Performance Management of Accelerated MapReduce Workloads in Heterogeneous Clusters

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

***Abstract*—Next generation data centers will be composed of thousands of hybrid systems in an attempt to increase overall cluster performance and to minimize energy consumption. New programming models, such as MapReduce, specifically designed to make the most of very large infrastructures will be leveraged to develop massively distributed services. At the same time, data centers will bring an unprecedented degree of workload consolidation, hosting in the same infrastructure distributed services from many different users. In this paper we present our advancements in leveraging the Adaptive MapReduce Scheduler to meet user defined high level performance goals while transparently and efficiently exploiting the capabilities of hybrid systems. While the Adaptive Scheduler was already able to dynamically allocate resources to co-located MapReduce jobs based on their completion time goals, it was completely unaware of specific hardware capabilities. In our work we describe the changes introduced in the Adaptive Scheduler to enable it with hardware awareness and with the ability to co-schedule accelerable and non-accelerable jobs on the same heterogeneous MapReduce cluster, making the most of the underlying hybrid systems. The developed prototype is tested in a cluster of Cell/BE blades and relies on the use of accelerated and non-accelerated versions of the MapReduce tasks of different deployed applications to dynamically select the best version to run on each node. Decisions are made after workload composition and jobs' completion time goals. Results show that the augmented Adaptive Scheduler provides dynamic resource allocation across jobs, hardware affinity when possible, and is even able to spread jobs' tasks across accelerated and non-accelerated nodes in order to meet performance goals in extreme conditions. To our knowledge this is the first MapReduce scheduler and prototype that is able to manage high-level performance goals even in presence of hybrid systems and accelerable jobs.**

***Keywords*—MapReduce, scheduling, accelerators, performance management, heterogeneity**

I. Introduction

Cloud computing has dramatically transformed the way many critical services are delivered to customers (i.e. the Software, Platform and Infrastructure as a service paradigms), and at the same time has posed new challenges to data center infrastructures. The result is a complete new generation of large scale infrastructures, bringing an unprecedented level of workload and server consolidation, that demand new programming models, management techniques and hardware platforms. At the same time, cloud computing has made available mainstream an unprecedented computing capacity that opens a new range of opportunities to build new services that require large scale computing. Therefore, data analytics is one of the more prominent fields that can benefit from next generation data center computing.

The intersection between cloud computing and next generation data analytics services [1] depicts a future scenario in which amassed data will be made available mainstream, and users will be able to process it to create added value services. Consequently, building new models to develop such applications and mechanisms to manage them are open challenges currently not yet addressed. An example of programming model especially well suited for large scale data analytics is MapReduce [2], introduced by Google back in 2004.

MapReduce workloads usually involve a very large number of small computations executing in parallel. High levels of computation partitioning and a relatively small size of individual tasks are a design point of these platforms. In that aspect, MapReduce workloads are closer to online web workloads than to HPC jobs. While MapReduce was originally used primarily for batch data processing, its use has been extended to shared, multi-user environments in which submitted jobs may have completely different priorities. 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. Such properties were already leveraged in the Adaptive MapReduce Scheduler [3].

At the same time, there is a clear trend towards the adoption of heterogeneous hardware [5] and hybrid systems [6] in the computing industry, due to its superior performance and energy efficiency to run specialized workloads. Heterogeneous hardware (mixing generic processors with accelerator cores such as GPUs or the SPUs in the Cell/BE [7] processor) will be leveraged to improve both performance and energy consumption, making the best of each specific platform. For example, a MapReduce framework enabled to run on hybrid systems, as the one introduced in [8], has the potential to impact immensely upon the future of many fields such as financial analysis, healthcare and smart cities management. The MapReduce framework and the domain-specific languages built upon it, provide an easy and convenient way to develop massively distributed data analytics services that exploit all the computing power of these large scale facilities, while low level generic languages, such as OpenCL [9], will provide portability across different hardware platforms. Huge clusters

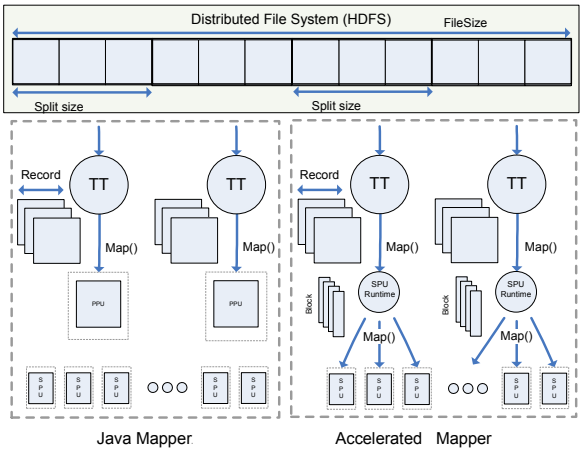


Fig. 1. Architecture of the system: 2 levels of parallelism

of hybrid many-core servers will bring workload consolidation strategies one step forward, becoming the norm in future platforms.

In this paper we leverage the Adaptive MapReduce Scheduler and the Cell/BE processor to show how next generation software and hardware can be combined to meet high level performance goals of customers while transparently and efficiently managing software and hardware resources. Such an approach requires an integrated management of next generation data centers that addresses two critical goals: meeting high level performance goals for data analytics services, and exploiting the capabilities of heterogeneous hardware. While both challenges have been addressed in the past by separate (see [3] and [8] for more details) their integration represents a completely new challenge. Such integration is addressed in this paper. To our knowledge, this is the first MapReduce scheduler that is able to manage heterogeneous hardware while observing jobs' completion time goals.

The remainder of the paper is structured as follows: Section II introduces some technical background necessary to understand MapReduce, Cell/BE processor and the Adaptive Scheduler. Section III introduces the basic techniques used in the Adaptive Scheduler to estimate job completion time. In Section IV we describe the extensions required to enable the Adaptive Scheduler with hardware heterogeneity awareness. Section V presents the experiments that support our contributions. Finally, Section VI goes through the related work and Section VII concludes the paper.

