Locality-aware batch scheduling of I/O intensive workloads

***Abstract*—Clusters make use of workload schedulers such as the Slurm Workload Manager to allocate computing jobs onto nodes. These schedulers usually aim at a good trade-off between increasing resource utilization and user satisfaction (decreasing job waiting time). However, these schedulers are typically unaware of jobs sharing large input files, which may happen in data intensive scenarios. The same input files may be loaded several times, leading to a waste of resources.**

We study how to design a *data-aware job scheduler* that is able to keep large input files on the computing nodes, without impacting other memory needs, and can use previously loaded files to *limit data transfers in order to reduce the waiting times of jobs*.

We present three schedulers capable of distributing the load between the computing nodes as well as re-using an input file already loaded in the memory of some node as much as possible.

We perform simulations using real cluster usage traces to compare them to classical job schedulers. The results show that keeping data in local memory between successive jobs and using data locality information to schedule jobs allows a reduction in job waiting time and a drastic decrease in the amount of data transfers.

***Index Terms*—Job input sharing, Data-aware, Job scheduling, High Performance Data Analytics**

I. Introduction

To meet the ever-increasing demand for scientific computation power, High-Performance Computing platforms are typically composed of large numbers of computation nodes. Scientists typically submit their computation jobs to a scheduler, which decides the ordering and mapping of the jobs on the platform. This needs to be performed with particular care of balancing between resource utilization and user satisfaction, so as to leverage the computation resources as efficiently as possible, while avoiding adverse pathological cases that could particularly impact some users rather than others.

Computation jobs however need data input which, more often than not, can be very large, notably for many subfields of life science with highly data-dependent workflows. Loading such data input from the storage nodes may consume a significant part of the job duration. This load penalty can however be avoided altogether when the data was actually already used by the previous job running on the computation node, and thus still immediately available on the node. Taking care of scheduling jobs that use the same data input one after the other thus allows to reduce the jobs completion times, leading to better platform usage efficiency. Unfortunately, classical job schedulers mostly do not take data input into account, and thus do not benefit from such data reuse ; most jobs always have to re-load their data input.

In this paper, we propose to model the benefits of re-using data loads between successive jobs, and we introduce new algorithms that add such data reuse to the scheduling balance. By tracking which data is loaded on which node for the scheduled jobs, they are able to significantly reduce data loads, thus improving both resource utilization and user satisfaction. We evaluated these algorithms thanks to traces of actual jobs submissions observed on a large cluster platform. This allows to assess the effectiveness of our heuristics over a variety of realistic working sets. This revealed that while our heuristics get slightly worse results over some working set samples (those which exhibit cluster ample underuse), most working set samples largely benefit from our heuristics, leading to interesting benefit overall.

We thus present the following contributions in this paper:

- We formalize our model of scheduling data-intensive jobs sharing input files on a cluster (Section III).
- We propose three new schedulers focusing on re-using input files while minimizing evictions and avoiding starvation (Section IV).
- We extract job information from historical logs of a cluster to build workloads that correspond to the needs and behaviors of real users (Section V-A).
- We implement all three heuristics as well as two state-of-the-art schedulers on a simulator and study the performance (mean stretch and amount of time spent waiting for a file to be loaded) obtained on 44 different workloads (Section V). Our evaluation demonstrate that our heuristics in most cases surpass the state of the art schedulers. We show that workloads that heavily saturate the cluster benefits much more from our strategies which results considerable reduction in the job waiting times.

II. Related work

A. Scheduling jobs on large clusters

A various number of workloads managers have emerged as a way to meet the rising numbers of high performance computing clusters. Workload managers like Slurm [1], OAR [2], TORQUE [3], LoadLeveler [4], Portable Batch System [5], SunGrid [6] or Load Sharing Facility [7] all offer various scheduling strategies.

The First-Come-First-Served (FCFS) algorithm is the prevalent default scheduler on most of these solutions [8]. Moreover, Slurm is used on most of the TOP500 supercomputers and its default strategy is FCFS [9] as well. A backfilling strategy is known to increase the use of supercomputer

resources [10] [11]. The most commonly used backfilling strategy, called conservative backfilling [12] [13] follows a simple paradigm: "a job can only be backfilled if it does not delay a job of higher priority". We can then safely assume that comparing ourselves to FCFS with and without conservative backfilling can bring significant insights on what improvements can be achieved on data-intensive workloads.

Other scheduling strategies exist like Maui [14], Gang scheduling [15], RASA [16] that use the advantages of both Min-min and Max-min algorithms, RSDC [17] that divides large jobs in sub jobs for a more refined scheduling, or PJSC [18] and PSP+AC [19] that are a priority-based schedulers; however these heuristics do not consider the impact input re-use could have on data-intensive workloads. We aim at resolving this issue in this paper.

B. Using distributed file systems to deal with data-intensive workloads

Distributed file systems are a solution to ease the access to shared input files. They facilitate the execution of I/O-intensive batch jobs by selecting appropriate storage policies.

Batch-Aware Distributed File System [20], is designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters. HDFS [21] (Hadoop Distributed File System) incorporates storage-aware scheduling. It migrates a computation closer to where the data is located, rather than moving the data to where the application is running, in order to reduce communication.

These solutions are mainly storage systems that uses a history of file locations to serve as a backup. In our scenario, we copy the data from an already-redundant system (an online database for example) and store it locally on the node in an ephemeral way. Thus, in the event of a crash, we do not manage the data which is already redundant, it simply results in an aborted job. Secondly, the scheduling can cause issues. Weets et al. describe some problems from MapReduce [22], the programming model used in HDFS, in detail. By not using HDFS or any distributed file system, we avoid these problems altogether. Lastly, file systems are particularly efficient when the input data used are identical over time. In our case, between users, the inputs will be largely different, making distributed file systems less efficient.

C. Using schedulers to deal with data-intensive workloads

Some schedulers tackle the issue of data-intensive workloads. A solution can be to minimize network contention by allocating nodes to even out node and switch contention [23]. In our model, we are not studying the network topology and consider independent nodes. This is reasonable, since our main concern is the cross-section bandwidth to a shared storage solution. Nikolopoulos et al. [24] focus on a better utilization of idling memory together with thrashing limitation. Our focus will be to control data loads in order to limit eviction and will thus naturally limit thrashing. Agrawal et al. [25] propose to schedule jobs not sharing a file first and to use a stochastic model of job arrivals for each input file to maximize re-use. This work is aimed at the Map-Reduce model and allows to predict future jobs arrivals, two prerequisites that we do not consider. Equipping each node with a scheduler that follows a work stealing strategy in order to manage both data locality and load balancing is also a solution to reduce data transfers [26]. Selvarani et al. propose an improved activity-based costing scheduler [27] where the processing capacity of each resource is evaluated to make the right decision. Our approach is more focused on maximizing data re-use on a set of identical nodes.

