Performance-Driven Task Co-Scheduling for MapReduce Environments

Jordà Polo, David Carrera, Yolanda Becerra,
Jordi Torres and Eduard Ayguadé
Barcelona Supercomputing Center (BSC) -
Technical University of Catalonia (UPC)
Barcelona, Spain

Malgorzata Steinder
and Ian Whalley
IBM T.J. Watson Research Center
Hawthorne, NY 10532

***Abstract*—MapReduce is a data-driven programming model proposed by Google in 2004 which is especially well suited for distributed data analytics applications. We consider the management of MapReduce applications in an environment where multiple applications share the same physical resources. Such sharing is in line with recent trends in data center management which aim to consolidate workloads in order to achieve cost and energy savings. In a shared environment, it is necessary to predict and manage the performance of workloads given a set of performance goals defined for them. In this paper, we address this problem by introducing a new task scheduler for a MapReduce framework that allows performance-driven management of MapReduce tasks. The proposed task scheduler dynamically predicts the performance of concurrent MapReduce jobs and adjusts the resource allocation for the jobs. It allows applications to meet their performance objectives without over-provisioning of physical resources.**

***Index Terms*—MapReduce, Performance management, Task scheduling**

I. Introduction

MapReduce [1] is a framework originally designed by Google to exploit large clusters to perform parallel computations. It is based on an implicit parallel programming model that provides an easy and convenient way to express certain kinds of distributed computations, particularly those that process large data sets. The framework is composed of an execution runtime and a distributed file system that helps with the distribution of tasks and data across nodes. The runtime and the distributed file system also provide fault tolerance and reliability, which are crucial in such a large-scale environment.

The MapReduce runtime consists of a single master process and a large number of slave processes. When a MapReduce application (or 'job') is submitted to the runtime, it is split into a large number of Map and Reduce tasks, which are executed by the slave nodes. The runtime is responsible for dispatching tasks to slave nodes and ensuring their completion.

While MapReduce was originally used primarily for batch data processing, it is now also being used in shared, multi-user environments in which submitted jobs may have completely different priorities: from small, almost interactive, executions, to very long programs that take hours to complete. This change makes task scheduling, which is responsible for selecting tasks for execution across multiple jobs, even more relevant. Task selection and slave node assignment govern a job's opportunity to make progress, and thus influences job performance.

In this paper we leverage Hadoop [2], a state of the art MapReduce runtime and the supporting distributed file system, to implement an application-centric task scheduler. The proposed scheduler relies on estimates of individual job completion times given a particular resource allocation, and uses these estimates so as to maximize each job's chances of meeting its performance goal.

The remainder of the paper is structured as follows. Section II introduces some technical background necessary to understand MapReduce and Hadoop. Section III introduces several techniques for estimating job completion time, and proposes the performance-based scheduling technique. Section IV presents the architecture of the scheduler and its integration into Hadoop. Section V presents the experiments that support the contribution of our work. Finally, Section VI shows the related work and Section VII concludes the paper.

II. Technical background

A. MapReduce

When implementing a MapReduce program, the programmer has to implement only two functions: *map()*, which processes fragments of input data to produce intermediate results, and *reduce()*, which combines the intermediate results to produce the final output. Each map input is a key-value pair (with types defined by the programmer) that identifies a piece of work. The output of each map is an intermediate result also expressed as a key-value pair (also defined by the programmer). The reduce input is composed of all the intermediate values identified by the same key; therefore, the reduce function can combine them to form the final result.

All nodes in the cluster execute the same function on different chunks of input data. The MapReduce runtime distributes and balances work across the nodes, dividing the input data into chunks, assigning a new chunk when a node becomes idle, and collecting the results.

There are many runtime implementations of this model in various environments. In this paper we present a prototype that is implemented on top of Hadoop [2], an open source MapReduce framework discussed in the following section.

B. Hadoop

Hadoop [2] is an open source MapReduce runtime provided by the Apache Software Foundation. It uses the Hadoop Distributed File System [3] (HDFS) as shared storage, enabling data to be shared among distributed processes using files. The HDFS implementation has a master/slave architecture: the master process ('NameNode') manages the global name space and controls operations on files, while a slave process ('DataNode') performs operations on blocks stored locally, following instructions from the NameNode.

The Hadoop runtime consists of two types of processes: 'JobTracker' and 'TaskTracker'. The singleton JobTracker partitions the input data into splits using a splitting method defined by the programmer, populates a local task-queue based on the number of obtained splits, and distributes work to the TaskTrackers that in turn process the splits. If a TaskTracker becomes idle, the JobTracker picks a new task from its queue to feed it. Thus, the granularity of the splits has considerable influence on the balancing capability of the scheduler. Another consideration is the location of the data blocks, as the JobTracker tries to minimize the number of remote blocks accessed by each TaskTracker.

