

Shared Cluster Scheduling: a Fair and Efficient Protocol

Pietro Michiardi Antonio Barbuzzi
EURECOM
Sophia Antipolis, France
first_name.last_name@eurecom.fr

Damiano Carra
University of Verona
Verona, Italy
damiano.carra@univr.it

ABSTRACT

In this work we focus on the problem of resource allocation in a shared cluster used for data-intensive scalable computing. Specifically, we target the open-source implementation of the MapReduce framework, Hadoop, and design a new scheduling algorithm that caters both to a fair and efficient utilization of a shared cluster. Our scheduler, labelled FSP, achieves both goals by “focusing” the resources of a cluster toward individual jobs rather than partitioning the cluster such that all submitted jobs run in parallel.

In this article, we discuss in detail the implementation challenges posed by our scheduler, and suggest a new technique to achieve preemption through Job suspension. Such an approach has several advantages over the traditional method of killing running jobs, but requires to extend the principle of data locality to all phases of a MapReduce Job.

Finally, we note that our scheduler is not oblivious to typical Job optimization techniques, a unique feature missing in state-of-the-art approaches.

1. INTRODUCTION

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [3] and Dryad [7], has created the need to organize and manage the resources of clusters of computers that operate in a shared environment. Initially designed for few and very specific batch processing jobs, data-intensive scalable computing frameworks are nowadays used by many companies (e.g. Facebook, LinkedIn, Google, Yahoo!, ...) for production, recurrent and even experimental data analysis jobs. Within the same company, many users *share* the same cluster because this avoids redundancy (both in physical deployments and in data storage) and may represent enormous cost savings.

In this work, we study the problem of resource *scheduling*, that is how to allocate the (computational) resources of a cluster to a number of concurrent jobs submitted by the users, and focus on the open-source implementation of MapReduce, namely Hadoop [1]. Despite scheduling is a

well known research domain, the distributed nature of data-intensive scalable computing frameworks makes it particularly challenging. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several solutions to the problem have been proposed to the community [11, 2, 5, 8, 9, 10]: in general, existing approaches aim at two key objectives, namely *fairness* and *performance*.

For example, [11] propose a scheduler that in principle is equivalent to processor sharing; in addition, the authors note that data locality, which imposes computation to be moved toward the data rather than vice-versa, plays an important role in the efficiency of a particular job schedule. By simply waiting for a suitable resource to be available, the “delay scheduler” [11] greatly reduces the amount of data that has to be moved within the shared cluster. In another work [2], scheduling is cast as an optimization problem: using an abstract system model, the authors focus on finding an approximate solution to an on-line scheduling problem whose objective function to minimize is the total time required to complete all jobs served by the system, while fairness is not considered. The work presented in [5] considers a non-cooperative scenario in which users may be inclined to “game” the scheduler in order to receive more than their fair share of the cluster, and proposes a new, strategy-proof metric to be enforced by the cluster scheduler.

Given the state-of-the-art, it is natural to question the need for another approach to scheduling cluster resources. In this work we observe that fairness and performance are non-conflicting goals, hence there is no reason to focus solely on one or the other objectives. We proceed with the design of a scheduling protocol that can be implemented in practice, and that caters both to a fair and efficient utilization of a shared cluster.

First, in Sec. 3, we give an high level overview of our algorithm, and compare it to traditional fair-sharing approaches. When the demand for cluster resources is high, *i.e.*, jobs necessitate all cluster machines, our scheduler “focuses” the resources to an individual job, instead of partitioning the cluster to accommodate all jobs. When the workload of a cluster is composed by heterogeneous jobs, in terms of resource requirements, our scheme accommodates the possibility of running small jobs in parallel, so that the cluster is fully utilized.

In Sec. 4, we delve into the implementation details of our scheduler. In contrast to previous works, we pinpoint the key aspects of the MapReduce framework that need to be considered when implementing a scheduling policy: as a result, we show that locality constraints are important both

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LADIS 2011 Seattle, US

Copyright 2011 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

for input data and for intermediate results generated during the job execution flow. Furthermore, we show that our approach to scheduling supports job optimization techniques that aim at minimizing I/O operations. This is very important as MapReduce job designers usually spend a non-negligible amount of time in optimizing their jobs: this effort can be nullified by a scheduler that is oblivious to such optimization techniques.

