Data-Driven Locality-Aware Batch Scheduling

No Author Given

No Institute Given

Abstract. Clusters employ 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 end up being 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 benefit from previously-loaded files to *decrease 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 input files already loaded in the memory of some node as much as possible.
We perform simulations using traces of real cluster usage, 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 improves performance compared to a widely-used scheduler (FCFS): a reduction in job waiting time (a 7.5% improvement in stretch), and a decrease in the amount of data transfers (7%).

1 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 like taxonomic identification of DNA fragments or ancestral reconstructions. It is a frequent use pattern for users of such communities to submit large batchs of jobs using the same input files. 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:

1. We formalize our model of scheduling data-intensive jobs sharing input files on a cluster (Section 3).
2. We propose three new schedulers focusing on re-using input files while minimizing evictions and avoiding starvation (Section 4).
3. We extract job information from historical logs of a cluster to build workloads that correspond to the needs and behaviors of real users (Section 5.2).
4. We implement all three heuristics as well as two state-of-the-art schedulers on a simulator and study the performance (mean stretch, amount of time spent waiting for a file to be loaded and number of core hours) obtained after scheduling almost 2 millions jobs (Section 5). Our evaluation demonstrates that our heuristics in most cases surpass the state-of-the-art schedulers. We show that workloads that heavily saturate the cluster benefit much more from our strategies, which results in considerable reduction in the job waiting times.

We used simulations to avoid disrupting users on a production cluster, but our strategies could be implemented on the Slurm workload by asking the users to flag their input files.

2 Related work

2.1 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 [22], OAR [5], TORQUE [20], LoadLeveler [14], Portable Batch System [11] or SunGrid [9] all offer various scheduling strategies.

The First-Come-First-Served (FCFS) algorithm is the prevalent default scheduler on most of these solutions [7]. Moreover, Slurm is used on most of the

TOP500 supercomputers and its default strategy is FCFS [1] as well. Backfilling strategies are known to improve the use of supercomputer resources [12]. The most commonly-used backfilling strategy, called conservative backfilling [19] [15], 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 [13], Gang scheduling [8], RASA [18], that use the advantages of both Min-min and Max-min algorithms, RSDC [6] that divides large jobs in sub-jobs for a more refined scheduling, or PJSC [10] that is a priority-based scheduler; however these heuristics do not consider the impact that input re-use could have on data-intensive workloads. We aim at resolving this issue in this paper.

2.2 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. BDFS [3] is designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters. HDFS [4] (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 use 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 (notably MapReduce, used in HDFS), as described by Weets et al. [21]. By not using a 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.

2.3 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 [16]. 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. [17] 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. [2] 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. That work is aimed at the Map-Reduce model and allows to predict future jobs arrivals, two prerequisites that we do not consider.

3 Framework

We consider the problem of scheduling a set of $\mathcal{J}$ independent jobs, denoted $\mathbb{J} = \{J_1, J_2, \ldots, J_{\mathcal{J}}\}$ on a set of $\mathcal{N}$ nodes: $\mathbb{N} = \{\text{Node}_1, \text{Node}_2, \ldots, \text{Node}_{\mathcal{N}}\}$. 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 J_i 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, $File(J_i)$ must be in the memory of Node_k. If this is not the case before starting computation of job J_i, then $File(J_i)$ is loaded into the memory. We denote by $\mathbb{F} = \{F_1, F_2, \ldots, F_{\mathcal{F}}\}$ the set of distinct input files, whose size is denoted by $Size(F_i)$. Each job runs on a single node, but they can make use of a different number 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: $SubDate(J_i)$;
- Requested running time: $WallTime(J_i)$: 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).

Each of the $\mathcal{J}$ jobs must be processed on one of the $\mathcal{N}$ 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.

Data-intensive jobs generally have very scarce outputs compared to the large inputs they use (e.g., genome databases). Therefore, we do not consider the output of jobs. We assume that jobs are devoted a fraction of the memory of a node 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 either of 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 between 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 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 $t_k + \mathit{WallTime}(J_i)$ devoted to (i) possibly loading the input file $\mathit{File}(J_i)$ in the memory (if it is not already present at time t_k) and (ii) executing job $J(i)$. If the job is not completed at time $t_k + \mathit{WallTime}(J_i)$, 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 throughout 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:

$$\mathit{Flow}(J_i) = \mathit{CompletionDate}(J_i) - \mathit{SubDate}(J_i).$$

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:

$$\mathit{ReferenceFlow}(J_i) = \frac{\mathit{Size}(\mathit{File}(J_i))}{\mathit{Bandwidth}} + \mathit{Duration}(J_i)$$
$$\mathit{stretch}(J_i) = \frac{\mathit{Flow}(J_i)}{\mathit{ReferenceFlow}(J_i)}.$$

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.