Each TaskTracker controls the execution of tasks on a node. It receives a split descriptor from the JobTracker, divides the split data into records (through the 'RecordReader' component), and spawns a new worker process that actually processes all the records in the split. Such worker process will run a so-called Map task. The TaskTracker will also be in charge to run the so-called Reduce tasks as soon as they can be initiated. Notice here that a Map task will evetually result in the execution of a *map()* function, and that a Reduce task will behave anlogously with *reduce()* functions. The programmer can also decide how many simultaneous *map()* and *reduce()* functions can be run concurrently on a node. When a TaskTracker finishes processing a split and is ready to receive a new one, it contacts the JobTracker.

The execution of a job is divided into a Map phase and a Reduce phase. In the Map phase, the Map tasks of the job are run. Each Map task comprises the execution of the actual *map()* function as well as some supporting actions (for example, data sorting). The data being output by each Map task is written to a circular memory buffer. As soon as this buffer reaches a threshold, its content is sorted by key and flushed to a temporary disk file. If a Map task generates more than one such file, they are merged into a single file and then served via HTTP to nodes running Reduce tasks.

During the Reduce phase, Reduce tasks are run. Reduce tasks are divided into three sub-phases: copy, sort and reduce, which makes it possible to allocate resources to Reduce tasks earlier. The first sub-phase (copy) can run whilst Map tasks are still executing, fetching the partial map results as they are completed. This is beneficial as Reduce tasks will then already have most of the data when the last map finishes, and also because it helps to balance the load on the network. The reduce sub-phase, which runs the actual *reduce()* function, is able to start after the map outputs that pertain to this particular reducer have been copied and sorted, and the final result is then written to the distributed file system.

III. Performance-Driven Co-Scheduling

In this section we present the design, operation and objectives of the task scheduling mechanism introduced in this paper. The mechanism consists of two components: completion time estimation, and task co-scheduling strategy.

A. Design goals

The main goal of the task scheduling mechanism presented in this paper is to enable a MapReduce runtime to dynamically allocate resources in a cluster of machines based on the observed progress rate achieved by the jobs, and the completion time goal associated with each job. A necessary component of such a system is an estimator that maps the resource allocation for a job to its expected completion time. Such an estimator may easily be obtained if information about the total amount of computation to be performed by a job is known in advance. One way to provide this information would be to derive it from prior executions of the job. However, this approach is neither practical, as we cannot guarantee that prior executions of the job exist, nor accurate, as prior executions of the job were likely performed over a different data set and may therefore have completely different characteristics.

In this paper, we propose a technique to dynamically estimate the completion time of a job during its execution. In doing so, we take advantage of the fact that MapReduce jobs are a collection of a large number of smaller tasks. Specifically, we hypothesize that from a subset of tasks that have completed thus far, we can predict the properties of remaining tasks. We acknowledge the fact that MapReduce tasks vary widely in their execution characteristics depending on the data set they process. Hence, we do not expect this estimation technique to provide accurate predictions all the time, but we do expect that when combined with dynamic scheduling it will permit fair management of completion times of multiple jobs.

The technique targets a highly dynamic environment, such as that described in [4], in which new jobs can be submitted at any time, and in which MapReduce workloads share physical resources with other workloads, either MapReduce or not. Thus, the actual amount of resources available for MapReduce applications can vary over time. The dynamic scheduler introduced in this paper uses the completion time estimate for each job given a particular resource allocation to adjust the resource allocation to all jobs. The minimum unit of resource allocation is the *slot*, which corresponds to a worker process created by a TaskTracker.

The scheduler presented here can be considered pre-emptive in that it can interrupt the progress of one job in order to allocate all of its resources to other jobs with higher priority; but it does not interrupt tasks that are already executing. Interrupting executing tasks could be beneficial in the case of reduce tasks with a long copy phase, an issue that is part of our future work. In the current system, when a job reaches

the system and a task for this job is scheduled to start, in the event that all the slots are occupied, the job has to wait for at least one task to complete before being started.

B. Job performance estimation

We are given a set of jobs M to be run on a MapReduce cluster. Each job m is associated with a completion time goal, T^m_{goal}. The Hadoop cluster includes a set of TaskTrackers TT, each TaskTracker (TT_t) offering a number of execution slots, s_t, which can host a task belonging to any job. A job (m) is composed of a set of tasks. Each task (t^m_i) takes time α^m_i to be completed, and requires one slot to execute. The set of tasks for a given job m can be divided into tasks already completed (C_m), not yet started (U_m), and currently running (R_m). We also use $C_{m,t}$ to denote the set of tasks of job m already completed by TaskTracker t.