Sec. 5 concludes this article with a series of further improvements that can be applied to our scheme, and with our research agenda toward the implementation and evaluation of our scheme.

2. MAPREDUCE

MapReduce, popularized by Google with their work in [3] and by Hadoop [1], is both a programming model and an execution framework. In MapReduce, a job consists in three phases and accepts as input a dataset, appropriately partitioned and stored in distributed file system. In the first phase, called MAP, a user-defined function is applied in parallel to input partitions to produce intermediate data stored on the local file system of each machine of the cluster; intermediate data is sorted and partitioned when written to disk. Next, during the SHUFFLE phase, intermediate data is “routed” to the machines responsible for executing the last phase, called REDUCE. In this phase, intermediate data from multiple mappers is sorted and aggregated to produce output data which is written back on the distributed file system.

In this Section we gloss over several details of MapReduce (due to lack of space) and focus on the key ingredients that define the performance of a job, in terms of execution time. Simply stated, disk and network I/O are the main culprits of poor job performance. The task of a job designer is then to optimize the amount of memory allocated to mappers and reducers, so as to minimize disk access. Moreover, a job may include an optional *Combiner* phase in which intermediate data is pre-aggregated before it is sent to reducers, to minimize the amount of data to be transmitted over the network. Job optimization is generally a manual process that requires the knowledge of the size of intermediate data sent to each reducer, and the characteristics of the cluster, including number of nodes, number of processors and cores and available memory.

A key component of the MapReduce framework is the scheduler, which is the subject of this work. The role of the scheduler in MapReduce is to allocate resources to running tasks: MAP and REDUCE tasks are granted independent slots on each machine. The number of MAP and REDUCE slots is a configurable parameter.

When a single job is submitted to the cluster, the default scheduler simply assigns as many MAP tasks as the number of machines in the cluster. Note that the total number of MAP tasks is equal to the number of partitions of input data: as such, the scheduler may need to wait for a portion of MAP tasks to finish before scheduling subsequent mappers. Similarly, REDUCE tasks are scheduled once all intermediate data output from mappers is available: reducers may receive data from potentially all mappers. Note that Hadoop implement an optimization called “pipelining”: if all reducers were to be scheduled once all mappers complete, the network would suffer from a burst in data transmission and congestion may arise. Instead, with pipelining, reducers are scheduled be-

fore mappers are done¹ such that they can start copying intermediate data as soon as some is available. However, the user-defined REDUCE function is only executed once all mappers are done.

When multiple jobs are submitted to the cluster, the scheduler decides how to allocate available task slots across jobs. The default scheduler in Hadoop implements a FIFO policy: the whole cluster is dedicated to individual jobs in sequence; optionally, it is possible to define priorities associated to jobs. Despite the well known problems of FIFO scheduling, which starves short jobs that sit in a queue waiting for a long job to finish, jobs can be optimized as if they were executing alone in the cluster, which is a desirable feature.

3. PROPOSED SCHEME

Scheduling in a distributed system like Hadoop represents a challenging problem: theoretical results for such a complex environment are hard to derive. For this reason, we adopt a top-down approach. We first consider the distributed system as a single processing resource: according to the measurements in [11], jobs arrive according to a Poisson process, while job size distribution is unknown. Therefore we can look at the general results for the M/G/1 queueing system and select the most promising scheduling policy. Such scheduling policy would be subsequently adapted to the specific context of Hadoop.

3.1 Scheduling Disciplines

Scheduling has been a subject of many studies. Here we focus on the theoretical results for scheduling policies in case of M/G/1 systems. We consider the mean response time (i.e. the total time spent in the system, given by the waiting and service time), and the fairness. While fairness is a complex subject, we consider the notion of fairness as equal share of the resources².

Among all the scheduling disciplines proposed in the literature (for a general overview, see [6] and the references therein), we consider only two policies that are relevant to our case: a policy that minimizes the mean response time and one that provides perfect fairness.

In the system we consider, the job size is known *a priori*: in this case the optimal preemptive scheduling policy that minimize the mean response time is the Shortest Remaining Processing Time (SRPT), where the job in service is the one with the smallest remaining processing time. Since the focus is on the mean response time, the fairness is not guaranteed, i.e., long jobs may starve. SRPT represents an interesting solution, since recent measurements [11] have shown that the job size distribution in MapReduce clusters belongs to the category for which SRPT may provide fairness; nevertheless, it is also true that there are many short periodic jobs – a case where SRPT may perform poorly [4].