II. Technical Background

A. MapReduce

MapReduce 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. Hadoop [10] is a state of the art MapReduce runtime and the supporting distributed file system (HDFS [11]) that helps with the distribution of 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. Hadoop runtime consists of a single master process (the JobTracker) and a large number of slave processes (TaskTrackers). HDFS uses a master process to manage data distribution (NameNode) and slave processes to host data locally in each node (DataNodes). 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 and, therefore, responsible for selecting tasks for execution across multiple jobs. Task selection and slave node assignment govern a job's opportunity to make progress, and thus influences job performance.

B. Hardware heterogeneity

Current research trends show that next generation data centers will contain a remarkable degree of heterogeneity of nodes (i.e. the RoadRunner [5] cluster, composed of Opterons and Cell/BE blades), in an attempt to improve data center power efficiency and performance. Such heterogeneity, involving generic and specialized processors co-existing in the same data center and performing differentiated tasks, hinders the efficient use of node resources in a convenient way. In this new scenario, the actual challenge is to exploit heterogeneous intra-node resources in a transparent way. For our particular experiments we pick the Cell/BE processor as an example of hybrid hardware.

The Cell BE architecture [12], has nine processing cores on a single chip: one 64-bit Power Processing Element (PPE core) and eight Synergistic Processing Elements (SPE cores) that use 18-bit addresses to access a 256K Local Store. The PPE core accesses system memory using a traditional cache-coherent memory hierarchy. The SPE cores access system memory via a DMA engine connected to a high bandwidth bus, relying on software to explicitly initiate DMA requests for data transfer. The DMA engine can support up to 16 concurrent requests of up to 16K, and bandwidth between the DMA engine and the bus is 8 bytes per cycle in each direction. Each SPE uses its Local Store to buffer data transferred to and from system memory. The bus interface allows issuing asynchronous DMA transfer requests, and provides synchronization calls to check or wait for previously issued DMA requests to complete.

C. Accessing Hardware accelerators from Hadoop

The work presented in this paper takes advantage of a system prototype that extends Hadoop runtime to access the capabilities of underlying hardware accelerators. Although such prototype was presented in [8], we include here a brief description of its main characteristics. The original prototype neither observed jobs' completion time goals nor modified Hadoop scheduling decisions in any way.

The architecture of the prototype has two main components, each one devoted to manage one of the parallelism levels (see

Figure 1). The first component is based on Hadoop and its purpose is to partition the data and to assign a piece of work to each node in the cluster. The second component implements a second level partition of the data, that is the intra-node distribution of the data, and executes the actual computing function. We have implemented this second component as a shared native library that allows us to divide and execute tasks on the SPUs. The processing routine executed by each node, that is, the map function executed by each TaskTracker, invokes the second component of our prototype, using Java Native Interface [13].

In previous work we developed this prototype to evaluate the benefits from exploiting the two levels of parallelism presented on clusters of hybrid processors. In this paper we go a step further by considering also heterogeneous clusters composed of both kinds of processors, with and without hardware acceleration support, and enabling a MapReduce runtime with a scheduler that considers these characteristics to dynamically decide which node is more suitable to host tasks for each application.

III. The Adaptive Scheduler

In this section we describe the Adaptive Scheduler, which is an application-centric multi-job task scheduler for MapReduce workloads. In our work, the Adaptive Scheduler is integrated with the prototype described in Section II, what results in an augmented version of the scheduler that is aware of the underlying hardware capabilities, to manage the existing hardware properly and to make hardware-aware scheduling decisions. The Adaptive Scheduler core is described in detail in [3], and downloadable from [14].

A. Original Design for Homogeneous Clusters

The Adaptive 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 main objective of the task scheduling mechanism 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. Recall that in its original form, the scheduler expected the cluster to be homogeneous, and was unable to distinguish between accelerator-enabled and regular nodes.

The scheduling 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, be it MapReduce or not. Thus, the actual amount of resources available for MapReduce applications can vary over time. The Adaptive Scheduler adjusts the resource allocation to all jobs, according to estimates on the completion time given a particular resource allocation. The minimum unit of resource allocation is the *slot*, which corresponds to a worker process created by a TaskTracker in a slave node. The goal of the estimation technique is not to provide accurate predictions all the time, but to permit fair management of completion times of multiple jobs when combined with a dynamic scheduling middleware.

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, denoted as $\mu_m = \frac{\sum_{i \in C_m} \alpha^m_i}{|C_m|}$. Let μ^t_m be the mean completion time for any task belonging to a job m and being run on a TaskTracker TT_t. 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 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 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, sorted by 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:

$$s^m_{req} = \frac{(\frac{\sum_{i \in R_m} \delta^m_i}{\mu_m} + |U_m|) \times \mu_m}{T^m_{goal} - T_{curr}} - |R_m| \quad (1)$$

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.

A more detailed description of how special cases are treated can be found in [3].

IV. Augmenting the Scheduler

In this Section we describe the changes introduced in the Adaptive Scheduler to exploit hardware accelerators in heterogeneous MapReduce clusters. The changes have required a redesign of the scheduling policy to make it more aware of the underlying hardware and thus favour the execution of accelerable mappers on machines with accelerators.

A. Adding Hardware Heterogeneity Awareness

Scheduling jobs that contain accelerable and non-accelerable MapReduce task implementations requires for the scheduler to keep track of different kinds of TaskTrackers depending on the hardware characteristics of the nodes in which they are running. Whenever a job is deployed with an accelerator-based version of its code and one of the tasks for that job is scheduled to run on an accelerator-enabled TaskTracker, the code that will run in that node will be the accelerator-specific version. Otherwise, regular non-accelerated Java version of the code will be run on the node.