Let μ_m be the mean completed task length observed for any running job m:

$$\mu_m = \frac{\sum_{i \subset C_m} \alpha^m_i}{|C_m|} \tag{1}$$

Let μ^m_t be the mean completion time for any task belonging to a job m and being run on a TaskTracker TT_t:

$$\mu^t_m = \frac{\sum_{t \subset TT, i \subset C_{m,t}} \alpha^m_i}{|C_{m,t}|} \tag{2}$$

Notice that as the TaskTrackers are not necessarily identical, in general $\mu_m \neq \mu^t_m$. When implementing a task scheduler which leverages a job completion time estimator, both μ_m and μ^t_m should be considered. However, in the work presented in this paper, only μ_m is considered, i.e., all μ^t_ms are presumed equal. Three reasons have motivated this decision: 1) a design goal is to keep the task scheduler simple, and therefore all slots are considered identical. Under this assumption, estimating the resource allocation required by each job given its completion time goal is an easy task that can be performed with cost $O(M)$. If the differences between TaskTrackers are taken into account, the cost of making the best task allocation for multiple jobs can easily grow to be exponential. 2) the scenario in which task scheduling occurs is highly dynamic, and thus the task scheduling and completion time estimate for each job refreshed every few minutes. Therefore, a highly accurate prediction provides little help when scheduling of tasks in a scenario in which external factors can change the execution conditions over time. The approach is focused on dynamically driving the slot allocation to different jobs under changing conditions; and 3) the completion time estimate for a job m can only benefit from having information relative to a particular TaskTracker if at least one task that belongs to the job has been scheduled in that TaskTracker. In practice, it is likely that each job will have had tasks scheduled on only a small fraction of the total TaskTrackers.

For any currently executing (on-the-fly) task t^m_i we define β^m_i as the task's elapsed execution time, and δ^m_i as the remaining task execution time. Notice that $\alpha^m_i = \beta^m_i + \delta^m_i$, and that δ_i and α^m_i are unknown. Our completion time estimation technique relies on the assumption that, for each on-the-fly task t^m_i, the observed task length α^m_i will satisfy $\alpha^m_i = \mu_m$, and therefore $\delta^m_i = \mu_m - \beta^m_i$.

C. Task Scheduling

The task scheduler presented in this work consists of two components: a scheduling policy that assigns a priority to each job, and an allocation algorithm that assigns free slots to jobs based on their priority. Jobs are organized in a ordered queue based on their priority.

The priority of each job is calculated based on the number of slots to be allocated concurrently to each job over time so that it can make its goal. For such purpose, our technique needs to estimate the amount of work still pending for each job, assuming that each allocated slot will be used for time μ_m. Such estimation needs to consider both the tasks that are in the task queue waiting to be started, as well as the tasks currently in execution. Based on these two parameters, we propose that the number s^m_{req} of slots to be allocated in parallel to this job can be estimated as:

$$s^m_{req} = \frac{(\frac{\sum_{i \subset R_m} \delta^m_i}{\mu_m} + |U_m|) * \mu_m}{T^m_{goal} - T_{curr}} - |R_m| \tag{3}$$

where T^m_{goal} is the completion time goal for the job m, and T_{curr} is the current time. Therefore, the order in the task list is defined by the value s^m_{req} dynamically calculated for each job. Notice that s^m_{req} is a real value, and it is unlikely that two jobs have equal s^m_{req} values at a given moment. If that does occur, the jobs will be differentiated arbitrarily.

The scheduling policy must consider some special cases. Immediately after a job is submitted, there is no data available and it is not possible to estimate the required slots or the completion time. Therefore, jobs with no completed or running tasks always take precedence over other jobs. If there is more than one such job, the oldest one comes first. There is another scenario that requires special attention: jobs that have already missed their deadline. For such a job, the scheduler tries to at least complete it as soon as possible, prioritizing it over any other kind of job, which helps avoid job starvation. In summary, the priority of the job is calculated as follows: First, jobs that have already missed the deadline. Second, recently submitted jobs for which there is no available data. Finally, executing jobs based on their s^m_{req}.

The second component of the task scheduler, the scheduling algorithm, assigns available task slots to jobs according to their priority. Notice that in the event that several jobs have the same priority, one of them is chosen arbitrarily. This is not a problem since once a slot is allocated to one of these jobs, its priority will decrease, and next slot will be allocated to one of the other jobs that previously had the same priority. When two jobs that have already missed their deadlines are competing for resources, the scheduler fairly equalizes their expected completion times with respect to their goals.

Although most of the behavior of the scheduler is defined by the scheduling policy, it can also significantly contribute to the execution progress made by the jobs. In particular, the scheduler must decide what to do with slots that are *not needed*: that is, the case in which:

$$\sum_{m \subset M} s^m_{req} < \sum_{t \subset TT} s_t \quad (4)$$

In this situation, the scheduler may decide to keep the extra slots free, or to allocate them to jobs currently in the system. The first case can be useful in a Hadoop cluster provisioned on demand, wherein decommissioning nodes that are not required can result in freeing resources and therefore in a reduction in the Total Cost of Ownership (TCO) of the cluster [5]. The second case is useful in static Hadoop clusters, where making as much progress as possible on current jobs leaves more resources available for future jobs. As both cases are interesting, we have implemented two versions of the scheduler. The *min-scheduler* allocates a maximum of s^m_{req} slots to job m . The *max-scheduler* allocates excess slots to jobs with the highest priority. Recall that priorities are updated after each allocation, so the process does not necessarily assign all the excess slots to the same job.

D. About Mappers and Reducers

The discussion in the previous sections applies to both Map and Reduce tasks. Each TaskTracker has a number of slots for Map tasks and another number of slots for Reduce tasks. The usual case for a MapReduce application is that execution time is dominated by the time required to complete the Map tasks, but cases where the Reduce tasks dominate can also occur. In both cases, jobs start with a Map phase, in which performance data is collected, and is followed by the Reduce phase.