III. Framework

We consider the problem of scheduling a set of N independent jobs, denoted $\mathbb{J} = \{J_1, J_2, \ldots, J_N\}$ on a set of P nodes: $\mathbb{N} = \{\mathrm{Node}_1, \mathrm{Node}_2, \ldots, \mathrm{Node}_P\}$. Each node Node_i is equipped with m cores noted: $c_1^i, \ldots, c_m^i$ sharing a memory of size M.

Each job depends on an input file noted $File(J_i)$, which is initially stored in the main shared file system. During the processing of a job J_i on Node_k, file $File(J_i)$ must be in the memory of Node_k. If this is not the case before starting computation of job J_i, then file $File(J_i)$ is loaded into the memory. We denote by $\mathbb{F} = \{F_1, F_2, \ldots, F_n\}$ the set of distinct input files, whose size is denoted by $Size(F_i)$. Each job runs on a single node, but on different numbers of cores.

Each job J_i has the following attributes:

- Resource requirement: job J_i requests $Cores(J_i)$ cores, such that $1 \leq Cores(J_i) \leq m$;
- Input file: $File(J_i) \in \mathbb{F}$;
- Submission date: $SubTime(J_i)$;
- Requested running time (or walltime): $WallTime(J_i)$: if not finished after this duration, job J_i is killed by the scheduler;
- Actual running time: $Duration(J_i)$ (unknown to the scheduler before the job completion).

We do not consider the data output of jobs. Each of the N jobs must be processed on one of the P nodes. As stated earlier, the shared file system initially contains all files in $\mathbb{F}$. Each node is connected to the file system with a link of limited bandwidth, denoted by $Bandwidth$: transferring a data of size S from the shared file system to some node's memory takes a time $S/Bandwidth$. The limited bandwidth as well as the large file sizes are the reasons why we aim at restricting the amount of data movement.

We consider that the memory of a node, denoted by M is only used by the jobs' input files, since all other data are negligible compared with the input files. We assume that jobs are devoted a fraction of the memory proportional to the number of requested cores, so that jobs willing to process large input files must request large number of cores. This way, we make sure that the memory of a node is large enough to accommodate all input files of running jobs. A file stored in the memory of a node can be shared by two jobs J_i and J_j only in the following situations:

1) J_i and J_j are computed in parallel on the same node.

2) J_i and J_j are computed on the same node consecutively (i.e., no job is started on this node before the completion of J_i and the start of J_j).

This can hold true if the file data is accessed through I/O (traditional or memory-mapped), allowing the same page cache to serve multiple processes from different jobs. Otherwise we consider that memory operations of jobs scheduled between J_i and J_j will cause the file to be evicted.

For each job J_i, the scheduler is in charge of deciding which node will process J_i, and more precisely which cores of this node are allocated to the job, as well as a starting time t_k for J_i. Job J_i is thus allocated a time window from t_k to $t_k + WallTime(J_i)$ devoted to (i) possibly loading the input file $File(J_i)$ in the memory (if it is not already present at time t_k) and (ii) executing job $J(i)$. If the job has not completed at time $t_k + WallTime(J_i)$, it is killed by the scheduler to ensure that later jobs are not delayed. The scheduler must also make sure that no two jobs are executed simultaneously on the same cores.

It is important to note that the file transfer is done before the computation and cannot be overlapped. Jobs are non-preemptible: when started, a job is executed through its completion.

Our objective is to reach an efficient usage of the platform and to limit job waiting times. Each user submitting jobs is interested in obtaining the result of jobs as soon as possible. Hence we focus on the time spent in the system for each job, also called the flow time (or flow) of the job:

$$Flow(J_i) = CompletionTime(J_i) - SubTime(J_i).$$

In the following, we want to consider aggregated performance metrics on job flows, such as average flow. However, the duration of a job significantly impacts its flow time. Jobs with the same flow but very different durations do not experience the same quality of service. To avoid this, the *stretch* metric has been introduced that compares the actual flow of a job to the one it would experience on an empty cluster:

$$\begin{aligned} ReferenceFlow(J_i) &= Duration(J_i) + \frac{Size(File(J_i))}{Bandwidth} \\ stretch(J_i) &= \frac{Flow(J_i)}{ReferenceFlow(J_i)}. \end{aligned}$$

The stretch represents the slow-down of a job due to sharing the platform with other jobs. Considering the stretch allows to better aggregate performance from small and large jobs.

IV. Job scheduling with input files

Here, we present various schedulers used to allocate jobs to computing resources. We start with two reference schedulers (FCFS and EFT) and then move to proposing three locality-aware job schedulers (named LEA, LEO and LEM). Each of these five schedulers can be used both without or with backfilling. For the sake of clarity, we first present the simpler version, without backfilling, before detailing the modifications needed to including backfilling.

As presented above, the task of the scheduler is to allocate a set of cores of one node to each job submitted until now: some jobs may be started right away, while other jobs may be delayed and scheduled later: resource reservations are made for these jobs. Jobs are presented to the scheduler in the form of a global queue, sorted by job submission time. These scheduling policies are online algorithms which are called each time a task completes (making cores available) or upon the submission of a new job.

A. Two schedulers from the state of the art: FCFS and EFT

A widely-used job scheduler that is typically considered to be efficient is First-Come-First-Serve (FCFS), detailed in Algorithm 1. Implementing this scheduler requires to remember the time of next availability for each core. Then, for each job, we look for the first time when a sufficient number of cores is available, and we allocate the job to those cores.

Algorithm 1 First-Come-First-Serve (FCFS)

1: **for each** $J_i \in$ the jobs queue **do**
2: **for each** $\mathrm{Node}_k \in \mathbb{N}$ **do**
3: Find smallest time t_k such that $Cores(J_i)$ cores are available on Node_k
4: Select Node_k with the smallest t_k
5: Schedule J_i on $Cores(J_i)$ cores of Node_k that are available starting from t_k
6: Mark these cores busy until time $t_k + WallTime(J_i)$

FCFS is a standard baseline comparison for job scheduling. However, it is not aware of the capability of the system to keep a large data file in the memory of a node between the execution of two consecutive jobs. A first step towards a locality-aware scheduler is to select a node for each job not based on the cores' availability time, but also using the file availability time, based on the file transfer time. This is the purpose of the Earliest-Finish-Time (or EFT) scheduler, described in Algorithm 2: by selecting the node that can effectively start the job at the earliest time, it minimizes the job completion time. There are three scenarios to compute the time t'_k at which the input file of a job J_i is available on Node_k:

1) A job started before J_i on Node_k uses the same input file and it is already in memory, thus $t'_k = t_k$;
2) The input file of J_i is not in memory and $t'_k = t_k + \frac{Size(File(J_i))}{Bandwidth}$;
3) The input file of J_i is partially loaded on Node_k: this happens when some job J_j using the same input file has been scheduled on other cores of the same node at time $StartTime(J_j) < t_k$ but the file transfer has not completed at time t_k. Then the file will be available at time: $t'_k = StartTime(J_j) + \frac{Size(File(J_i))}{Bandwidth}$.