TaskTrackers are configured and grouped into pools of machines with different features: TaskTrackers running on regular machines (non accelerator-enabled) are included in the regular pool P_{reg}, while TaskTrackers running on accelerated machines are included into the accelerated pool denoted by P_{acc}.These pools are used not only to favour the affinity and execute accelerable tasks on accelerator-enabled nodes, but also to detect whether new jobs may benefit from accelerators or not. During an initial *calibration* stage, right after jobs are submitted, the scheduler first makes sure to execute at least one map task on each pool. Then, as soon as these initial tasks are completed, the scheduler decides whether a job will benefit or not from running on machines from that pool by comparing the observed elapsed task times on accelerated TaskTrackers (μ^m_{acc}) with those obtained on the regular pool (μ^m_{reg}). Recall that some jobs that are I/O bound may not clearly benefit from task acceleration even if their code can be run on top of an accelerator. In that case, providing affinity to accelerator-enabled nodes would bring no benefit and could even result in competition for accelerators with other jobs that in fact could take advantage of them. Therefore, only in the case that the speedup obtained when running on one of the accelerated pools (μ^m_{reg}/μ^m_{acc}) passes a certain configurable threshold, the job will be marked as accelerable. Otherwise it will be marked as regular and will be preferably executed on P_{reg}. Accelerable jobs will preferably run on TaskTrackers from P_{acc}.

Other than detecting accelerable jobs, the scheduler itself still assigns resources depending on job priority, which is in turn primarily based on the need of task slots to meet the completion time goal (s^m_{req}). However, this estimation is now slightly different: for accelerable jobs, only the mean time of tasks executed on accelerator-enhanced hardware are observed:

$$s^m_{req,acc} = \frac{(\frac{\sum_{i \in R^m_{acc}} \delta^m_i}{\mu^m_{acc}} + |U^m_{acc}|) \times \mu^m_{acc}}{T^m_{goal} - T_{curr}} - |R^m_{acc}| \quad (2)$$

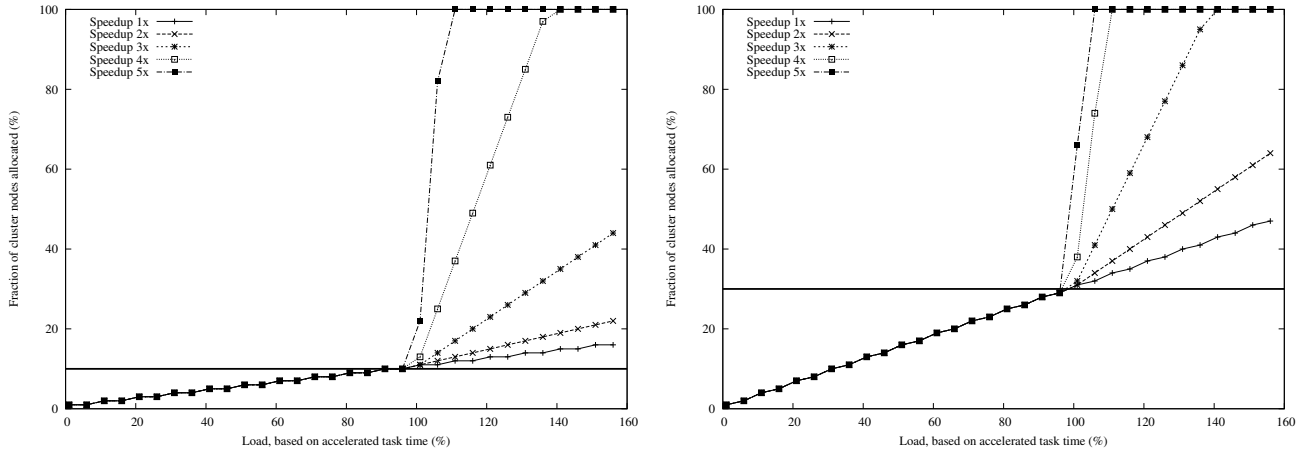
It should be noted, though, that this approach entails another issue: even though jobs can be easily prioritized within each pool, the scheduler still needs a global prioritization if there are not enough resources to meet the completion time goals. This is accomplished by normalizing the need of slots depending on the observed speedup as well as the capacity of the accelerated pool:

$$s^m_{req,extra} = (\frac{\mu^m_{reg}}{\mu^m_{acc}}) \times (s^m_{req,acc} - \textstyle\sum_{t \in P_{acc}} s_t) \quad (3)$$

For the sake of clarity, take for instance a cluster with 10 slots running on accelerator-enabled machines, and a job whose map tasks take $50s$ to complete on an accelerator and $100s$ on a regular machine. Since it is accelerable, if the job needs 10 or less slots to meet the completion time goal, the scheduler will try to schedule it on accelerators only. However, if the job is missing its goal and accelerable nodes are not enough, the number of required slots will change accordingly: an estimation of 15 accelerated slots to meet the completion goal will be normalized to 10 accelerated slots + 10 regular slots ($(100/50) \times (15 - 10)$).

This way accelerable jobs are prioritized over regular ones with a similar need of resources, but the latter are not excluded from running if they have a much tighter completion time goal. Similarly, if there are not enough resources to complete an accelerable job on time, these are able to request additional non-accelerated slots, taking into account how much slower these nodes are compared to running them on machines with accelerators.

Finally, notice that while the presented mechanism assumes



(a) 10% of the cluster accelerated

(b) 30% of the cluster accelerated

Fig. 2. Slot allocation as a function of load – non-accelerated pool runs tasks slower than the accelerated one, so applications with a higher speedup need a larger fraction of slots from the non-accelerated pool as soon as there is not enough capacity in the accelerated pool

that only two pools are defined in the system, regular and accelerated, it could be easily extended to support different types of accelerators. Then, affinity could be enforced across pools based on the speedup observed for each one of them, being the generic pool the last on to use for accelerable jobs.