The scheduler cannot make assumptions about the Reduce phase before seeing some Reduce tasks completing. Therefore, in the absence of information from previous runs, the scheduler needs to estimate the effort of the Reduce phase compared to the Map phase, such that overall job execution time can be predicted before the Reduce phase starts. In our system, a user can use the configuration files of Hadoop to provide an estimate of the relative cost of a Reduce task compared to that of a Map task. If no estimate is provided, the scheduler assumes that the computational cost of a Map task is the same as that of a Reduce task. As the number of Map and Reduce tasks to complete for a job is known in advance (when the input is split), the scheduler can estimate the cost of the Reduce phase once the Map phase is started. All the experiments presented in this paper assume the Map phase dominates the computational cost of the jobs.

IV. System Prototype

The scheduler in Hadoop is part of the JobTracker and is triggered by heartbeats sent from the TaskTrackers. These heartbeats include status information such as the TaskTracker's number of free slots, and give the scheduler the data necessary to schedule tasks. In response to the heartbeat, the scheduler returns a number of tasks to be executed by the TaskTracker that originated it.

Hadoop provides an interface that schedulers implement, thereby permitting alternate schedule implementations. This interface primarily consists of a function that takes the status of a TaskTracker, and returns a list of tasks to be executed.

Our scheduler maintains a priority queue of tasks as described in Section III-C. Each job for which a task has to be scheduled is in one of the following states: NODATA for jobs with no completed tasks, for which there is not yet enough data to estimate the requirements of these jobs; ADJUST for jobs for which data is available ans so the scheduler can decide whether more slots or fewer slots are required; UNDEAD for jobs that have missed their deadline.

The scheduler also determines $s^m_{alloc,i}$, the number of slots that should be allocated to each job m until the next run of the scheduler. For jobs in the NODATA and UNDEAD state, the maximum number of allocatable slots is the same as the number of remaining tasks ($|U_m|$), while for ADJUST jobs it is based on the needs to meet the deadline (s^m_{req}).

As mentioned in Section III-C, we implemented two versions of the scheduler: *min-scheduler* and *max-scheduler*. The *min-scheduler* processes the queue once, running at most $s^m_{alloc,i}$ tasks concurrently for each job. The *max-scheduler*, however, wishes to allocate all available slots. Therefore, it processes the queue multiple times until all slots are allocated.

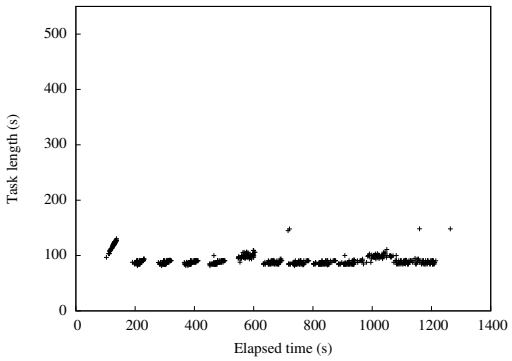
The current implementation updates the priority queue on every call to the scheduler, which has a cost of O($n \log n$), where n is the number of running jobs. This has proven adequate for testing purposes and keeps the prototype simple. However, as the queue may not change much between updates, and the number of available slots is usually small, this approach results in unnecessary work. We plan to improve efficiency by updating the queue in a background process.

The scheduler presented in this paper can be found at [6].

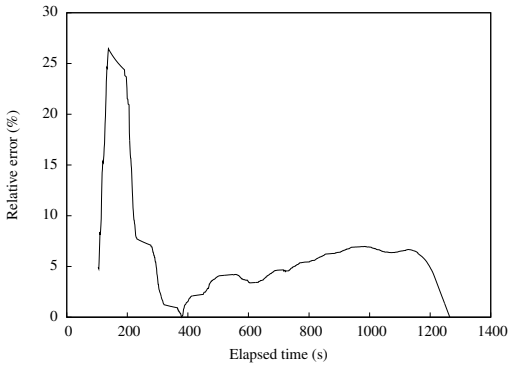
V. Experiments

In this section we present four experiments carried out in order to investigate the effectiveness of the job completion estimation technique, as well as to evaluate the task scheduling techniques we have implemented in the Hadoop JobTracker. In a first experiment, we run three representative MapReduce applications in isolation to evaluate the efficacy of the completion time prediction technique described in Section III-B. In a second experiment we present an illustrative scenario in which four applications with completion time goals compete for resources, managed by the system prototype presented in Section IV and using the *min-scheduler* described in Section III-C. In a third experiment, we use the *max-scheduler* in the same scenario. Finally, in a fourth experiment, we show the behavior of Hadoop's FairScheduler [7] in the same scenario.

