In section 2 we present an overview of pipelined MapReduce as well as the scenarios in which this type of processing is employed. Section 3 introduces the mechanisms we propose and shows their implementation in Hadoop. Section 4 is dedicated to the experiments we performed; we detail the environmental setup and the scenarios we selected for execution in order to measure the impact of our approach. Section 5 summarizes the contributions of this work and presents directions for future work.

2 Pipeline MapReduce applications: overview and related work

Many of the computations that fit the MapReduce model, cannot be expressed as a single MapReduce execution, but require a more complex design. These applications that consist of multiple MapReduce jobs chained into a long-running execution, are called *pipeline MapReduce applications*. Each stage in the pipeline is a MapReduce job (with 2 phases, "map" and "reduce"), and the output data produced by one stage is fed as input to the next stage in the pipeline. Usually, pipeline MapReduce applications are long-running tasks that generate large amounts of *intermediate* data (the data produced between stages). This type of data is transferred between stages and has different characteristics from the meaningful data (the input and output of an application). While the input and output data are expected to be persistent and are likely to be read multiple times (during and after the execution of the application), intermediate data is *transient* data that is usually *written once*, by one stage, and *read once*, by the next stage.

However, there are few scenarios in which users directly design their application as a pipeline of MapReduce jobs. Most of the use cases of MapReduce pipelines come from applications that *translate* into a chain of MapReduce jobs. One of the drawbacks of the extreme simplicity of the MapReduce model is that it cannot be straightforwardly used in more complex scenarios. For instance, in order to use MapReduce for higher-level computations (for example, the operations performed in the database domain) one has to deal with issues like multi-stage execution plan, branching data-flows, etc. The trend of using MapReduce for database-like operations led to the development of high-level query languages that are executed as MapReduce jobs, such as Hive [14], Pig [12], and Sawzall [13]. Pig is a distributed infrastructure for performing high-level analysis on large data sets. The Pig platform consists of a high-level query language called *PigLatin* and the framework for running computations expressed in PigLatin. PigLatin programs comprise SQL-like high-level constructs for manipulating data that are interleaved with MapReduce-style processing. The Pig framework compiles these programs into a pipeline of MapReduce jobs that are executed within the Hadoop environment.

The scenarios in which users actually devise their applications as MapReduce pipelines, involve binary data whose format does not fit the high-level structures of the aforementioned frameworks. In order to facilitate the design of pipeline MapReduce applications, Cloudera recently released Crunch [4], a tool that generates a pipeline of MapReduce jobs and manages their execution. While there are several frameworks that generate pipeline MapReduce applications, few works focus on optimizing the actual execution of this type of applications. In [11], the authors propose a tool for estimating the progress of MapReduce pipelines generated by Pig queries. The Hadoop Online Prototype (HOP) [7] is a modified version of the Hadoop MapReduce framework that supports online aggregation, allowing users to get snapshots from a job as it is being computed. HOP employs pipelining of data between MapReduce jobs, i.e., the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job. However, by circumventing the storing of data in a distributed file system (DFS) between the jobs, fault tolerance cannot be guaranteed by the system. Furthermore, as the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped, the jobs in the pipeline are executed sequentially.

3 Introducing dynamic scheduling of map tasks in Hadoop

3.1 Motivation

In a pipeline of MapReduce applications, the intermediate data generated between the stages represents the output data of one stage and the input data for the next stage. The intermediate data is produced by one job and consumed by the next job in the pipeline. When running this kind of applications in a dedicated framework, the intermediate data is usually stored in the distributed file system that also stores the user input data and the output result. This approach ensures intermediate data availability, and thus, provides fault tolerance, a very important factor when executing pipeline applications. However, using MapReduce frameworks to execute pipeline applications raises performance issues, since MapReduce frameworks are not optimized for the specific features of intermediate data. The main performance issue comes from the fact that the jobs in the pipeline have to be executed sequentially: a job cannot start until all the input data it processes has been generated by the job in the previous stage of the pipeline. Consequently, the framework runs only one job at a time, which results in inefficient cluster utilization and basically, a waste of resources.