B. Running Accelerated Tasks in the Non-accelerated Pool

Under some circumstances, such as jobs with very tight deadlines or overloaded systems, the scheduler is faced to the situation of having to deal with accelerable jobs that cannot meet their goal with the resources of the accelerated pool only. Therefore, the scheduler is forced to spread tasks from a job across both pools, accelerated and non-accelerated, in order to make progress faster.

While we will show real examples of such situation in Section V, in this Section we develop a theoretical model to illustrate how different jobs exhibiting different acceleration pose different challenges to the scheduler. For such purpose, we take a hypothetical cluster of machines running one single accelerable job, and suppose that a fraction of the available nodes are enabled with hardware accelerators (the accelerated pool) while the remaining are not enabled with this technology (the non-accelerated pool). We use two configurations of the cluster for our study, varying the size of each pool. In one case we use a proportion of 10% to 90% for the accelerated - non-accelerated pools, and in the second the proportion is 30% to 70%.

As defined above, let $s^m_{req,acc}$ be the number of slots to be allocated in parallel to an accelerable job m in the accelerated pool to make its goal, TT the set of TaskTrackers in the cluster, and s_t the number of execution slots offered by a TaskTracker t. Then, we define the load of the system at any moment of time as:

$$load = \frac{s^m_{req,acc}}{\sum_{t \in P_{acc}} s_t} \quad (4)$$

Therefore, a load of 50% means that job m requires an allocation equivalent to the 50% of the slots available in the accelerated pool P_{acc} to meet its goal. Notice that this definition of load is based on $s^m_{req,acc}$ and thus, on μ^m_{acc}. Therefore, $load$ is calculated based on the average value μ^m_{acc} observed for the group of TaskTrackers of the accelerated pool P_{acc}.

Results, in Figure 2, show load level for the system (defined in Figure case as described above) in the X-axis, and the fraction of nodes of the whole cluster that the job would be allocated in the Y-axis. Recall that the horizontal line indicates the fraction of the cluster that is accelerator-enabled for each scenario. Whenever a fraction of the cluster is allocated to the job, if the fraction fits in the accelerated pool, then accelerated nodes are allocated, providing on this way pool-affinity through the scheduler.

Figure 2 represents the effect of running accelerated tasks on the non-accelerated pool. Such situation is required when the load of the accelerated pool is beyond its capacity. This point can be seen in the figures when the allocated fraction of the cluster is above the size of the accelerated pool, indicated with the horizontal line and corresponding to 10% for Figure 2(a) and 30% for Figure 2(b). Beyond that point, accelerable tasks start running also on the non-accelerated pool, using non-accelerated versions of the code. Therefore, their performance is lower than when they run in the accelerated pool, and the difference depends on the per-task speedup of each job. In the Figure we have included the simulation for different jobs, each one with a different of per-task speedup. In particular we show values for per-task speedups of 1x, 2x, 3x, 4x and 5x. Recall that the higher the speedup is, the higher the allocation in the non-accelerated pool must be to compensate for the lack of additional accelerated nodes. As an example, for a per-task speedup factor of 5x, going a little beyond the capacity of the accelerated pool results in needing to allocate the entire cluster to the job to meet its completion time goal. This example illustrates how jobs that show high per-tasks speedups in the accelerated pool will force the scheduler to steal many

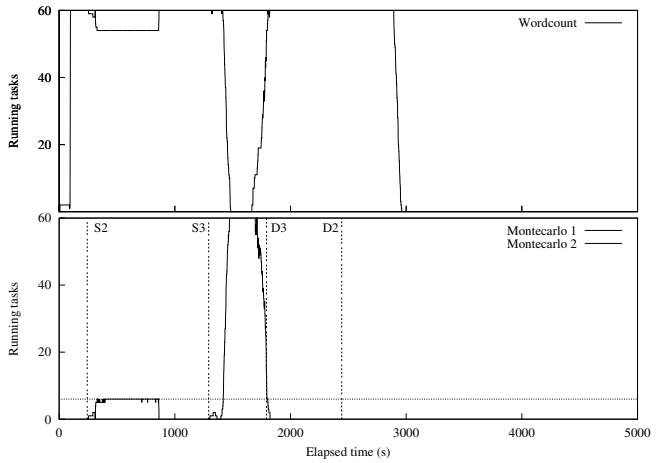


Fig. 3. Exp. 1A: Augmented Adaptive Scheduler

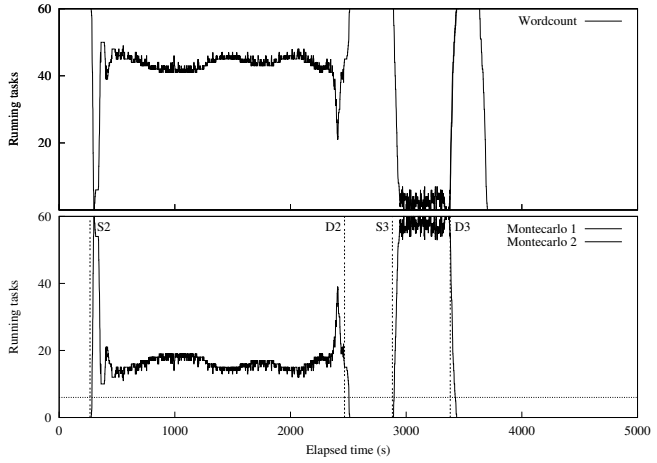


Fig. 4. Exp. 1B: Adaptive Scheduler without hardware affinity support

resources from the non-accelerated pool to satisfy the demand of the job, missing the goal in many situations. Section V will show this effect on real workloads running in a prototype on top of a medium-size cluster.