B. Data-locality-based schedulers

The previous strategies focus on starting (FCFS) or finishing (EFT) a job as soon as possible, respectively. Those are good methods to avoid node starvation and reduce queue times.

Algorithm 2 Earliest-Finish-Time (EFT)

for each $J_i \in$ the jobs queue **do**
  for each $\text{Node}_k \in \mathbb{N}$ **do**
    Find smallest time t_k such that $Cores(J_i)$ cores are available Node_k
    Find time $t'_k \geq t_k$ at which $File(J_i)$ is available on Node_k
  Select Node_k with the smallest t'_k
  Schedule J_i on $Cores(J_i)$ cores of Node_k that are available starting from t_k
  Mark these cores busy until time $t_k + WallTime(J_i)$

However, they may lead to loading the same input file on a large number of nodes in the platform, only to minimize immediate queue times. Time is thus spent loading the input file multiple times. This can affect the global performance of the system by delaying subsequent jobs. We present three strategies that attempt to take data locality into account in a better way to reduce queuing times in the long run by increasing data reuse.

The first proposed algorithm called Locality and Eviction Aware (LEA) and detailed in Algorithm 3 aims at a good balance between node availability and data locality. We consider three quantities to rank nodes:

- The availability time for computation t_k;
- The time needed to complete loading the input file for the job on Node_k ($t'_k - t_k$);
- The time required to reload files that need to be evicted before loading the input file; this time is computed using all files in memory and considering that a fraction of these files need to be evicted, corresponding to the fraction of the memory used by the job.

The intuition for the third criterion is that if loading a large file in memory requires the eviction of many other files, these files will not be available for later jobs and may have to be reloaded. In the LEA strategy, we put a strong emphasis on data loading, in order to really favor data locality. Hence, when computing the score for each node Node_k, we sum the previous three quantities, with a weight W for the second one (loading time). In our experiments, based on empirical evaluation, we set this value to $W = 500$, incidentally roughly equivalent to the number of nodes. Note that the other two quantities usually have very different values: the availability time is usually much larger than the time for reloading evicted data. Hence this last criterion is mostly used as a tie-break in case of equality of the first two criteria.

The LEA strategy puts a dramatic importance on data loads. Hence, it is very useful when the platform is fully loaded and some jobs can safely be delayed to favor data reuse and avoid unnecessary loads. However, when the platform is not fully loaded, delaying jobs can be detrimental, as it can increases the response time for some jobs, without any benefit for other jobs. Our second proposed strategy, named Locality and Eviction Opportunistic (LEO) and described in Algorithm 4, tries to adapt based on the current cluster load: if we find some nodes that can process the job right away, we select the one that will minimize the completion time (as in the EFT strategy). Otherwise, we assume that the platform is fully loaded and we apply the previous LEA strategy, to favor data reuse.

We present a third strategy called Locality and Eviction Mixed (LEM) and described in Algorithm 5 that takes a similar approach to LEO but performs a simple mix between the EFT and the LEA strategies: when the load of the platform (measured as the number of nodes running at least one job) is above a criterion, the LEA strategy is applied, otherwise the EFT strategy is used.

Algorithm 3 Locality and Eviction Aware (LEA)

for each $J_i \in$ the jobs queue **do**
  for each $\text{Node}_k \in \mathbb{N}$ **do**
    Find smallest time t_k such that $Cores(J_i)$ cores are available
    Find time $t'_k \geq t_k$ at which $File(J_i)$ is available on Node_k
    $LoadOverhead \leftarrow t'_k - t_k$
    Let $\mathcal{F}$ be the set of files in the memory of node Node_k at time t_k
    $EvictionPenalty \leftarrow (\sum_{F_j \in \mathcal{F}} Size(F_j) \times Size(File(J_i))/M)/Bandwidth$
    $score_{\text{Node}_k} \leftarrow t_k + W \times LoadOverhead + EvictionPenalty$
  Select Node_k with the smallest $score_{\text{Node}_k}$
  Schedule J_i on $Cores(J_i)$ cores of Node_k that are available starting from t_k
  Mark these cores busy until time $t_k + WallTime(J_i)$

Algorithm 4 Locality and Eviction Opportunistic (LEO)

for each $J_i \in$ the jobs queue **do**
  for each $\text{Node}_k \in \mathbb{N}$ **do**
    Find smallest time t_k such that $Cores(J_i)$ cores are available
    Find time $t'_k \geq t_k$ at which $File(J_i)$ is available on Node_k
    $LoadOverhead \leftarrow t'_k - t_k$
    Let $\mathcal{F}$ be the set of files in the memory of node Node_k at time t_k
    $EvictionPenalty \leftarrow (\sum_{F_j \in \mathcal{F}} Size(F_j) \times Size(File(J_i))/M)/Bandwidth$
    if $t_k = current\_time$ **then**
      $score_{\text{Node}_k} \leftarrow t'_k$
    else
      $score_{\text{Node}_k} \leftarrow t_k + W \times LoadOverhead + EvictionPenalty$
  Select Node_k with the smallest $score_{\text{Node}_k}$
  Schedule J_i on $Cores(J_i)$ cores of Node_k that are available starting from t_k
  Mark these cores busy until time $t_k + WallTime(J_i)$

Algorithm 5 Locality and Eviction Mixed (LEM)

1: **for each** $J_i \in$ the jobs queue **do**
2: $load \leftarrow$ percentage of nodes running at least one job at current time
3: **if** $load < 80$ **then**
4: $EFT(\mathbb{J}, \mathbb{N})$
5: **else**
6: $LEA(\mathbb{J}, \mathbb{N})$

C. Adding backfilling to all strategies

As mentioned above, backfilling has been proposed to increase the performance of job schedulers, by allowing jobs with lower priority to be scheduled before jobs with higher priority. In our setting, the priority is directly linked to the submission order: if J_i is submitted before J_j, then J_i has a higher priority than J_j. In order to avoid jobs being perpetually delayed, bounds have to be set on how already-scheduled jobs can be affected by backfilling. As discussed above, Conservative Backfilling is the most restrictive version and one of the most commonly-used strategies to improve cluster utilization. It forbids any modification on the resource reservations of high-priority jobs: a job may be scheduled before other jobs that appear earlier in the queue, provided that it does not impact the starting time of these jobs.