3.2 Executing pipeline MapReduce applications with Hadoop

The Hadoop project provides an open-source implementation of Google's MapReduce paradigm through the Hadoop MapReduce framework [5, 15]. The framework was designed following Google's architectural model and has become the reference MapReduce implementation. The architecture is tailored in a master-slave manner, consisting of a single master *jobtracker* and multiple slave *tasktrackers*. The jobtracker's main role is to act as the task *scheduler* of the system, by assigning work to the tasktrackers. Each tasktracker disposes of a number of available *slots* for running tasks. Every active map or reduce task takes up one slot, thus a tasktracker usually executes several tasks simultaneously. When dispatching "map" tasks to tasktrackers, the jobtracker strives at keeping the computation as close to the data as possible. This technique is enabled by the data-layout information previously acquired by the jobtracker. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes closer to the data (belonging to the same network rack). The jobtracker first schedules "map" tasks, as the reducers must wait for the "map" execution to generate the intermediate data.

Hadoop executes the jobs of a pipeline MapReduce application in a sequential manner. Each job in the pipeline consists of a "map" and a "reduce" phase. The "map" computation is executed by Hadoop tasktrackers only when all the data it processes is available in the underlying DFS. Thus, the mappers are scheduled to run only after all the reducers from the preceding job have completed their execution. This scenario is also representative for a Pig processing: the jobs in the logical plan generated by the Pig framework are submitted to Hadoop sequentially. In consequence, at each step of the pipeline, at most the "map" and "reduce" tasks of the same job are being executed. Running the mappers and the reducers of a single job involves only a part of the cluster nodes. The rest of the computational and storage cluster capabilities remains idle.

3.3 Our approach

In order to speed-up the execution of pipeline MapReduce applications, and also to improve cluster utilization, we propose an optimized Hadoop MapReduce framework, in which the scheduling is done in a *dynamic* manner. For a better understanding of our approach, we first detail the process through which "map" and "reduce" tasks are created and scheduled in the original Hadoop MapReduce framework.

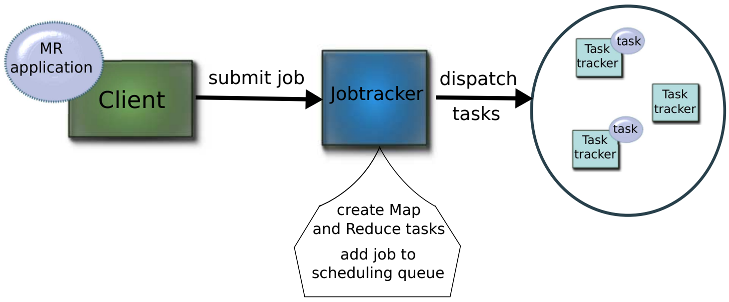

Fig. 1. Job submission process in Hadoop.

Figure 1 displays the job submission process. The first step is for the user to specify the "map" and "reduce" computations of the application. The Hadoop client generates all the job-related information (input and output directories, data placement, etc.) and then submits the job for execution to the jobtracker. On the jobtracker's side, the list of "map" and "reduce" tasks for the submitted job is created. The number of "map" tasks is equal to the number of chunks in the input data, while the number of "reduce" tasks is computed by taking into account various factors, such as the cluster capacity, the user specification, etc. The list of tasks is added to the *job queue* that holds the jobs to be scheduled for execution on tasktrackers. In the Hadoop MapReduce framework, the "map" and "reduce" tasks are created by the jobtracker when the job is submitted for execution. When they are created, the "map" tasks require to know the location of the chunks they will work on.

In the context of multiple jobs executed in a pipeline, the jobs are submitted by the client to the jobtracker sequentially, as the chunk-location information is available only when the previous job completes. Our approach is based on the remark that a "map" task is created for a single input chunk. It only needs to be aware of this very chunk location. Furthermore, when it is created, the only information that the "map" task requires, is the list of nodes that store the data in its associated chunk. We modified the Hadoop MapReduce framework to create "map" tasks *dynamically*, that is, as soon as a chunk is available for processing. This approach can bring substantial benefits to the execution of pipeline MapReduce applications. Since the execution of a job can start as soon as the first chunk of data is generated by the previous job, the total runtime is significantly reduced. Additionally, the tasks belonging to several jobs in the pipeline can be executed at the same time, which leads to a more efficient cluster utilization.