V. Evaluation

In this Section we present a series of tests that demonstrate the effectiveness of our techniques to enable the Adaptive Scheduler for Hadoop with hardware-awareness. We use a cluster of 60 IBM's Cell/BE blades to run MapReduce workloads composed of a set of jobs, each one deployed with a user-defined completion time goal. Although our cluster is homogeneous, in each test we decide to consider that only a fraction of the machines is accelerator-enabled, and non-accelerated nodes are used to run non-accelerated code only. In practice, it reproduces the configuration of a heterogeneous cluster in which some nodes are accelerator-enabled. Although we do not take advange of low level programming languages (i.e. OpenCL [9]) that support different parallel platforms (accelerators and multi-core generic processors, for instance),

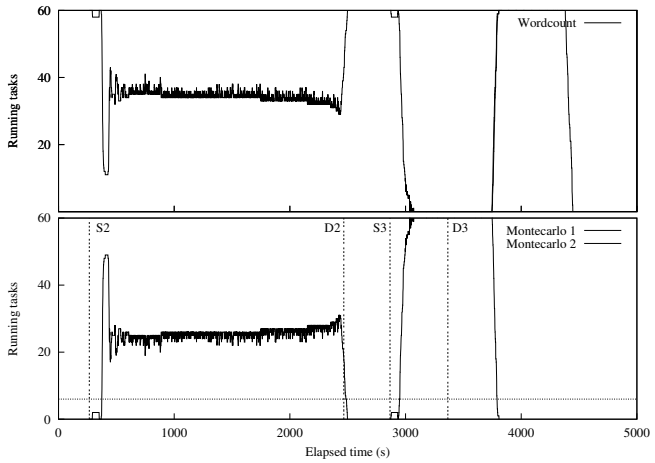


Fig. 5. Exp. 1C: Adaptive Scheduler (empty accelerated pool)

we emulate such behavior by deploying some jobs with multiple versions of their low-level code, that is, different versions of MapReduce's *map* function are implemented. Then, the runtime can dynamically choose which version to run in each node, depending on the characteristics of the node (accelerated or not accelerated). Notice that for any accelerable job, multiple MapReduce tasks can be scheduled at the same time on different nodes and thus, multiple versions of its *map* function (accelerated and non-accelerated) can be running simultaneously in different nodes, being their results collected and merged by Hadoop runtime after their execution.

In each test we evaluate different properties of the decisions made by the Adaptive Scheduler. Different tests are grouped into two experiments. In the first experiment we look at the capacity of the scheduler to dynamically estimate the speedup shown by accelerable applications, use that information to estimate the resource requirements for each job, and provide accelerated pool-affinity when possible. In the second experiments, we explore how the Adaptive Scheduler handles competition within the accelerated pool (competition for non-accelerated resources was already explored in [3]), as well as how accelerable tasks are run for some jobs in the non-accelerated pool to make the necessary progress to meet their completion time goal.

A. Testing environment

The system used to run the experiments is a 60 IBM QS22 blades cluster, each one equipped with 2x 3.2Ghz Cell processors and 8GB of RAM. These nodes were used as Hadoop DataNodes and TaskTrackers. The JobTracker and the NameNode run in a separate machine. All the nodes were connected using a Gigabit ethernet. To simulate an environment in which some of the nodes are enabled with accelerators, we create two logical partitions in the system: a 54 node partition is considered to have no acceleration support (running code will be limited to the PPU of the Cell processors), and 6 nodes are accelerated (having full access to their SPUs).

For this experiment we use three MapReduce applications that will be colocated on the same physical cluster:

- Montecarlo: we choose a Montecarlo simulation application to represent CPU intensive applications with very little input data. We have two implementations of the map section of this MapReduce application, one is written in pure Java while the other is Cell/BE-accelerated.
- Crypt: we use a data encryption application to represent data-intensive accelerable applications. We have implemented a 128 bits key AES encryption algorithm using both a Java code and a Cell-accelerated code. The Java version is based on the methods offered in the standard `javax.crypto.Cipher` package. This application encrypts an input of 60GB in all experiments.
- WordCount: the third application is the WordCount sample application, only in non-accelerated version, processing an input between 57 and 125GB worth of text files.

Both Montecarlo and Crypt can benefit from executing on nodes with acceleration support (with a speedup factor of up to 25x and 2.5x respectively). The TaskTrackers are enabled to decide at runtime which version of the mapper should be used depending on the locally available hardware. WordCount does not have an accelerated map version, so the performance of its tasks is the same on all nodes.

B. Experiment 1: Scheduling accelerable tasks

In this section we evaluate the execution of the same workload using three different configurations, and will show the benefits of adding hardware affinity support to the Adaptive Scheduler when running simultaneously accelerable and non-accelerable applications. The configurations are as follows:

- Experiment 1A. Augmented Adaptive Scheduler with hardware affinity and with 10% of the nodes enabled with hardware acceleration support. In this case, the Adaptive Scheduler prioritizes the allocation of accelerator-enabled nodes to accelerable applications, as described in Section IV-A.
- Experiment 1B. Original Adaptive Scheduler, that is, without hardware awareness. No hardware affinity is provided. In this configuration, although the scheduler does not distinguish the kind of hardware, 10% of the nodes are still enabled with hardware acceleration support. Thus, all map tasks assigned to accelerated nodes execute the accelerated version of the code (if available). Variance of μ_m is high.
- Experiment 1C. Augmented Adaptive Scheduler with hardware affinity considerations, but without nodes enabled with hardware acceleration support. In this case, only non-accelerated versions of the map tasks are executed across the cluster.

Figure 3, Figure 4 and Figure 5 show the results for the execution on the three configurations. In all configurations, the workload is composed of one instance of the WordCount application, which is not able to exploit hardware acceleration support, and two instances of the Montecarlo simulation, which exhibit a high speedup on accelerated nodes. The first job that is submitted is the WordCount application. Afterwards, the first Montecarlo simulation is submitted at S2, and a second Montecarlo job is submitted 300 seconds *after* the first one completes, at time S3. Recall that S3 will vary depending on the actual completion time of the first instance. WordCount is set to have a relaxed completion time goal, of 5000 seconds, while Montecarlo simulations have tighter completion time goals, being 2200 seconds (D2) for the first instance and 500 seconds (D3) for the second one. The goal for the first instance of Montecarlo can be met running only with resources from the accelerated pool, while the goal for the second instance is so tight that this job needs to spread over both pools to meet it.