For each of the previous scheduling strategies, we consider a variant using conservative backfilling (suffixed by -BF). To add backfilling, Algorithms 1, 2, 3 and 4 have to be modified: we change the choice of the earliest time when resources are available for a job (Line 3). Instead of considering the time at which cores are (indefinitely) available, we look for an availability time window starting at t_k long enough to hold the job. Specifically, we change Line 3 into:

3′: Find smallest time t_k such that $Cores(J_i)$ cores are available from t_k until $t_k + WallTime(J_i)$ on Node_k

Note that this requires the schedulers to store the whole occupation profile of each core (with availability and unavailability times), whereas the version of each scheduler without backfilling simply requires the time of last job completion on each core.

V. Performance evaluation and analysis

Actual workloads at HPC resources shared by a great number of users with diverse needs can contain structures that are non-trivial to replicate in a fully artificial simulated job pattern. We believe that this is especially true for data-dependent patterns, where a project might launch a burst of jobs using the same file just a few thousand core hours in length, then be quiet for a long time processing the results, and then launch another such burst. However, changing the scheduling strategy of a production cluster for scheduling research would disrupt the community using this cluster.

For this reason, we choose to perform simulations based on logs of a real computing platform. In this section, we describe how we used these logs as well as the results of the corresponding simulations. All strategies as well as the two baselines have been implemented on a simulator that we developed[1].

A. Usage of real cluster logs

The logs we use contain historical data on jobs, namely their exact submission time, size, stated walltime, and actual duration. Since explicit data dependencies are not encoded in SLURM job specifications, we do not have access to the actual input files of these jobs. We thus create an artificial data dependency pattern that replicates user behaviors. We consider two jobs J_i and J_j. These jobs are considered to share their input file if they match the three following requirements:

1) $Cores(J_i) = Cores(J_j)$, i.e., they are requesting the same number of cores.
2) J_i and J_j were submitted by the same user.
3) J_i and J_j were submitted within an 800 seconds time frame. We consider this timeframe to be a reasonable amount of time for a user to submit all of their jobs using the same input file.

Otherwise, we consider that J_i and J_j are using distinct input files. The platform where the logs have been extracted consists of 486 nodes with 20 cores each, with most of them having 128GB of RAM, and a few larger nodes. To avoid an additional constraint, while maintaining a model close to real clusters, we consider that the platform is made of 486 homogeneous nodes of size $M = 128GB$.

We consider that these jobs are dedicated to processing their input file. Hence, the more cores the job requests, the larger its input file. This allows to compute the size of files as follows:

$$Size(File(i)) = \frac{Cores(J_i)}{20} \times 128GB$$

The utilization levels in the log of the platform are typically high ($> 90\%$), but not fully consistently so. The vast majority of jobs on these resources are single-node jobs and thus fit in our framework. The few multi-nodes jobs are not representative of the typical usage of the platform, and are replaced by as many single-node jobs as necessary to represent the same workload. We notice that job durations extend up to 10 days, while some jobs only last a few minutes. Even if the workload is not homogeneous, it is representative of the real usage from an actual user community including, but not exclusively consisting of, many subfields of the life sciences with highly data-dependent workflows.

In order to avoid simulating the whole workload (one year) but to target different scenarios, we randomly select a number of days (44 days) and extract the jobs corresponding to these days from the logs. However, to simulate these jobs in a realistic, steady-state operation of the platform, we consider both jobs submitted before and after the day under consideration. More precisely, we proceed as illustrated in Figure 1.

[1]See code and anonymized logs at: https://github.com/userccgrid/Locality-aware-batch-scheduling

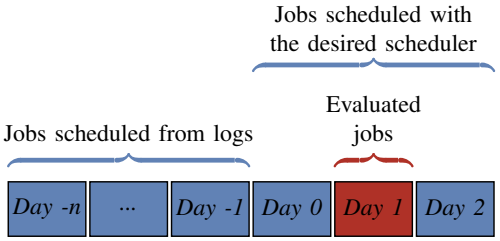


Figure 1: Methodology followed to schedule and evaluate jobs from a specific day while avoiding edge effects.

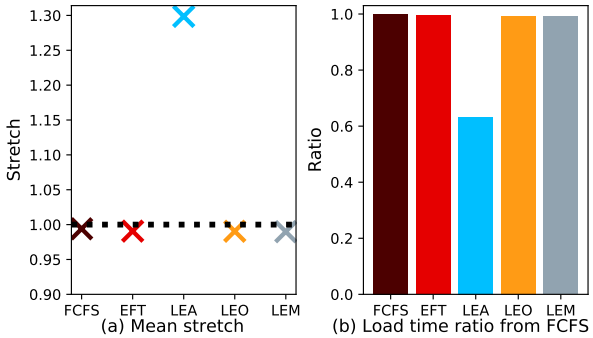


Figure 2: Average stretch of all jobs evaluated and total transfer time on July 16.

We evaluate the mean stretch and the total amount of time spent waiting for a file to be available. Our main competitors on these metrics are FCFS and EFT, with and without backfilling.

B. An underutilized cluster, LEA issues and the appeal of LEM

We first study the behavior of our schedulers on different workload conditions, offering different saturations of the cluster and start with an underloaded cluster. Figure 2 depicts the average stretch obtained by each strategy. In this figure, the horizontal black dotted line corresponds to a stretch of 1, which is the value all jobs would get it there were scheduled on an empty cluster (it corresponds to the waiting time of each job being exactly the time it takes to load the input file). As the cluster is not very loaded in this scenario, FCFS, EFT, LEO and LEM have mean stretches close to 1. EFT, LEO and LEM have a slight benefit over FCFS thanks to file re-use. LEA has a stretch 23% slower.

Figure 2b shows the total amount of time spent waiting for a file to be ready before starting the computation, relative to the total waiting time of FCFS. On this workload, LEA re-uses a file for 6205 out of the 7434 evaluated jobs. All the other strategies re-use a file for around 5500 jobs. In this case, LEA is the only strategy that increases the file re-use, which explains why the total time spent waiting for a load is lower.

a) Understanding LEA's poor performance: To understand the issues LEA encounters on this workload, we need to study the cluster's usage over time. Figure 3 shows the cluster usage over time when using FCFS. The vertical axis

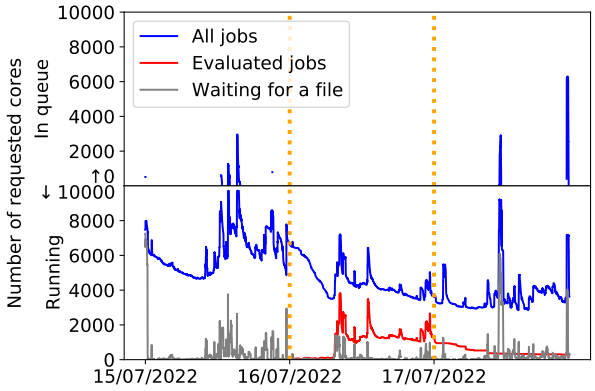


Figure 3: Visualization of the utilization rate of the cluster on the workload of July 16 with FCFS.