The modifications and extensions of the Hadoop MapReduce framework that we propose, are further presented and summarized on Figure 2.

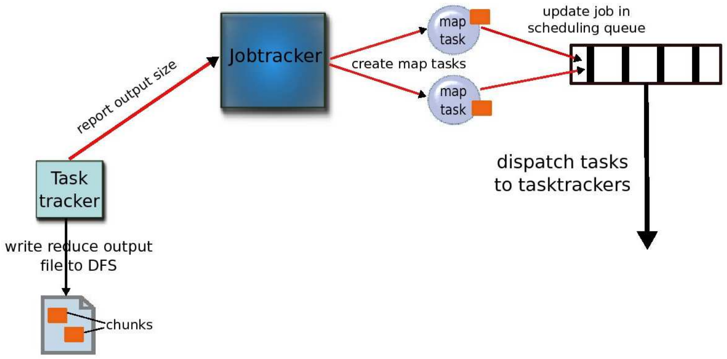
Jobtracker
report output size
create map tasks
map task
map task
update job in scheduling queue
dispatch tasks to tasktrackers
Task tracker
write reduce output file to DFS
chunks

Fig. 2. Dynamic creation of "map" tasks.

Algorithm 1 Report output size (**on tasktracker**)

```
procedure CommitTask
    $(size, files) \leftarrow tasktracker.writeReduceOutputData()$
    $jobtracker.transmitOutputInfo(size, files)$
end procedure
```

Job-submission process

Client side. On the client side, we modified the submission process between the Hadoop client and the jobtracker. Instead of waiting for the execution to complete, the client launches a *job monitor* that reports the execution progress to the user. With this approach, a pipeline MapReduce application employs a single Hadoop client to run the application. The client submits all the jobs in the pipeline from the beginning, instead of submitting them sequentially, i.e., the modified Hadoop client submits the whole set of jobs $job_1...job_n$ for execution.

Jobtracker side. The job-submission protocol is similar to the one displayed on Figure 1. However, at submission time, only the input data for job_1 is available in the DFS. Regarding $job_2...job_n$, the input data has to be generated throughout the pipeline. Thus, the jobtracker creates the set of "map" and "reduce" tasks only for job_1. For the rest of the jobs, the jobtracker creates only "reduce" tasks, while "map" tasks will be created along the pipeline, as the data is being generated.

Job scheduling

Tasktracker side: For a job_i in the pipeline, the data produced by the job's "reduce" phase ($reduce_i$) represents the input data of job_{i+1}'s "map" task (map_{i+1}). When $reduce_i$ is completed, the tasktracker writes the output data to the backend storage. We modified the tasktracker code to notify the jobtracker whenever it successfully completes the execution of a "reduce" function: the tasktracker informs the jobtracker about the size of the data produced by the "reduce" task.

Algorithm 2 Update job (**on jobtracker**)

```
procedure TRANSMITOUTPUTINFO($size, files$)
    invoke updateJob(size, files) on taskscheduler
end procedure

procedure UPDATEJOB($size, files$)
    for all $job \in jobQueue$ do
        $dir \leftarrow job.getInputDirectory()$
        if $dir = getDirectory(files)$ then
            if $writtenBytes.contains(dir) = False$ then
                $writtenBytes.put(dir, size)$
            else
                $allBytes \leftarrow writtenBytes.get(dir)$
                $writtenBytes.put(dir, allBytes + size)$
            end if
            $allBytes \leftarrow writtenBytes.get(dir)$
            if $allBytes \geq$ CHUNK_SIZE then
                $b \leftarrow job.createMapsForSplits(files)$
                $writtenBytes.put(dir, allBytes - b)$
            else
                $job.addToPending(files)$
            end if
        end if
    end for
end procedure
```

Jobtracker side: In our modified framework, the jobtracker keeps track of the output data generated by reducers in the DFS. This information is important for the scheduling of the jobs in the pipeline, as the output directory of job_i is the input directory of job_{i+1}. Each time data is produced in job_i's output directory, the jobtracker checks to see if it can create new "map" tasks for job_{i+1}. If the data accumulated in job_{i+1}'s input directory is at least of the size of a chunk, the jobtracker creates "map" tasks for the newly generated data. For each new chunk, the jobtracker creates a "map" task to process it. All the "map" tasks are added to the scheduling queue and then dispatched to idle tasktrackers for execution.