Figure 3 shows the results for Experiment 1A. An horizontal line indicates the limit of the accelerated pool. Recall that the WordCount application initially runs across the two pools, but without exploiting the acceleration capabilities of the accelerated pool, because no accelerated code is provided for this application. When the first Montecarlo job is submitted(S2), the Adaptive Scheduler starts the calibration phase, putting two tasks to run for this job, one in each pool, to evaluate the speedup. When the first estimation of resource demand is done, the scheduler starts allocating resources to the job in the accelerated pool. The job completes at time 863s. Short after that, the second Montecarlo instance is submitted (S3). Due to its tighter completion time goal, and after the initial execution of one task in each partition to calibrate the estimation, the scheduler determines that for this second instance to meet its goal, the job needs to be spread across both partitions, in a combination of pure Java and accelerated versions of the code. The job meets its completion time goal and completes at time 1824s. Notice that this second Montecarlo instance performs just slightly faster than the first Montecarlo instance, although it is running most of the time using 9x times more nodes than the first one. Further details of this situation are discussed in Section IV-B. After that point, the WordCount job gets all the resources allocated again and runs until completion.

Figure 4 shows the results for Experiment 1B, in which the Adaptive Scheduler is not aware of hardware heterogeneity. In this case, all maps of all jobs execute across all nodes. As it can be observed, the number of nodes assigned to the first Montecarlo job is higher than when using the Adaptive Scheduler with hardware affinity. The reason for this is that, as the Adaptive Scheduler assigns nodes to this job that do not have hardware acceleration support, the execution of the non-accelerated version of these maps increases considerably the average map time for this job (recall that this non-accelerated version is around 25x slower than the accelerated version). Thus the job requires more nodes to meet its completion time goal. We have to remark that, although the higher number of assigned nodes, the execution time of this job is still more than twice the execution of the same job under the same execution conditions but using hardware affinity during the scheduling decisions. Notice also that this configuration also increases the execution time of the WordCount application. This is because,

as this application has a very relaxed completion time goal, it has a low priority for the scheduler that assigns to WordCount only the nodes that the Montecarlo simulation does not need to meet its tight completion time goal. Regarding the execution of the second job of the Montecarlo simulation there are not noticeable differences between this configuration and the configuration considering hardware affinity because its tight completion time goal requires in both configurations to get most of the nodes in the cluster. Finally, Figure 5 shows the results for Experiment 1C, in which there are no nodes with hardware acceleration support. Thus, in this configuration only non-accelerated versions of the map tasks are executed. This is the configuration that requires a higher number of nodes to meet the completion time goal of the first Montecarlo job and, thus, the configuration that gets the worst execution time for the WordCount application. This configuration also illustrates the performance penalty that Montecarlo simulation suffers when there are not accelerated nodes available and only it is possible to execute the non-accelerated version of its map function. Although the second job of Montecarlo get all the nodes in the cluster, its execution time doubles the execution time observed when all nodes are assigned too but 10% of them are enabled with hardware acceleration support (see Figure 3). Thus, this second job misses its completion time goal, as it was set tight enough to require the allocation of all the cluster when the size of the accelerated pool is 10% of the cluster used for these experiments.

C. Experiment 2: Arbitrating between pools

In this section we evaluate the Adaptive Scheduler when arbitration between both pools is required. We use a workload composed of a WordCount job (which is not accelerable), the Montecarlo simulation (which is accelerable and has a high per-task speedup) and the Crypt application (which is accelerable and has a moderated per-task speedup).

We show the results for the following three configurations:

- Experiment 2A. This configuration shows an scenario in which completion time goal of each job is relaxed enough to be met using only nodes of the pool with highest affinity for each job. All of the jobs run simultaneously.
- Experiment 2B. In the second configuration we show how the Adaptive Scheduler arbitrates the allocation inside the accelerated pool when two accelerable jobs compete for its nodes. In addition, the tight completion time goal for accelerable jobs forces the scheduler to allocate them nodes from the non-accelerated pool.
- Experiment 2C. This configuration illustrates how a non-accelerable job can steal nodes from the accelerated pool when needed to meet its completion time goal.

Figure 6 shows the results for Experiment 2A. We execute one instance of each application of the workload, all of them with a relaxed completion time goal, which can be met using only the nodes from the affinity pool of each application (accelerated pool for the Montecarlo simulation and Crypt and non-accelerated pool for WordCount). An horizontal line in the graphs marks the number of nodes in the accelerated pool. We first launch the job that executes the WordCount application. The scheduler executes one map task on each pool to determine whether the application is accelerable or not. After determining that it is not, the job is assigned all available resources in the cluster until the job that executes the Crypt application is submitted (instant S2). At this moment, the scheduler assigns to the Crypt job one node of each pool to decide the affinity of the job. Once the scheduler decides that this job is accelerable, it starts applying the affinity criteria: as WordCount map tasks running on accelerated nodes are finishing the scheduler assigns them to Crypt. When the job for the Montecarlo simulation starts (instant S3), both jobs have to share the accelerated pool. The scheduler decides how to allocate these nodes considering the completion time goal of each job. In this example, Montecarlo has a tighter completion time than Crypt and, for this reason, it gets more nodes than Crypt. At the same time, WordCount continues the execution using only nodes from the non-accelerated pool, as they are enough for this job to meet its completion time goal. Montecarlo ends the execution meeting its performance goal (instant D3) and at this moment all accelerated nodes are assigned to the Crypt job that ends the execution meeting its performance goal at instant D2.