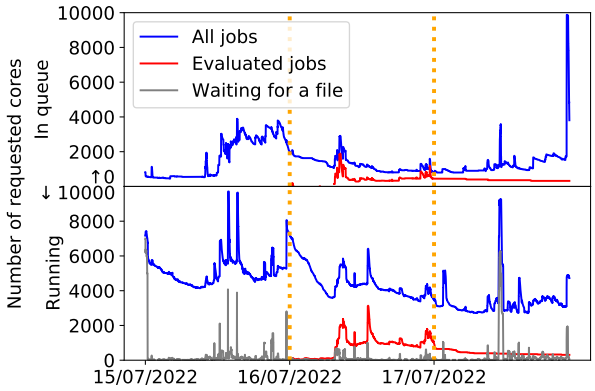
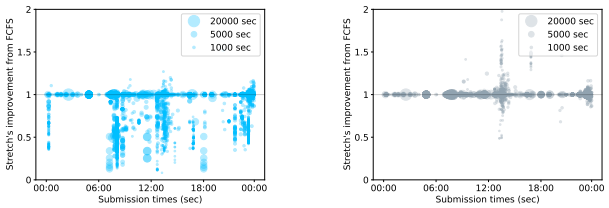


Figure 4: Visualization of the utilization rate of the cluster on the workload of July 16 with LEA.

represents the number of requested cores either running on a node (lower half) or in the queue of jobs waiting to be executed (top half). The maximum is the total number of cores on our cluster: 9720. The blue line shows the number of cores from all jobs. The red lines show the number of cores used by the evaluated jobs, i.e., those jobs that have their submission time within our evaluation window. Thus, this forms a subset of the set indicated by the blue line. If a line is present in the top half, it means that some jobs could not be scheduled and are thus waiting in the queue of available jobs. A node may have some available cores but not enough to accommodate some jobs with large requirements. This explains why the waiting job queue may not be empty even if the lower half does not reach the maximum. The gray line represents the number of cores currently loading a file. Lastly, the orange lines delimits the submission times of our evaluated jobs, in other words it is our evaluation window.

By looking at the top half of Figure 3 we understand that the job queue is empty most of the time. In this situation, FCFS and EFT are very efficient. The earliest available node is in most cases a node that can start the job immediately, explaining the mean stretch close to 1 in Figure 2a. LEO is



(a) With LEA. (b) With LEM.

Figure 5: Stretch times of each job compared to FCFS on the workload of July 16.

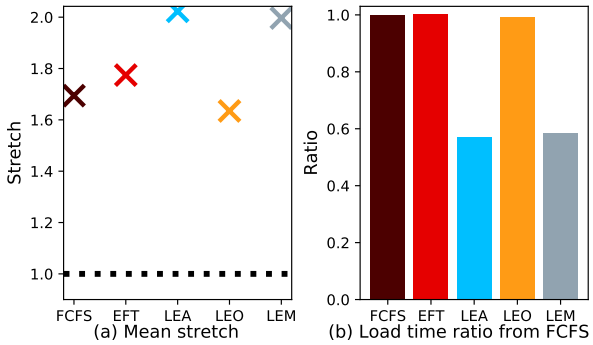


Figure 6: Average stretch of all jobs evaluated and total load time on September 09.

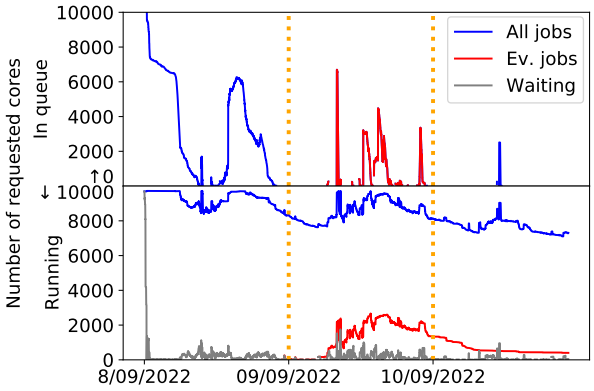


Figure 7: Visualization of the utilization rate of the cluster on the workload of September 9 with FCFS.

a strategy that uses the earliest available time t_k of a node to decide if it should compute a score like LEA, that puts a large weight on transfer time or weigh equally t_k, the transfer and eviction durations. Thus, on underused clusters, LEO has a behavior close to EFT, while trying to minimize evictions. Similarly, LEM switches between EFT and LEA depending on the cluster's usage. On this particular workload the cluster's usage is under 80%, so LEM behaves similarly to EFT. On the contrary, LEA favors data re-use over an early start time for a job. On an underused cluster, this increases starvation. We can notice this when looking at Figure 4, which shows the cluster usage when using LEA. LEA uses fewer cores, notably before the first vertical orange dotted line. This translates into a larger queue of jobs visible on the top half of the figure. For LEA, most of the jobs in this queue are jobs that already have a valid copy of their file loaded on a node. The benefit of scheduling these jobs immediately on another node and loading the file appears inferior to waiting for a file re-use for LEA and thus creates this queue of jobs that does not exist for FCFS. The consequence for the stretch is immediately distinguishable on Figure 5a. This visualization shows for each job, the ratio of its stretch with LEA by its stretch with FCFS (hence a value above 1 means LEA improves the stretch). The size of a circle is proportional to the job's duration. On the workload of July 16, we observe several "columns" of jobs submitted at the same time (hence sharing the same file with large probability) with a ratio lower than 1, hence a worst performance with LEA. This means LEA is waiting to re-use the files before starting the jobs, whereas FCFS paid the cost of loading the file on other nodes, but started the jobs earlier than LEA, leading to shorter completion times. From Figure 5b concerning LEM, we see that most jobs have an improvement close to 1, showing that LEM does not fall into LEA's pathological case.

To summarize, on an underutilized cluster, LEA's focus on locality does not allow optimal utilization of the cluster, whereas LEO and LEM, thanks to their flexibility, achieve performance identical to EFT.

C. An almost saturated cluster, the weakness of LEM and the resilience of LEO

In the previous section, we saw the benefit of using LEO or LEM over LEA. Here we evaluate our strategies on a workload that almost saturates the cluster, underlining the weakness of LEM.

Despite greatly reducing the amount of file transfers as can be seen on Figure 6b, LEA and LEM do not manage to reach better stretch than FCFS (see Figure 6a). LEO however has a smaller stretch than FCFS. LEA suffers from the same issues as on the last workload: the cluster is not fully used, so the strong focus on locality prevents us from using all available cores. Unfortunately, LEM reaches similar performance, as explained below.

a) A pathological scenario of LEM: The workload of September 9, managed by FCFS (see Figure 7) results in a cluster that alternates between full utilization and an approximately 80% utilization rate. Thus the queue of requested cores alternates between a few thousands and 0.