For our experiments, we created a Hadoop cluster consisting of 61 2-way 2.1Ghz PPC970FX nodes with 4GB of RAM. All nodes were connected using gigabit ethernet and were running Hadoop 0.21-dev on IBM JDK 1.6 on a 64-bit PPC 2.6.16 Linux kernel. One of the nodes was configured to be both the

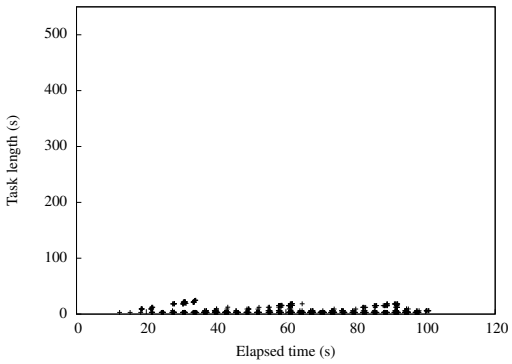


(a) Task completion time

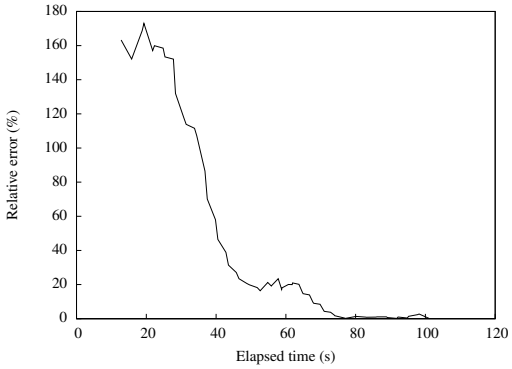


(b) Relative error of the estimation

Fig. 1. Completion time estimation: WordCount

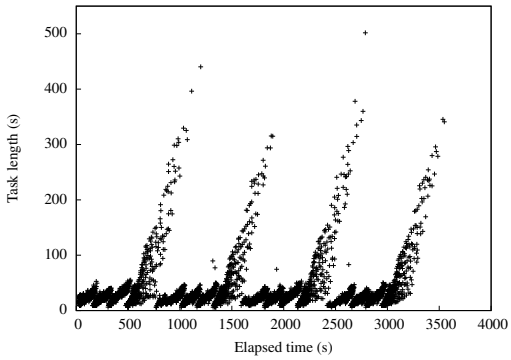


(a) Task completion time

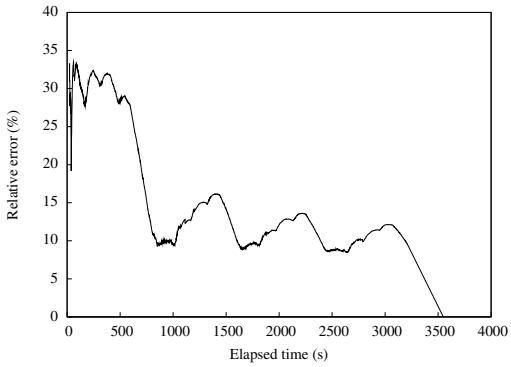


(b) Relative error of the estimation

Fig. 2. Completion time estimation: Join



(a) Task completion time



(b) Relative error of the estimation

Fig. 3. Completion time estimation: Simulator

JobTracker and the NameNode, and the 60 remaining nodes were used as DataNodes and TaskTrackers. Each TaskTracker was configured to run a maximum of one task in parallel (one slot for Map tasks and one for Reduce tasks), so one core was available for the TaskTracker and the DataNode processes, and the other was available for actual computation.

A. Applications

To evaluate the job estimation technique we use three MapReduce applications, one that has regular distributions of task lengths (each task takes approximately the same amount of time to execute), and two with an irregular distribution. Specifically, the applications used are:

- WordCount: The 'WordCount' application is one of the sample applications contained in the Hadoop distribution. It takes a set of text files as input, and counts the number of times that each word appears. This is a case of application that exhibits regular distribution of task lengths.
- Join: The 'Join' application joins two data tables on a given column. In this implementation, the Map tasks each process a split of one of the two data tables—for each record in the split, the mappers emit *key,value* where the key is the join key and the value is the record (tagged to indicate which of the two tables it came from). The Reduce tasks then separate the input records according to the tag, and perform a cross-product on the resulting two sets of records. This application is irregular in its distribution of task lengths.

- Simulator: The 'Simulator' application is an execution harness for the placement algorithm described in [4]. By varying the numbers of nodes and applications, in addition to the memory and CPU capacities of the nodes and demands of the applications, the algorithm can be made to execute for different lengths of time. This application shows irregular distribution of task lengths.

In the four experiments presented in the following sections, we first study the characteristics of each of the presented in this section, to later use a synthetic mix of them to compose a realistic scenario of a MapReduce execution environment. The application set is composed of 4 different MapReduce applications that share resources during their execution. We configure each application with a particular completion time goal, which is derived from the completion time goal that each applications exhibits when is run in isolation. The scenario is realistic in terms of the mix of applications, as well as in terms of setting realistic completion time goals so that can be met by all the submitted applications.