If we consider only the fairness, Processor Sharing (PS) represents the policy that guarantees a fair share of the resources. In PS, if there are N jobs in the system, each job

¹Precisely, a configuration parameter indicates the fraction of mappers that are required to finish before reducers are awarded a slot.

²Alternatively, the fairness can be defined through the total time spent in the system normalized to the job size, which should be proportional to $1/1 - \rho$, where ρ is the server utilization.

receives a $1/N$ th fraction of the server resources. Unfortunately, the mean response time is higher (especially for high loads) than SRPT.

Given the two objectives, good performance (in terms of mean response time) and fairness, is it possible to obtain both with a scheduling policy? The solution of this problem is represented by the Fair Sojourn Protocol (FSP) [4], described in the following section.

3.2 How FSP Works

The main idea of FSP is to run jobs in series rather than in parallel. In practice, assuming a PS policy, where each job has its fair share, it is possible to compute the completion time. The order at which jobs complete in PS is used as a reference to schedule jobs in series. In the basic single server configuration, this means that at most one job is served at a time, and the job can be preempted by a newly arrived job.

In order to show how FSP works, we make an example. Assume that there are three jobs, j_1 , j_2 and j_3 , which require 100% of the cluster. The jobs arrive at time $t_1 = 0s$, $t_2 = 10s$ and $t_3 = 15s$ respectively, and it takes 30 seconds to process job j_1 , 10 seconds to process job j_2 and 10 seconds to process job j_3 . For simplicity, assume that the processing time is independent from the location of the tasks, i.e., even if some tasks need to be moved within the cluster, the processing time remains the same.

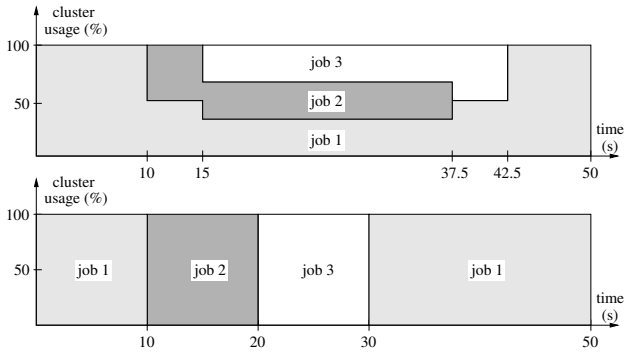


Figure 1: Comparison between PS (top) and FSP (bottom).

Figure 1 (top) represents the cluster usage over time in case of PS: when job j_2 arrives, the cluster is shared between j_1 and j_2 , and, when job j_3 arrives, the cluster is shared among the three jobs. The job completion order is j_2 , j_3 and j_1 . The bottom part of the figure shows the FSP approach. When job j_2 arrives, since it would finish before job j_1 in case of PS, it preempts job j_1 . When job j_3 arrives, it does not preempt job j_2 , since it would finish after it in case of PS; when job j_2 finishes, job j_3 is scheduled since it would finish before job j_1 in case of PS.

The FSP scheme is able to assure that each job receives the fair amount of resources as in the PS scheduling. At the same time, the scheme is able to decrease the mean job completion time. In particular, long jobs tends to have the same completion time as in PS, while short jobs finish before.

While the formulation of FSP is simple in case of single server, when we take into accounts a cluster of servers, we should adapt the scheme to this specific context. In particular, it may happen that a job is composed by few tasks, and it is sufficient to use a portion of the cluster to process such

job at the maximum possible speed. We illustrate this situation in the following example. Assume that jobs j_1 , j_2 and j_3 require 100%, 55% and 35% of the cluster respectively. The arrival times are $t_1 = 0s$, $t_2 = 10s$ and $t_3 = 13s$ and the processing time (if the required percentage is given) is 30 seconds for job j_1 , 10 seconds for job j_2 and 10 seconds for job j_3 .