Figure 8 shows the same visualization with LEM. We see that the queue of requested cores never hits 0. In this case, the occupation rate is above 80%, but far from 100%, thus LEM stays on the same strategy as LEA, as the threshold to switch between EFT and LEA has been set to 80%. This threshold value was found experimentally by testing different thresholds on various types of workloads: choosing a higher value would help for this specific workload, but would decrease the overal performance. As explained above, following LEA's strategy is

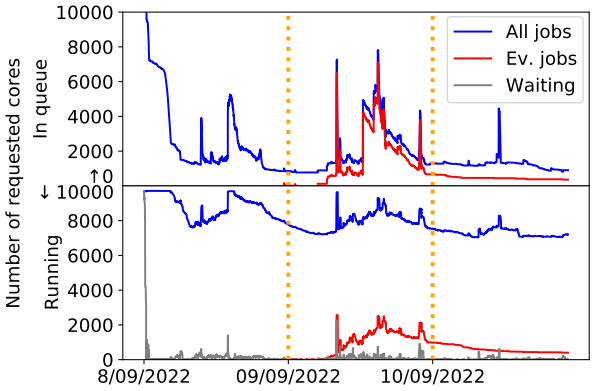


Figure 8: Visualization of the utilization rate of the cluster on the workload of September 9 with LEM.

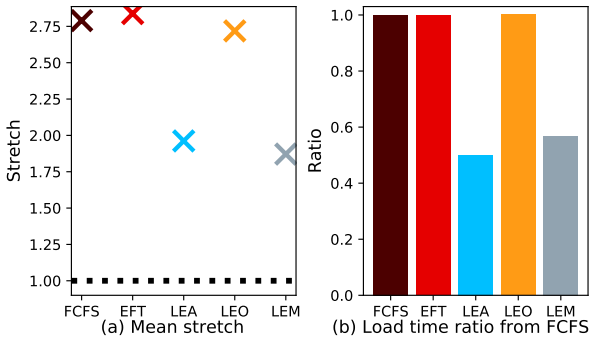


Figure 9: Average stretch of all jobs evaluated and total load time on March 26.

a sub-optimal choice when the cluster is not fully saturated. It creates a queue of jobs that, as long as the cluster's utilization is above 80%, will not be scheduled on a node for a long time unless it can re-use their file. Normally, with this behavior, using LEA leaves a lot of unused nodes that allows LEM to periodically switch to EFT. But in the case of an almost saturated cluster, we still stay above 80% while not completely using the cluster, resulting in a queue of jobs clearly visible on the top half of Figure 8. For the jobs in this queue, the stretch is higher than with FCFS, explaining the poor performance of LEM on Figure 6a.

D. A saturated cluster, the great benefits of LEA and LEM

The workload presented here saturates the cluster with FCFS. Indeed, as we can see on Figure 10, there is a queue of several thousands requested cores for the whole duration of the evaluated day. In this situation, re-using files has a significant impact on the queue times. On this workload, LEA and LEM re-uses files for approximately 3500 of the 4721 evaluated jobs. In contrast, FCFS only re-uses files for 2100 jobs. This is confirmed by Figure 9a: more data re-use is associated with a smaller stretch for LEA and LEM.

On a saturated cluster, filling all cores with the first jobs of the queue, like FCFS does, is not crucial. It is more beneficial

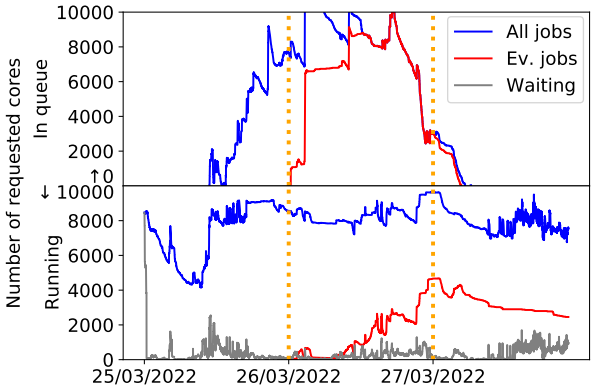


Figure 10: Visualization of the utilization rate of the cluster on the workload of March 26 with FCFS.

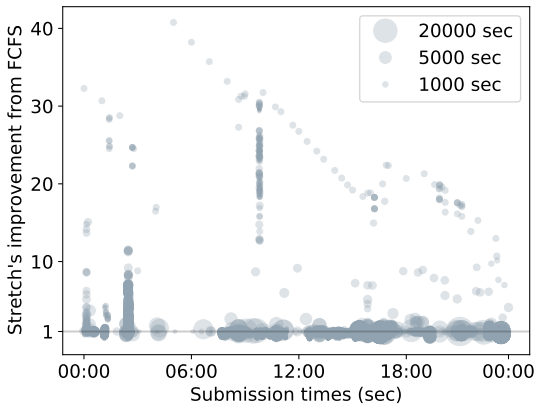


Figure 11: Stretch times of each job of LEM compared to FCFS on the workload of March 26.

to group jobs using the same file. The first few jobs have a longer queue than with FCFS, but, over time, re-using files causes a snowball effect that reduces the queue times of all subsequent jobs. Moreover, the queue contains enough jobs to fill all the nodes even when grouping them by input file. We thus avoid the pathological cases presented in sections V-B and V-C. In this case, LEA's strategy, also found in LEM, allows to greatly reduce the mean stretch. We can observe this, job by job, on Figure 11: very few jobs have a worse stretch than FCFS and a large amount of jobs are above an improvement of 2.

E. A cluster saturated with small jobs, the best case scenario for LEA and LEM

From Figure 12a, we observe that LEA has a stretch 33 times smaller than its competitors. LEM is about 7 times smaller. This drastic diminution is explained by the two particularities of this workload. Firstly, it is a workload that heavily saturates the cluster. So, for the same reasons as in Section V-D, LEA and LEM largely reduce load times. Secondly, it is a workload largely composed of jobs that are

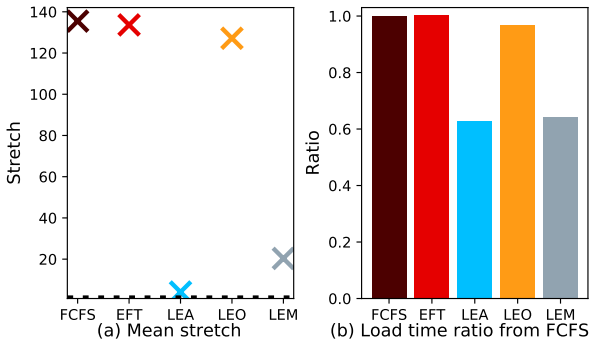


Figure 12: Average stretch of all jobs evaluated and total load time on August 16.