The modifications on the tasktracker side are described in Algorithm 1. We extended the code with a primitive that sends to the jobtracker the information about the "reduce" output data: the files written to the DFS and the total size of the data. Algorithm 2 shows the process of updating a job with information received from tasktrackers. The algorithm is integrated in the jobtracker code, mainly in the scheduling phase. The jobtracker also plays the role of *task scheduler*. It keeps a list of data written to the input directory of each job. For each received update, the jobtracker checks if the data in the job's input directory reaches at least a chunk in size (64 MB default). If it is the case, "map" tasks will be created, one per each new data chunk. Otherwise, the job's information is stored for subsequent processing. The mechanism of creating "map" tasks is presented in Algorithm 3, executed by the jobtracker, and integrated into the job

Algorithm 3 Create map tasks (**on job**)

```
procedure ADDTOPENDING($files$)
    $pendingFiles.addAll(files)$
end procedure
function CREATEMAPSFORSPLITS($files$) returns splitBytes
    $pendingFiles.addAll(files)$
    $splits \leftarrow getSplits(pendingFiles)$
    $pendingFiles.clear()$
    $newSplits \leftarrow splits.length$
    $jobtracker.addWaitingMaps(newSplits)$
    for $i \in [1..newSplits]$ do
        $maps[numMapTasks + i] \leftarrow newMapTask(splits[i])$
    end for
    $numMapTasks \leftarrow numMapTasks + newSplits$
    $notifyAllReduceTasks(numMapTasks)$
    for all $s \in splits$ do
        $splitBytes \leftarrow splitBytes + s.getLength()$
    end for
    return $splitBytes$
end function
```

code. We extended the code so that each job holds the list of files that were generated so far in the job's input directory. When the jobtracker computes that at least a chunk of input data has been generated, new "map" tasks are created for the job. The data in the files is split into chunks. A "map" task is created for each chunk and the newly launched tasks are added to the scheduling queue. The jobtracker also informs the "reduce" tasks that the number of "map" tasks has changed. The reducers need to be aware of the number of mappers of the same job, as they have to transfer their assigned part of the output data from all the mappers to their local disk.

4 Evaluation

We validate the proposed approach through a series of experiments that compare the original Hadoop framework with our modified version, when running pipeline applications.

4.1 Environmental setup

The experiments were carried out on the Grid'5000 [10] testbed. The Grid'5000 project is a widely-distributed infrastructure devoted to providing an experimental platform for the research community. The platform is spread over 10 geographical sites located through on French territory and 1 in Luxembourg. For our experiments, we employed nodes from the Orsay cluster of the Grid'5000. The nodes are outfitted with dual-core x86_64 CPUs and 2 GB of RAM. Intra-cluster communication is done through a 1 Gbps Ethernet network. We performed an initial test at a small scale, i.e., 20 nodes, in order to assess the impact of our approach. The second set of tests involves 50 nodes belonging to the Orsay cluster.

4.2 Results

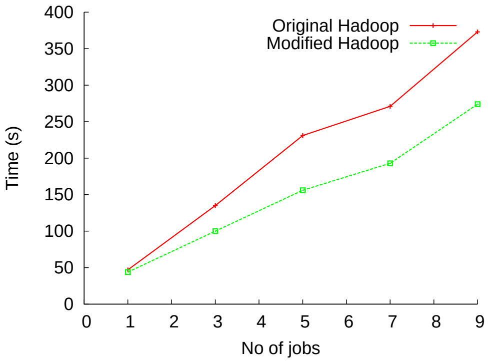

Fig. 3. Completion time for short-running pipeline applications.

The evaluation presented here focuses on assessing the performance gains of the optimized MapReduce framework we propose, over the original one. To this end, we developed a benchmark that creates a pipeline of n MapReduce jobs and submits them to Hadoop for execution. Each job in the pipeline simulates a load that parses key-value pairs from the input data and outputs 90% of them as final result. In this manner, we manage to obtain a long-running application that generates a large amount of data, allowing our dynamic scheduling mechanism to optimize the execution of the pipeline. The computation itself is not relevant in this case, as our goal is to create a scenario in which enough data chunks are generated along the pipeline so that “map” tasks can be dynamically created. We run this type of application first with the original Hadoop framework, then with our optimized version of Hadoop. In both cases, we measure the pipeline completion-time and compare the results.