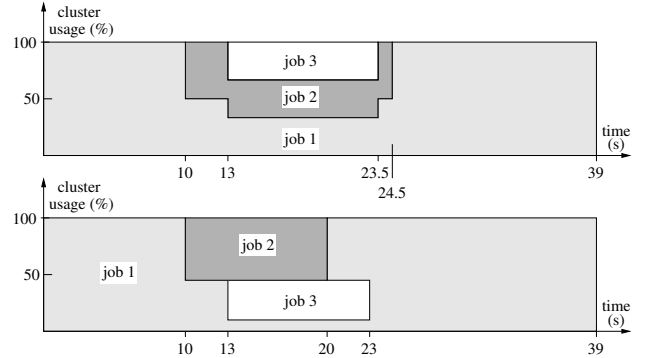


Figure 2: Comparison between PS (top) and FSP (bottom), with jobs that do not require the full cluster.

Figure 2 compares the processing in case of PS (top) and FSP (bottom) approach. With FSP, job j_2 would preempt job j_1 ; since j_2 requires only 55% of the cluster, the remaining 45% can still be used by j_1 . When job j_3 arrives, it would preempt job j_1 (but not job j_2), but it is sufficient to allocate 35% of the cluster to serve it, leaving 10% of the cluster to job j_1 . Even in this case, FSP is able to decrease the mean job completion time, yet maintaining the fair allocation of the resources. Note that the final order of job completion with FSP is different from the PS case (j_2 , j_3 and j_1 instead of j_3 , j_2 and j_1): in this case job j_2 finishes before the corresponding completion time in case of PS, therefore the fair allocation of the resources is not compromised.

Having described with some examples the general behavior of the scheme, we provide the basic algorithm for task allocation implemented in the scheduler. As observed in [11], when the scheduler needs a resource, it can either kill some existing running tasks or wait for task completion. Since killing task would be a waste of resources, it is preferable to wait for available task slots: thanks to the high rate at which task finish, this does not represent a major problem. In our specific case, if a newly submitted job preempts the existing one (or ones), we simply let the current running task complete and launch the new job as task slots become available.

The general scheme is therefore composed by two parts which perform different actions for the two possible events: (i) job submission or completion, and (ii) task completion (see Algorithm 1).

When a job is submitted or finishes, the algorithm computes the fair share for each job and updates the finish times. These values are used to order the jobs, i.e. priority is given according to the completion time. When a task slot becomes available, the scheduler considers the ordered list of current jobs and assigns the task to the higher priority job that has at least one unlaunched task – a job may still be running, but all its tasks (e.g., all the REDUCE tasks) may have been already allocated.

Algorithm 1 Basic Task Allocation

```
1. when a job is submitted / finishes do
2.   for all jobs do
3.     compute the fair share
4.     compute the finish time
5.   end for
6.   sort jobs according to their finish time
7. end when
8.
9. when a task slot is available on machine M do
10.  for j in jobs do
11.    if j has an unlaunched task t then
12.      launch t on M
13.    return
14.    end if
15.  end for
16. end when
```

The proposed basic scheme does not take into account different aspects, e.g., task locality or user fair share (as opposed to job fair share) which are discussed in the next section.

3.3 Improving basic FSP scheme

The scheme summarized in Algorithm 1 is divided into two parts: job sorting and task assignment. These two building blocks can be modified independently, since they solve different problems.

The job ordering can be done taking into account multiple aspects: for instance, we may assume that jobs belong to different classes, and each class has different shares of the cluster. In this case the job completion times are computed considering the Generalized Processor Sharing (GPS), where every job i has a weight ϕ_i and it receives a share $\phi_i / \sum_j \phi_j$ (the sum over j is done considering the current jobs) of the resources. Another modification may take into account users rather than jobs, *i.e.*, jobs submitted by the same user are served with FIFO, and the resources are shared among users.

The task assignment mainly tries to solve problems related to locality. Given the ordering of the jobs, the task assignment may skip some jobs if the locality requirement is not met. The implementation of this approach can be borrowed directly from the Delay Scheduler [11].

As a final remark, we should mention that FSP needs to estimate the job finish time. While, as a first approximation, we may consider the job size, we will evaluate more detailed mechanisms (that take into account, for instance, the difference between CPU-intensive and memory-intensive jobs) as future work.

4. IMPLEMENTATION DETAILS

In Sec. 3 we assume a simplified version of the MapReduce framework to outline the key ideas behind FSP: for instance, we do not differentiate between MAP and REDUCE tasks. We now drop such assumptions and delve into the implementation details of FSP.