Figure 7 shows the result for Experiment 2B, in which the load over the accelerated pool is high. The experiment is basically the same as that shown in Figure 6, except an additional Montecarlo simulation is submitted in order to increase the load on the accelerated pool. Initially, a WordCount instance is submitted. At time S2 an instance of the Crypt application is submitted and at instant S3 the first instance of the Montecarlo simulation reaches the system. After the first instance of the Montecarlo simulation is completed, a second instance of Montecarlo is submitted, in this case with a very tight completion time goal. This forces the Adaptive Scheduler to assign to this job most of the nodes in the cluster, from both accelerated pool and non-accelerated pool. When this job completes, Crypt and WordCount continue to run using only the nodes from the accelerated and non-accelerated pool respectively, until the scheduler detects that the Crypt has not enough resources to meet its completion time goal. At that moment, Crypt starts getting allocated some non-accelerated nodes. When Crypt completes, WordCound starts running across all nodes in the cluster and finally meets its goal.

Notice from this experiment that when the scheduler detects that accelerated nodes are not enough to meet the completion time goal of accelerable job (Crypt and Montecarlo), the number of non-accelerated nodes required to compensate the shortage of accelerated nodes is considerably higher for Montecarlo than for Crypt. As discussed in Section IV-B, this is due to the different per-task speedup of both jobs: 25x in the case of Montecarlo tasks and 2.5x in the case of Crypt tasks.

Finally, Figure 8 shows the results for Experiment 2C, in which the accelerable jobs have relaxed completion time goals and the non-accelerable job is submitted with a very tight

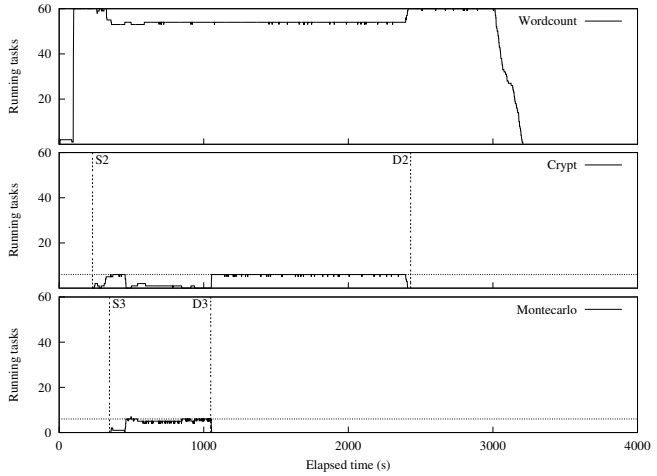


Fig. 6. Exp. 2A: Low load scenario

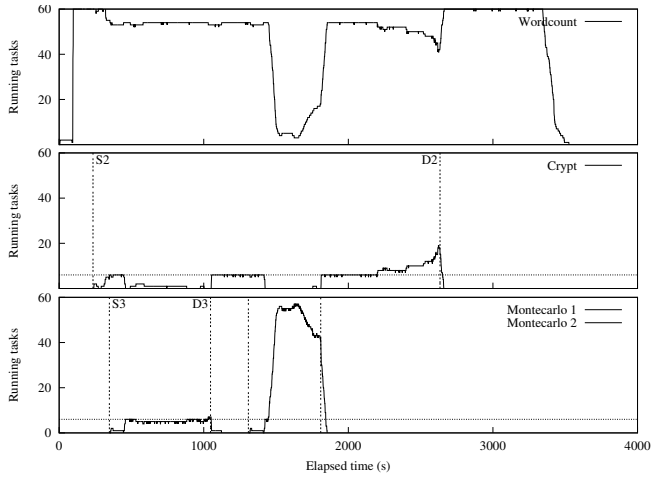


Fig. 7. Exp 2B: Heavy load on the accelerated pool

completion time goal. Initially, the scheduler estimates that WordCount and Crypt will be able to meet their completion time goals. But short after Montecarlo is submitted, the scheduler notices that WordCount will need more resources to meet its goal and thus claims some nodes from the accelerated pool. After WordCount is completed, the remaining jobs continue sharing the accelerated pool, but most of it is assigned to Crypt since it has a higher need of slots. Once the scheduler acknowledges that Crypt too will meet its completion goal and Montecarlo becomes more needy (at around instant 2850), both jobs start sharing the pool more equally until completion.

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 ([15]). There has also been some work focused on adaptive scalable schedulers based on job sizes ([16], [17]), but in addition to some of these ideas, the proposed scheduler takes advantage of one of the key features of MapReduce: the fact that jobs are composed of a large number of similar tasks.

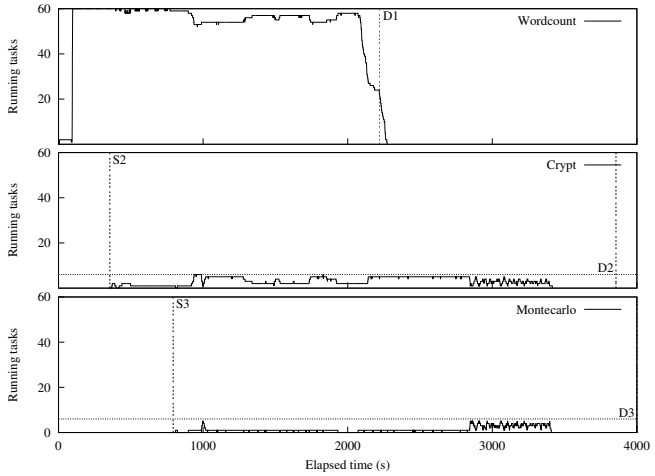


Fig. 8. Exp. 2C: Heavy load on the non-accelerated pool

There is little specific work on scheduling for MapReduce applications. The initial scheduler provided by 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 project is working on a system for provisioning dedicated Hadoop clusters to applications [18], but this approach can result in resource underutilization. In [19] 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 and achieve maximum utilization of the resources. However, scheduling decisions are not dynamically adapted based on job progress, so this approach isn't appropriate for applications with different performance goals.