Short-running pipeline applications

In a first set of experiments, we run the benchmark in a small setup involving 20 nodes, on top of which HDFS and Hadoop MapReduce are deployed as follows: a dedicated machine is allocated for each centralized entity (namenode, jobtracker), a node serves as the Hadoop client that submits the jobs, and the rest of 17 nodes represent both datanodes and tasktrackers. At each step, we keep the same deployment setup and we increase the number of jobs in the pipeline to be executed. The first test consists in running a single job, while the last one runs a pipeline of 9 MapReduce jobs. The application’s input data, i.e., job_1’s input data, consists of 5 data chunks (a total of 320 MB). Job_i keeps 90% of the input data it received from job_{i-1}. In the case of the 9-job pipeline, this data-generation mechanism leads to a total of 2 GB of data produced throughout the pipeline, out of which 1.6 GB account for intermediate data.

Figure 3 shows the execution time of pipeline applications consisting of an increasing number of jobs (from 1 to 9), in two scenarios: when running on top of the original Hadoop, and with the pipeline-optimized version we proposed. In the first case, the client sequentially submits the jobs in the pipeline to Hadoop's jobtracker, i.e., waits for the completion of job_i before submitting job_{i+1}. When using our version of Hadoop, the client submits all the jobs in the pipeline from the beginning, and then waits for the completion of the whole application. As expected, the completion time in both cases increases proportionally to the number of jobs to be executed. However, our framework manages to run the jobs faster, as it creates and schedules "map" tasks as soon as a chunk of data is generated during the execution. This mechanism speeds-up the execution of the entire pipeline, and also exhibits a more efficient cluster utilization. Compared to the original Hadoop, we obtain a performance gain between 26% and 32%.

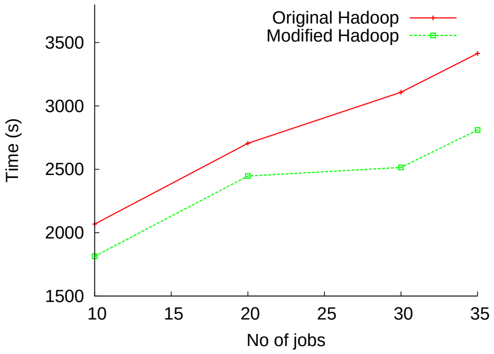

Fig. 4. Completion time for long-running pipeline applications.

Long-running pipeline applications

The first experiment we presented was focused on pipeline applications that consist of a small up to a medium number of jobs (1 to 9). Due to the long-running nature of pipeline applications and considering the significant size of the intermediate data our benchmark generates, we performed experiments with larger applications and larger datasets in a different setup, including 50 nodes. HDFS and Hadoop MapReduce are deployed as for the previous experiment, employing thus 47 tasktrackers. The size of the input data for each pipeline application amounts to 2.4 GB (40 data chunks). We vary the number of jobs to be executed in each pipeline, from 10 to 35. For the longest-running application, the generated data add up to a total of 24.4 GB.

The results for this setup are displayed on Figure 4. Consistently with the previous results, our approach proves to be more efficient for long-running applications as well. The performance gains vary between 9% and 19% in this scenario. The benefits of our optimized framework have a smaller impact in this case, because of the data size involved in the experiment. Since more chunks are used as input, and substantially more chunks are being generated throughout the pipeline, a large part of the tasktrackers is involved in the current computation, leaving a smaller number of resources available for dynamically running created "map" tasks.

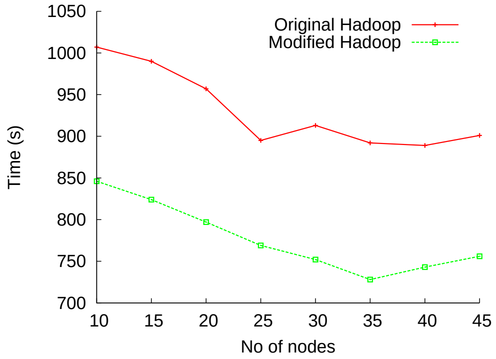

Fig. 5. Impact of deployment setup on performance.