4.1 Preempting jobs

The solution we propose is based on job preemption. How such preemption should be managed? In order to understand this aspect, let's focus on a detailed example, shown in Fig. 3, where we consider two jobs, j_1 and j_2 : j_1 operates on a dataset split in 10 blocks, involving 10 MAP tasks and 3 REDUCE tasks; j_2 is smaller, and consists in 4 blocks, 4 MAP

tasks and 4 REDUCE tasks. In the example we consider a cluster of 4 machines, with 1 MAP and 1 REDUCE slot each. Fig. 1 illustrates, on a time-line, the task allocation per machine, per slot and per task, with salient points annotated on the time-line.

Note that the ingredients of this example are similar in spirit to that of Fig. 1 (although with two jobs instead of three). Both jobs require all cluster resources (at least in the MAP phases) since input blocks are located on all the cluster machines: as such, j_2 preempts j_1 .³

In Fig. 3, $map(i, j)$ indicates MAP task number j of job i ; similarly $reduce(i, j)$ indicates a REDUCE task number j of job i . First, j_1 arrives and occupies the whole cluster: the first 4 MAP tasks of j_1 are scheduled. After some time, j_2 arrives: based on Algorithm 1, j_2 preempts j_1 . Preemption of MAP tasks is not immediate: the FSP scheduler waits for mappers from j_1 to finish to schedule mappers from j_2 .

It is now important to note how FSP handles REDUCE task scheduling: as soon as⁴ MAP tasks from j_1 finish, REDUCE tasks from the same job are scheduled on the available slots: in the figure, the three REDUCE tasks from j_1 are scheduled on machines M_1, M_3, M_4 .

Now, when mappers from j_2 are done⁵, FSP schedules the reducers. To do so, two options are available: REDUCE tasks from j_1 can be killed or, as we suggest, *suspended*. Killing REDUCE tasks, as implemented in [11], is convenient because the scheduler is simplified: reducers are simply tagged as not completed and will be scheduled subsequently. However, this simplification comes at a price: all work done by the reducers, including disk and network I/O is wasted.

Since I/O is the major factor that determines the job execution time, we instead suggest to suspend reducers and save their state to disk. REDUCE task suspension can be implemented as follows: the scheduler can force the TaskTracker in charge of a reducer to *spill* data to disk; once this is done, the scheduler pushes the suspended reducers in the queue of pending REDUCE tasks. One key observation is necessary: as intermediate (and sorted) data is materialized on disk, suspended REDUCE tasks must be resumed on the same machines they were suspended on. In practice, *data locality* applies to both MAP and REDUCE tasks. We show this in Fig. 3: once mappers of j_2 are done, reducers for j_2 are scheduled, while reducers from j_1 are suspended. In the meanwhile, new MAP tasks from j_1 are scheduled on available MAP slots. When reducers from j_2 complete (and j_2 finishes), the reducers from j_1 can be resumed: however, note that although machine M_2 has an available REDUCE slot, this cannot be used by any of the suspended reducers from j_1 , due to data locality. When the REDUCE slot on machine M_3 becomes available, the REDUCE task for j_1 can be resumed. Fig. 3 illustrates (through an example) also the effect of data locality on mappers: the 9-*th* mapper from j_1 is not scheduled on machine M_1 as it becomes available, but it is delayed to wait for machine M_2 , which is hosting an un-processed block of data.

³A close look at Fig. 3 reveals that job “serialization” is not as sharp as illustrated in Fig. 1.

⁴The default behavior in Hadoop is that reducers are launched when 5% of the mappers have completed. As such, reducers can initiate intermediate data transfers; however, the REDUCE function is only executed once all mappers from a job have completed.