The set of applications selected is composed of big Simulator job (J1), which is configured to have a relaxed completion time goal of 6,000s; a WordCount job (J2) configured with a completion time goal of 3,000s; and two identical runs of the Join application (J3 and J4), each with a completion time goal of 150s. Notice that the completion time goals set for each application compared to their length observed when running in isolation represent a factor of 1.69X (3,549s vs 6,000s) for the Simulator, 2.37X (1,264s vs 3,000s) for WordCount and 1.18X (101s vs 120s) for both Joins.

B. Experiment One: Job completion estimation efficacy

The aim of the first experiment is to evaluate the effectiveness of the job completion time prediction technique, as well as to study the behaviour of the applications used in our experiments. We use the three applications presented above: for each application, we show the task length distribution during an isolated execution. We also show the relative error of the prediction technique's estimate over time compared to the job's actual completion time. As has been discussed before, the goal of the technique is to drive the resource allocation and not to make accurate predictions, although this is a desirable property when the behavior of the applications is predictable. Note that in this case we do not perform any management, we simply examine the behaviour of the prediction technique.

Figures 1(a) and 1(b) show the task length distribution and the relative error of the estimation over time respectively for the WordCount application. Notice that Figure 1(a) shows the actual completion time for each task of the job once the task has been completed. It can be observed that WordCount demonstrates regular task length, centered in the range of 80s – 100s. A few outliers are seen, resulting from some issues related to the runtime, but these do not impact the result of the experiment. WordCount splits the input into HDFS blocks, each of which is processed by a single TaskTracker. The input used here was a 43GB set of text files, composed of several copies of a collection of plain-text books. This resulted in 738 maps and 1 reduce being executed across the 60 available slots. Looking at Figure 1(b), it can be observed that the relative error is below 5% for most of the job execution time. The prediction is initially quite pessimistic, because the set of tasks completed in the first 100s of the experiment take longer (probably due to initialization issues) than the other tasks yet to be started. Once shorter tasks start completing, the regular pattern of the task length distribution allows the system to produce a much more accurate completion time estimate.

Figures 2(a) and 2(b) show the task length distribution and the relative error of the estimation over time respectively for the Join application. This application exhibits irregular task length, with most tasks finishing in less than 25s but showing a high variability between them. The application was configured to perform a join operation between two data tables, and executed 1,000 map and 4 reduce tasks. Looking at Figure 2(b), it can be observed that the behaviour is similar to that observed for the WordCount application, but with more extreme values: there is an initial phase in which the estimation is extremely inaccurate, but as the execution progresses, the estimation becomes more accurate, particularly during the second half of the experiment, once a representative number of samples of different task lengths has been observed.

Figures 3(a) and 3(b) show the task length distribution and the relative error of the estimation over time respectively for the Simulator application. This application also exhibits a completely irregular task length distribution, as it depends on the difficulty of the problem posed to the simulation itself. Each map task, in this case, corresponds to to a single simulation execution. The experiment resulted in the execution of 6,400 map tasks, and 1 reduce task. Figure 3(a) shows the high variability of task length. Further, the four phases that can be seen demonstrate the order in which the simulation problem space is explored. The relative error for this application is high, and it is not until the end of the execution that the error drops below 10%. This is a clear example of an unpredictable application (unless information is kept between executions), and therefore a best-effort approach is followed for such an scenario.

C. Experiment Two: Jobs with deadlines, min-scheduler

In the second experiment we use the synthetic set of applications presented in Section V-A to evaluate the behavior of the *min-scheduler* under realistic conditions. Figure 4 shows the number of slots allocated to each application over time. Jobs J1, J2, J3 and J4 are submitted at times S1, S2, S3 and S4 respectively, and the completion time goals are D1, D2, D3 and D4 respectively.

Simulator (J1) is the first job submitted (at S1). As we are here using the *min-scheduler*, and the estimation for J1 is initially optimistic (as discussed previously in Section V-B), the system only allocates around 20 slots to this job. Notice also that at time 0, the scheduler has no data about this job and it is therefore a high priority task. Consequently, an initial burst of up to 60 slots is allocated. Some time later, WordCount (J2) is submitted (at S2), and starts sharing resources with