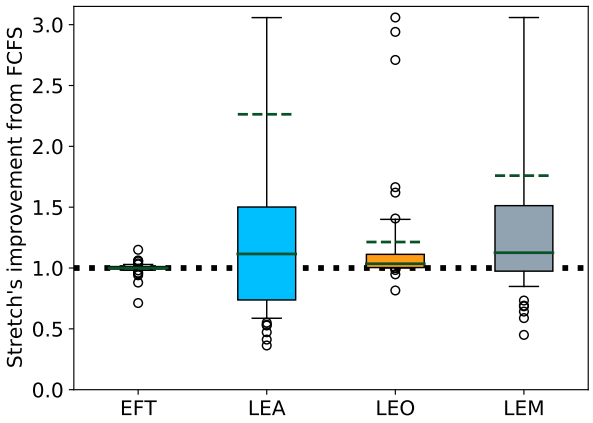


Figure 13: Mean stretch's improvement from FCFS on all evaluated workloads. The whiskers are the octiles. The solid green line represent the median and the dashed one the mean. 6 outliers for LEA and LEM (with maximum value at 33.5) are not depicted for better readability.

short and using less than five cores. On shorter jobs, the proportion of the duration spent on the file transfer is very high. Thus, reducing the transfer time of small jobs has a much greater effect on the stretch, resulting in this drastic reduction for LEA and LEM.

F. Aggregated results on 44 different evaluated workloads

We evaluated our 5 schedulers on 44 different days. Figure 13 represents with box-plots the aggregated results of the ratio of the stretch of each strategy with the one of FCFS. A result above the black dotted line (at 1) is thus an improvement compared to FCFS average stretch. On this figure, the box represents results within the first and the last quartile (from 25% to 75%, thus half of the results). The whiskers delimit the octiles (12.5% and 87.5%), thus 75% of the results are contained within the whiskers. This also means that between the lower side of the box and the maximum value, we find 75% of the results (and similarly between the upper side of the box and the minimum value). The solid green lines show the median while the dashed ones show the mean result. Each

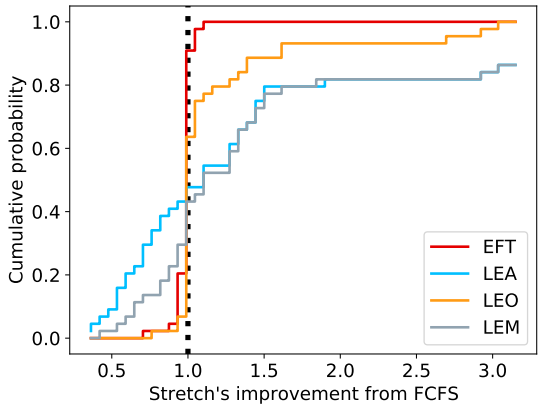


Figure 14: Empirical distribution function of the mean stretch's improvement from FCFS on all evaluated workloads.

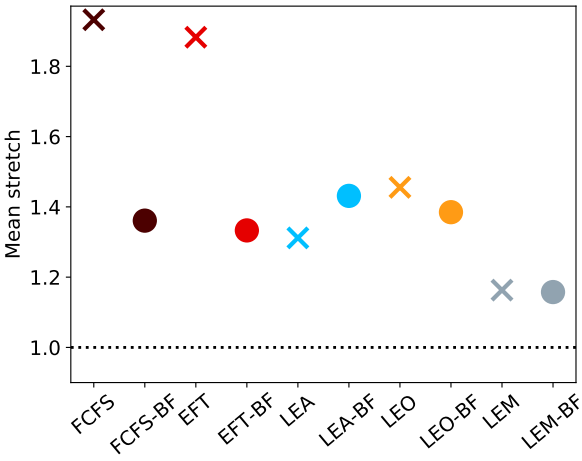


Figure 15: Average stretch of all jobs evaluated on January 28 with (circles) and without (crosses) backfilling.

white circle is an outlier whose improvement is in the first or last octile. For better readability, 6 outliers above the upper whisker for LEA and LEM have been removed, with maximum value respectively at 33.5 and 6.7.

As in the previous results, we observe that EFT does not bring any real improvement compared to FCFS. For LEA, LEO and LEM, the medians are respectively 1.12, 1.03 and 1.13. However the mean values are much higher at 2.26, 1.21 and 1.76.

We can explain the larger mean values for LEA and LEM from the good performance of LEA's strategy on heavily saturated clusters (see Section V-D and V-E). In these cases, the mean stretch values of LEA or LEM are much smaller than those of FCFS or EFT. Re-using the same files is not detrimental to the filling of all the nodes because there are enough jobs to cover all nodes. A large decrease in the time spent waiting for a file greatly reduces the stretch of each job. Compared to LEA, LEM has a smaller mean value. However 75% of its results are above 1, i.e., an improvement, whereas

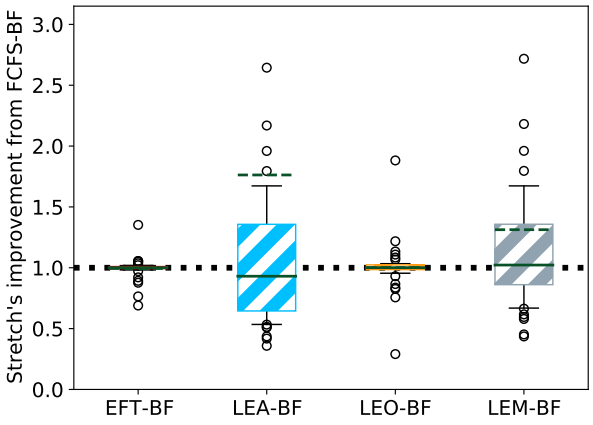


Figure 16: Mean stretch's improvement from FCFS-BF on all evaluated workloads.

for LEA, only approximately 57% of the results are above 1. LEM is a more versatile strategy and offers higher sustained performance on non-saturated cluster at the cost of fewer extreme improvements on heavily saturated clusters.

From the same data shown on Figure 13, we plot an empirical distribution function on Figure 14. EFT's low variance is clearly visible in the sudden jump in probability around an improvement of 1. It is interesting to note that for LEA and LEM, 20% of the results are above an improvement of 300%. In addition, thanks to its switch between the strategies of EFT and LEA, LEM clearly reduces performance losses on the left of the black line. We can also learn from this figure that LEO is in every respect a better version of EFT. At no point, it shows a higher cumulative probability before 1. Above 1 it shows significantly better results beyond an improvement of 20%.

Thus, without backfilling, LEM is the best strategy to observe significant improvements. It can compute jobs between 1 and 1.5 times faster in 50% of the cases, between 1.5 and 3 times faster in 12.5% of the cases and between 3 and 6.7 times faster in 12.5% of the cases. It is slower in only 25% of the workloads, and 12.5% of those are within a 0.15 slow-down. LEO is a more sustained strategy with 87.5% of its results with an improvement compared to FCFS, which shows that our opportunistic strategy is much more consistent, while still having great improvements in some cases, as can be seen with the outliers above the 1.5 mark.