⁵Precisely, when one mapper is finished

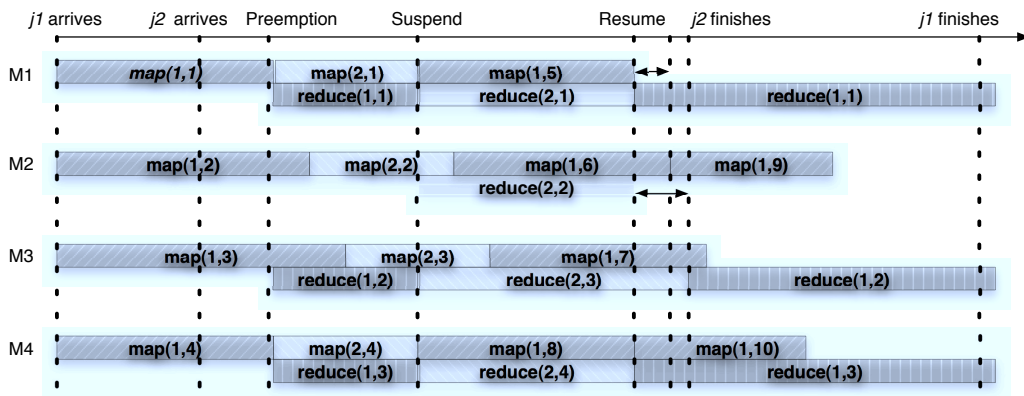


Figure 3: A detailed scheduling example with FSP.

4.2 Optimized jobs

MapReduce job optimization consists in tuning the parameters of the framework, and in using *combiners*, so as to minimize I/O. Simply stated, given the characteristics of the machines of the MapReduce cluster, an optimized job rarely writes on (the local) disk, reduces the amount of intermediate data to be *shuffled*, and performs sorting operations in memory. As such, each job must be properly tuned in order to use at best the resources available to a machine.

With the default scheduler, namely a FIFO approach, jobs are serialized and can thus fully exploit the resources of each machine. Alternative scheduling approaches appeared in the literature, instead, tend to parallelize as much as possible jobs, so as to achieve a fair sharing of cluster resources. This has the negative impact of hindering job optimization because it is impossible to predict the number (and the nature) of concurrent jobs scheduled on each machine. As such, from the practical perspective, a scheduling approach that achieves fairness while granting the job designer with the possibility of tuning each job, is a desirable property.

Our approach to scheduling achieves exactly what discussed above: when jobs require all cluster resources, they have access to the full capabilities of each machine they are scheduled on, in contrast to having the “illusion” of running alone in the cluster. When jobs are small, they share the cluster and can be processed in parallel, as shown in Fig. 2.

5. CONCLUSIONS AND FUTURE WORK

In this work we presented the design of a new scheduling algorithm for a shared cluster dedicated to MapReduce jobs. In contrast to previous works, that abstract away parts of or the whole system, we pinpoint the impact of scheduling on the inner components and phases of MapReduce jobs, paying attention to current best practices in job optimization. Such an approach allowed us to extend the concept of data locality and apply it to the REDUCE phase, which is a desirable feature especially when reducers can be suspended instead of being simply killed, as done in previous works.

The scheduling algorithm presented in this article, labelled FSP, achieves two objectives that have only been considered separately in the literature: fairness in resource allocation and job performance. Currently, we focus on the implementation of FSP: inspired by the work in [11], we plan to issue a JIRA and contribute the the Hadoop community

with a “contrib” FSP module. Subsequently, we will proceed with a thorough experimental evaluation of FSP and compare its behavior with other available schedulers, for a variety of job types (short and long) and composition (number of mappers and reducers).

6. REFERENCES

- [1] Hadoop: Open source implementation of mapreduce. <http://lucene.apache.org/hadoop/>.
- [2] H. Chang, M. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee. Scheduling in MapReduce-like Systems for Fast Completion Time. In *Proc. of IEEE INFOCOM*, 2011.
- [3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.
- [4] E. Friedman and S. Henderson. Fairness and efficiency in web server protocols. In *Proc. of Sigmetrics*, 2003.
- [5] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resources types. In *Proc. of NSDI*, 2011.
- [6] M. Harchol-Balter. Queueing disciplines. In *Wiley Encyclopedia Of Operations Research and Management Science*. John Wiley & Sons, 2009.
- [7] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, 2007.
- [8] K. Kc and K. Anyanwu. Scheduling Hadoop jobs to meet deadlines. In *Proc. of CloudCom*, 2010.
- [9] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In *Proc. of Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.
- [10] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A slot allocation scheduling optimizer for MapReduce workloads. In *Proc. of International Middleware Conference*, 2010.
- [11] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of ACM EuroSys*, 2010.