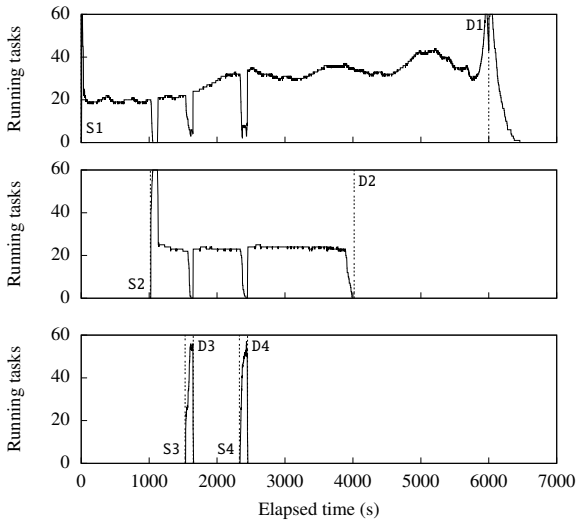


Fig. 4. Experiment Two: Allocated Resources (slots) using the min-scheduler

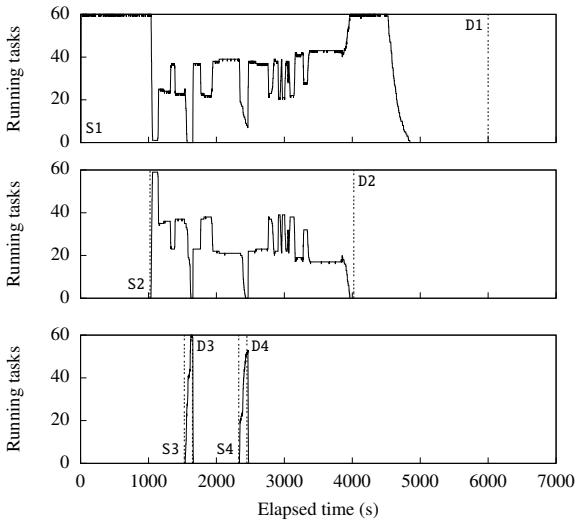


Fig. 5. Experiment Three: Allocated Resources (slots) using max-scheduler

the Simulator, showing also an initial burst at which all slots are committed to WordCount. As the prediction for Simulator starts being more realistic (see Figure 3(b)), the Simulator starts getting a higher allocation. Later, a short and high priority Join (J3) is submitted (at S3), and is allocated most of the resources in order to meet its goal. When the J3 is finished, the resources it was using are returned to J1 and J2. A second instance of Join (J4) is submitted (at S4), and again is assigned most of the resources. J2 completes close to its goal, and finally J1, which suffers from irregular task length distribution, misses its goal due to the fact that the final stage of simulation involves long simulations. Observe that J1 could have met its goal if had made more progress at the beginning of its execution, exactly what happens when using *max-scheduler*, as will be shown in the next experiment.

D. Experiment Three: Jobs with deadlines, max-scheduler

In the third experiment, the same jobs, submission times, and deadlines are used, but we use *max-scheduler* allocating resources. The first obvious difference with the second experiment is the amount of slots that are allocated to J1 at the beginning of the test–every slot is allocated to it, as there is nothing else in the system. When J2 is submitted (at S2), it shares resources with J1, and together they use every slot in the system. The behavior of the system when the two instances of the Join application are submitted does not differ from what was observed for the *min-scheduler*: they get most of the resources until they complete. When J2 finishes, close to its goal again, J1 is once again allocated the entire system, and in this case it meets its goal.

E. Experiment Four: Comparison with the FairScheduler

In the fourth experiment, we use the well-known FairScheduler [7], which uses job priorities to make scheduling decisions, in place of our completion time goal oriented scheduler. We perform two tests: in the first, presented in Figure 6 all the jobs are configured with the same weight; in the second, presented in Figure 7, we estimate a weight for each job based on its deadline and minimum completion time when run in isolation so that the behavior of the FairScheduler is as close as possible to the behavior of our proposed scheduler. Notice that the only possible comparison is with *max-scheduler*, since the FairScheduler does not consider the possibility of not allocating all the slots while jobs remain unfinished.

Looking at the results, it can be seen that when no weights are set for the jobs, high-priority Join jobs miss their goal (as expected). Notice also that the way in which slots are allocated to the different jobs follows a Round Robin approach when resource sharing between jobs occurs. When different weights are set to different jobs, the FairScheduler is able to mimic the behavior of *max-scheduler*. In order to achieve this behavior, the jobs were organized in three different applications pools, with the Simulator (J1) pool having a weight of 1, the WordCount (J2) pool 1.4, and the Join (J3 and J4) pool 122. These weights were derived from the difference between the time required to complete each job in isolation and its established completion time goal.

This experiment shows that while the behavior of the *max-scheduler* can be obtained when using FairScheduler, the mapping between completion time goals and job weights is not obvious, and the system cannot be configured directly based on high-level performance objectives. Also, it is not possible to achieve the behavior of the *min-scheduler*, which is especially well-suited for emerging MapReduce on demand clusters.

VI. Related Work

Process scheduling is a deeply explored topic for parallel applications, considering different type of applications, different scheduling goals and different platforms architecture ([8], [9], [10]). However, there is little work on scheduling for MapReduce applications. The initial scheduler presented in the Hadoop distribution uses a very simple FIFO policy, considering five different application priorities. In addition, in order to isolate the performance of different jobs, the Hadoop

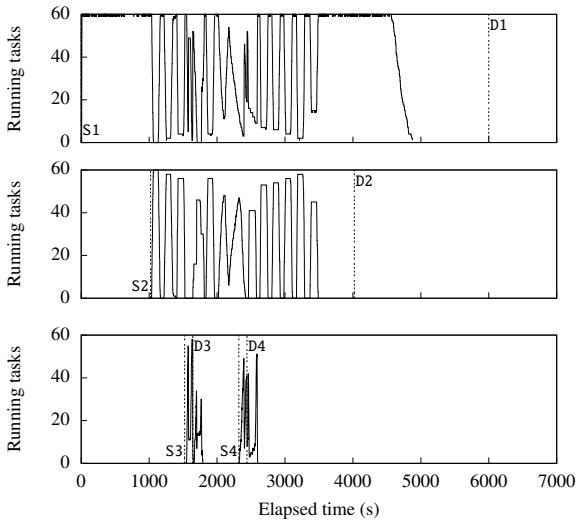


Fig. 6. Experiment Four: Allocated Resources (slots) – Fair Scheduler without weights