Figure 16 shows the results with the backfilling version of our schedulers and compared to FCFS with backfilling (FCFS-BF) on all workloads. We notice that our schedulers have smaller improvements with backfilling. Figure 15 shows that with backfilling (circles), the mean stretch is much lower for FCFS-BF and EFT-BF. However, our proposed strategies do not benefit from backfilling as much as FCFS-BF does for two reasons:

- Even if we consider data locality when backfilling, trying to fill a node as much as possible and optimizing data re-use are two contrary goals. Backfilling a job can compromise a re-use pattern that was planned by our locality-aware strategy, thus reducing the total amount of re-used files.
- Our strategies are already able to nicely fill the nodes without needing backfilling. Grouping jobs by input file implies that similar jobs end up on the same nodes. Jobs having the same duration and number of requested cores can much more easily fill a node to its fullest than a completely heterogeneous set of jobs. Consequently, the shortfall without backfilling is much lower for LEA-BF, LEO-BF and LEM-BF, than for FCFS-BF and EFT-BF.

Compared to FCFS-BF, our strategies still reduce the total queue time with backfilling. However, the difference is less significant. One can observe an example of this in Figure 15 by looking at LEM and LEM-BF compared to FCFS and FCFS-BF.

Figure 16 shows that the improvement of both EFT-BF and LEO-BF compared to FCFS-BF is not significant. LEA-BF has a median slightly worse than our competitor. However it still achieves a better average result.

Out of our four heuristics, LEM-BF is the best compromise. It is better than FCFS-BF in more than 50% of the cases, with 25% of those results above an improvement of 1.35. Moreover, improvements are still important on heavily saturated clusters, as can be seen with the outliers. Among the slow-downs, only 25% are superior to 0.15.

VI. Conclusion

Batch schedulers are key components of computing clusters, and aim at improving resource utilization as well as decreasing jobs' response time. We have studied how one may improve their performance by taking job input file into account: in clusters dedicated to data analysis, users commonly submit dozens jobs using the same multi-GB input file. Classical job schedulers are unaware of data locality and thus fail to re-use data on nodes. We have proposed three new locality-aware strategies, named LEA, LEO and LEM, capable of increasing data locality by grouping together jobs sharing inputs. The first one has a major focus on data locality, while the other two target a balance between data locality and load balancing. We have performed simulations on logs of an actual cluster. Our results show that LEM significantly improves the mean waiting time of a job, especially when the cluster is under a high computing demand. Without backfilling, LEM is better than our baseline in 75% of the cases (50% of the cases with backfilling). This work opens several exciting future directions. Firstly, LEM can be tuned to better adapt to the utilization rate of the cluster. Switching between a locality first and distribution first strategy could be done gradually. Secondly, we saw that LEO was very resilient but did not re-use enough the files to provide significant improvements. A direction is to improve locality for LEO. Lastly we could improve the pairing of backfilling and file re-use. In the long run, our objective is to consider others issues raised by batch scheduling: improving fairness between users, or dealing with advance reservations.

REFERENCES

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[2] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, vol. 2, 2005, pp. 776–783 Vol. 2.

[3] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 8–es. [Online]. Available: https://doi.org/10.1145/1188455.1188464

[4] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, "Workload management with loadleveler," *IBM Redbooks*, vol. 2, no. 2, p. 58, 2001.

[5] R. L. Henderson, "Job scheduling under the portable batch system," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 279–294.

[6] W. Gentzsch, "Sun grid engine: towards creating a compute power grid," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 35–36.

[7] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software: practice and Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.

[8] Y. Etsion and D. Tsafrir, "A short survey of commercial cluster batch schedulers," *School of Computer Science and Engineering, The Hebrew University of Jerusalem*, vol. 44221, pp. 2005–13, 2005.

[9] "Slurm workload manager," https://slurm.schedmd.com/sched_config.html, accessed: 2022-12-06.

[10] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the maui scheduler," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–102.

[11] S. Leonenkov and S. Zhumatiy, "Introducing new backfill-based scheduler for slurm resource manager," *Procedia Computer Science*, vol. 66, pp. 661–669, 2015, 4th International Young Scientist Conference on Computational Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915034249

[12] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," 02 2002, pp. 514 – 519.

[13] S. Leonenkov and S. Zhumatiy, "Introducing new backfill-based scheduler for slurm resource manager," *Procedia Computer Science*, vol. 66, pp. 661–669, 2015, 4th International Young Scientist Conference on Computational Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915034249

[14] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the maui scheduler," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 87–102.

[15] D. G. Feitelson and M. A. Jettee, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 238–261.

[16] S. Parsa and R. Entezari-Maleki, "Rasa: a new grid task scheduling algorithm," *JDCTA*, vol. 3, pp. 91–99, 01 2009.

[17] A. G. Delavar, M. Javanmard, M. B. Shabestari, and M. K. Talebi, "Rsdc (reliable scheduling distributed in cloud computing)," *International Journal of Computer Science, Engineering and Applications*, vol. 2, no. 3, p. 1, 2012.

[18] S. Ghanbari and M. Othman, "A priority based job scheduling algorithm in cloud computing," *Procedia Engineering*, vol. 50, no. 0, pp. 778–785, 2012.

[19] D. M. Dakshayini and D. H. Guruprasad, "An optimal model for priority based service scheduling policy for cloud computing environment," *International journal of computer applications*, vol. 32, no. 9, pp. 23–29, 2011.

[20] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit control in the batch-aware distributed file system." in *NSDI*, vol. 4, 2004, pp. 365–378.

[21] D. Borthakur *et al.*, "Hdfs architecture guide," *Hadoop apache project*, vol. 53, no. 1-13, p. 2, 2008.

[22] J.-F. Weets, M. K. Kakhani, and A. Kumar, "Limitations and challenges of hdfs and mapreduce," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, 2015, pp. 545–549.

[23] P. Mishra, T. Agrawal, and P. Malakar, "Communication-aware job scheduling using slurm," in *49th International Conference on Parallel Processing-ICPP: Workshops*, 2020, pp. 1–10.

[24] D. S. Nikolopoulos and C. D. Polychronopoulos, "Adaptive scheduling under memory constraints on non-dedicated computationalfarms," *Future Gener. Comput. Syst.*, vol. 19, pp. 505–519, 2003.

[25] P. Agrawal, D. Kifer, and C. Olston, "Scheduling shared scans of large data files," *Proc. VLDB Endow.*, vol. 1, no. 1, p. 958–969, aug 2008. [Online]. Available: https://doi.org/10.14778/1453856.1453960

[26] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 119–128.

[27] S. Selvarani and G. S. Sadhasivam, "Improved cost-based algorithm for task scheduling in cloud computing," in *2010 IEEE International Conference on Computational Intelligence and Computing Research*. IEEE, 2010, pp. 1–5.