In [20], the authors introduce a system to manage and dynamically assign the resources of a shared cluster to multiple Hadoop instances. Priorities are defined by users using high-level policies such as budgets. This system is designed for virtualized environments, unlike the proposed work, which is implemented as a regular Hadoop MapReduce scheduler and thus is able to run on standard Hadoop installations and provide more accurate estimations.

There are several works in the literature that consider the heterogeneity trend on current execution platforms. For example, there are some works conducting research on combining two programming models (MPI and OpenMP) under a hybrid approach [21], aiming to favour scalability without renouncing to performance. These hybrid solutions have been extensively used for scientific computing, but have not become generally used in other fields basically for two key issues.

Recent work introduced in [22] presents an API to develop programs to run on hybrid architectures, regardless the particular underlying core or hardware accelerator.

VII. Conclusions

In this paper we have shown how MapReduce runtimes can be leveraged to run heterogeneous sets of workloads,

including accelerated and non-accelerated applications, on top of heterogeneous clusters, composed of regular nodes and accelerator-enabled systems. Hybrid systems and heterogeneous processors can offer advantages to some MapReduce workloads, providing specialized cores that can efficiently perform some critical tasks. Exploiting such hardware infrastructure requires some kind of infrastructure-awareness on the task scheduler, providing hardware affinity when necessary. Real time monitoring of the tasks allows the scheduler to evaluate the real benefits of running each workload on different platforms, and the scheduler may decide the best distribution of tasks on nodes accordingly. Depending on individual performance goals of each job and on the availability of generic and hardware-specific code for each application, the scheduler may decide to run any version of the code on top of the available hardware. Low level programming languages (i.e. OpenCL [9]) that support different parallel platforms can provide great advantage in heterogeneous scenarios.

We have developed a prototype, based on the Adaptive Scheduler for Hadoop, that is driven by user-defined high-level performance goals and that is able to make the most of the underlying hardware. The scheduler logically splits the MapReduce cluster into two partitions, two pools of servers with different capacity to run accelerated tasks or not. Then, it takes into account different jobs' properties, including their completion time goals and the availability of accelerable versions of their code, to schedule their tasks across the cluster. Jobs that are deployed with accelerated and non-accelerated versions of their tasks will exhibit different performance depending on the pool in which their tasks are scheduled. Therefore, the scheduler provides accelerated pool-affinity to accelerable jobs when possible, but is not limited to run jobs' tasks in one single pool. We have demonstrated the effectiveness of the Adaptive Scheduler through a series of experiments run on top of a medium-sized cluster of Cell/BE blades.

The Adaptive Scheduler provides dynamic resource allocation across jobs, hardware affinity when possible, and is even able to spreads jobs' tasks across accelerated and non-accelerated nodes in order to meet performance goals in extreme conditions. To our knowledge this is the first MapReduce scheduler and prototype that is able to manage high-level performance goals even in presence of hybrid systems and accelerable jobs.

ACKNOWLEDGEMENTS

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, and by IBM through the 2010 IBM Faculty Award program.

REFERENCES

[1] Nasa Nebula project.
URL http://nebula.nasa.gov

[2] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: OSDI '04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, 2004, pp. 137–150.

[3] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, I. Whaley, Performance-driven task co-scheduling for mapreduce environments, in: NOMS '10: Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium, IEEE, Osaka, Japan, 2010.

[4] D. Carrera, M. Steinder, I. Whalley, J. Torres, 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] Los Alamos National Laboratory, High-Performance Computing: Roadrunner.
URL http://www.lanl.gov/roadrunner/

[6] B.-G. Chun, G. Iannaccone, G. Iannaccone, R. Katz, G. Lee, L. Niccolini, An energy case for hybrid datacenters, in: Workshop on Power Aware Computing and Systems (HotPower'09), Big Sky, MT, USA, 2009, 2009.

[7] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, A novel simd architecture for the cell heterogeneous chip-multiprocessor, 2005.

[8] Y. Becerra, V. Beltran, D. Carrera, M. Gonzalez, J. Torres, E. Ayguadé, Speeding up distributed mapreduce applications using hardware accelerators, in: ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 42–49.

[9] K. Group, OpenCL (Open Computing Language) - The open standard for parallel programming of heterogeneous systems.
URL http://www.khronos.org/opencl/

[10] Apache Software Foundation, Hadoop MapReduce Tutorial.
URL http://hadoop.apache.org/mapreduce/

[11] Apache Software Foundation, Hadoop Distributed File System (HDFS) Architecture.
URL http://hadoop.apache.org/core/docs/current/hdfs_design.html

[12] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. Gschwind, Optimizing compiler for the cell processor, in: PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005, pp. 161–172.

[13] Sun Microsystems, Inc, Java Native Interface.
URL http://java.sun.com/docs/books/jni

[14] Adaptive Scheduler. Apache isue MAPREDUCE-1380.
URL https://issues.apache.org/jira/browse/MAPREDUCE-1380

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

[16] P. R. Jelenkovic, X. Kang, J. Tan, Adaptive and scalable comparison scheduling, in: SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM, New York, NY, USA, 2007, pp. 215–226.

[17] A. Wierman, M. Nuyens, Scheduling despite inexact job-size information, in: SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM, New York, NY, USA, 2008, pp. 25–36.

[18] Apache Software Foundation, Hadoop on demand.
URL http://hadoop.apache.org/core/docs/r0.20.0/hod_user_guide.html

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Job scheduling for multi-user MapReduce clusters, Tech. Rep. UCB/EECS-2009-55, EECS Department, University of California, Berkeley (Apr 2009).

[20] T. Sandholm, K. Lai, MapReduce optimization using regulated dynamic prioritization, in: SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, ACM, New York, NY, USA, 2009, pp. 299–310.

[21] N. Drosinos, N. Koziris, Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04), 2004.

[22] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, in: Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science, Vol. 5704 of Lecture Notes in Computer Science, Springer, Delft, The Netherlands, 2009, pp. 863–874.