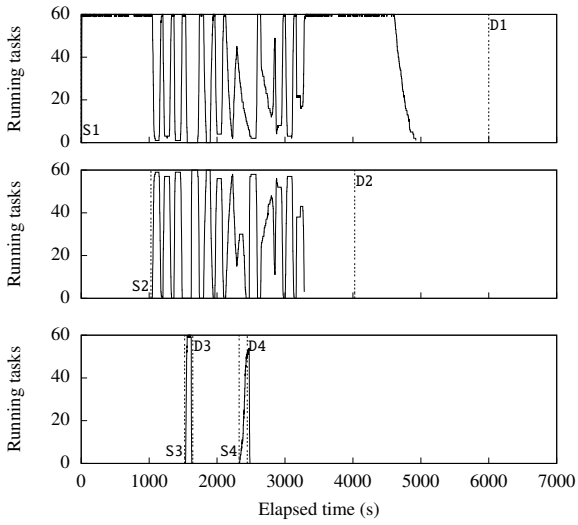


Fig. 7. Experiment Four: Allocated Resources (slots) – Fair Scheduler with weights

project is working on a system for provisioning dedicated Hadoop clusters to applications [11]. However this approach can result in resource underutilization. In [12] the authors propose a fair scheduling implementation to manage data-intensive and interactive MapReduce applications executed on very large clusters. The main concern of this scheduling policy is to give equal shares to each user. In addition, as exploiting data locality is a must, it tries to execute each task near the data it uses. However, this approach is not appropriate for long-running applications with different performance goals. In addition, scheduling decisions are not dynamically adapted based on job progress.

VII. Conclusions and Future Work

In this paper we have presented a prototype of a task scheduler for MapReduce applications. It has been implemented on top of Hadoop, the Apache's open-source implementation of a MapReduce framework. The scheduler dynamically estimates the completion time for each MapReduce job in the system, taking advantage of the fact that each MapReduce job is composed of a large number of tasks (maps and reduces) known in advance during the job initialization phase (when the input data is split), and that the progress of the job can be observed at runtime. The scheduler takes each submitted and not yet completed Hadoop job and monitors the average task length for already completed tasks. This information is used to predict the job completion time. Based on these estimates, the scheduler is able to dynamically adapt the number of task slots that each job is allocated. In this way, we introduce runtime performance management in the Hadoop MapReduce framework. To our knowledge, this is the first scheduler for a MapReduce runtime that is able to manage the performance of the MapReduce applications based on high-level policies. The scheduler can use two strategies to allocate resources: the *max-scheduler* approach, in which all the resources of the MapReduce cluster are allocated when there are enough tasks; and the *min-scheduler* approach, in which although the completion time goal for each job is carefully observed, resources are freed if possible, which is an interesting approach when the new generation of dynamic MapReduce clusters are used. We plan to add consideration of other high-level objectives as well as to integrate the scheduler in the the EmotiveCloud [13] middleware.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI '04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004, pp. 137–150. [Online]. Available: http://labs.google.com/papers/mapreduce.html

[2] Apache Software Foundation. Hadoop map/reduce tutorial. [Online]. Available: http://hadoop.apache.org

[3] ——. Hdfs architecture. [Online]. Available: http://hadoop.apache.org/core/docs/current/hdfs_design.html

[4] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé, "Enabling resource sharing between transactional and batch workloads using dynamic application placement," in *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008, pp. 203–222.

[5] C. K. Jacob Leverich, "On the energy (in)efficiency of hadoop clusters," in *HotPower'09*, 2009.

[6] J. Polo, "Deadline scheduler for hadoop," 2009. [Online]. Available: http://www.bsc.es/autonomic

[7] M. Zaharia, "Hadoop Fair Scheduler Design Document," 2009. [Online]. Available: http://developer.yahoo.net/blogs/hadoop/FairSharePres.ppt

[8] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," in *JSSPP*, 1995, pp. 1–18.

[9] D. G. Feitelson, "Job scheduling in multiprogrammed parallel systems," Tech. Rep. RC 19790 (87657), August 1997.

[10] C. E. Volker, V. Hamscher, and R. Yahyapour, "Economic scheduling in grid computing," in *Scheduling Strategies for Parallel Processing*. Springer, 2002, pp. 128–152.

[11] Apache Software Foundation. Hadoop on demand. [Online]. Available: http://hadoop.apache.org/core/docs/r0.20.0/hod_user_guide.html

[12] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html

[13] Autonomic Systems and eBusiness Platforms research line. Barcelona Supercomputing Center (BSC). Emotive cloud. [Online]. Available: http://www.emotivecloud.net