Scaling out

In the context of pipeline applications, the number of nodes involved in the Hadoop deployment can have a substantial impact on completion time. Furthermore, considering our approach of dynamic scheduling "map" tasks, the scale of the deployment is an important factor to take into account. Thus, we performed an experiment in which we vary the number of nodes employed by the Hadoop framework. At each step, we increase the number of nodes used for the deployment, such that the number of tasktrackers that execute "map" and "reduce" tasks is varied from 10 to 45. In each setup, we run the aforementioned benchmark with a fixed number of 7 jobs in the pipeline. The input data is also fixed, consisting of 25 chunks of data, i.e., 1.5 GB.

Figure 5 shows the completion time of the 7-job pipeline when running with both original Hadoop and modified Hadoop, while increasing the deployment setup. As the previous experiments also showed, our improved framework manages to execute the jobs faster than the original Hadoop. In both cases, as more nodes are added to the deployment, the application is executed faster, as more tasktrackers can be used for running the jobs. However, increasing the number of nodes yields performance gains up to a point, which corresponds to 25 tasktrackers for the original Hadoop. This number is strongly related to the number of chunks in the input data, since the jobtracker schedules a tasktracker to run the "map" computation on each chunk. For the modified Hadoop, the point after which expanding the deployment does not prove to be profitable any longer, is higher than for the original Hadoop. The reason for this behavior lies in the scheduling approach of both frameworks: in original Hadoop, the scheduling of jobs is done sequentially, while in modified Hadoop, the "map" tasks of each job are scheduled as soon as the data is generated. The completion time starts to increase in both cases after a certain point, as the overhead of launching and managing a larger number of tasktrackers overcomes the advantage of having more nodes in the deployment.

5 Conclusions

In this paper we address a special class of MapReduce applications, i.e., applications that consist of multiple jobs executed in a pipeline. In this context, we focus on improving the performance of the Hadoop MapReduce framework when executing pipelines. Our proposal consists mainly of a new mechanism for creating tasks along the pipeline, as soon as their input data become available. This dynamic task scheduling leads to an improved performance of the framework, in terms of job completion time. In addition, our approach ensures a more efficient cluster utilization, with respect to the amount of resources involved in the computation. The approach presented in this paper can be further extended so as to allow the overlapping of several jobs in the pipeline. However, this aspect would require careful tuning of the scheduling of tasks in MapReduce frameworks. Deciding whether to execute reducers for the current job or to start mappers for the next jobs is a crucial aspect that requires complex metrics. As future direction, we also plan to validate the proposed approach through experiments with higher-level frameworks in the context of pipelined MapReduce, such as Pig.

Acknowledgments

This work was supported in part by the Agence Nationale de la Recherche (ANR) under Contract ANR-10-SEGI-01 (MapReduce Project). The experiments presented in this paper were carried out using the Grid'5000 testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.org/).

References

1. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.
2. Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.
3. Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.
4. Crunch. https://github.com/cloudera/crunch.
5. The Hadoop MapReduce Framework. http://hadoop.apache.org/mapreduce/.
6. The Windows Azure Platform. http://www.microsoft.com/windowsazure/.
7. T. Condie, N. Conway, P. Alvaro, and all. Mapreduce online. In *Procs of NSDI'10*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.
8. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
9. T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. MapReduce in the Clouds for Science. In *Procs. of CLOUDCOM '10*, pages 565–572, Washington, DC, 2010.
10. Y. Jégou, S. Lantéri, J. Leduc, and all. Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. *Intl. Journal of HPC Applications*, 20(4):481–494, Nov. 2006.
11. K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Procs. of ICDE*, pages 681–684. IEEE, 2010.
12. C. Olston, B. Reed, U. Srivastava, and all. Pig Latin: a not-so-foreign language for data processing. In *Procss of SIGMOD'08*, pages 1099–1110, NY, USA, June 2008. ACM.
13. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13:277–298, October 2005.
14. A. Thusoo, J. S. Sarma, N. Jain, and all. Hive: A warehousing solution over a MapReduce framework. In *Procs of VLDB'09*, pages 1626–1629, 2009.
15. T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.