4 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 include 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.

4.1 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)

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 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 $t_k + WallTime(J_i)$

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 only 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 t_k 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, thus $t'_k = t_k + \frac{Size(File(J_i))}{Bandwidth}$;
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 been completed at time t_k. Then the file will be available at time: $t'_k = StartTime(J_j) + \frac{Size(File(J_i))}{Bandwidth}$.

Algorithm 2 Earliest-Finish-Time (EFT)

```
for each J_i ∈ the jobs queue do
    for each Node_k ∈ ℕ do
        Find smallest time t_k such that Cores(J_i) cores are available on Node_k
        Find time t'_k ≥ 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)
```

4.2 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. 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 increase the response

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 $\mathfrak{F}$ be the set of files in the memory of node Node_k at time t_k
    $EvictionPenalty \leftarrow (\sum_{F_j \in \mathfrak{F}} Size(F_j) \times Size(File(J_i))/M)/Bandwidth$
    $score_{\text{Node}_k} \leftarrow t_k + W \times LoadOverhead + EvictionPenalty$
  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
    if $t_k = current\_time$ **then**
      $score_{\text{Node}_k} \leftarrow t'_k$
    else
      $LoadOverhead \leftarrow t'_k - t_k$
      Let $\mathfrak{F}$ be the set of files in the memory of node Node_k at time t_k
      $EvictionPenalty \leftarrow (\sum_{F_j \in \mathfrak{F}} Size(F_j) \times Size(File(J_i))/M)/Bandwidth$
      $score_{\text{Node}_k} \leftarrow t_k + W \times LoadOverhead + EvictionPenalty$
  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 $t_k + WallTime(J_i)$

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) that takes a similar approach to LEO but performs a simple mix between the EFT and the LEA strategies: when the platform is saturated (each node is running at least one job), the LEA strategy is applied, otherwise the EFT strategy is used.

4.3 Adding backfilling to all strategies

As mentioned above, backfilling has been proposed to increase the performance of production cluster 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 that is 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

5 Performance evaluation and analysis

Actual workloads on HPC resources shared by a great number of users with diverse needs can contain projects that 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. On our cluster, we have observed such behavior. 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.

5.1 Platform description

We use a university cluster shared between several research laboratories. A variety of applications including genomics analysis are performed on this cluster. Our cluster contains 9720 cores spread over 486 nodes. Each node has two 10-core Intel Xeon V4 CPU at 2.20 GHz/core. Most of the nodes have 128 GB of RAM, and a few are larger. To avoid an additional constraint, while maintaining a model close to the real cluster, we consider that the platform is made of 486 homogeneous nodes of size $M = 128\,GB$.

The platform is managed by the SLURM scheduler, and as mentioned above, we observed the submission of I/O-intensive jobs. The cluster is often used for taxonomic identification of DNA fragments. To perform such identification, input databases[1] are often used, whose size vary from a few GB up to 800 GB. We also

[1] See `https://benlangmead.github.io/aws-indexes/k2`

noticed the use of genome alignment tools that require to store large text files, and it is likely that they need to load multiple of them, resulting in high I/O demands. From this platform we extract historical workloads containing the following information.

5.2 Usage of real cluster logs

The logs we use contain historical data on jobs, namely their exact submission time, their requested walltime, their actual duration, the number of cores they required and the corresponding user's name. 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. Each job uses exactly one input file. 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 request the same number of cores;
2. J_i and J_j are submitted by the same user;
3. J_i and J_j are 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. In theory, two users could share the same file. However, because they are using subsets of different databases, it is very unlikely that two users would work on the same project using the same databases, therefore we ignore this possibility.

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 estimate the size of files as follows:

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

Indeed, if a user needs 128 GB of memory but would request only 1 of the 20 available cores, that user would block the node for all other users, for lack of remaining memory. Consequently, a user who needs the whole memory of a node will reserve all its cores.

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 jobs 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 our experiments we evaluate our schedulers week by week. Our workload is constituted of 51 weeks, ranging from January the 3rd 2022 to December the

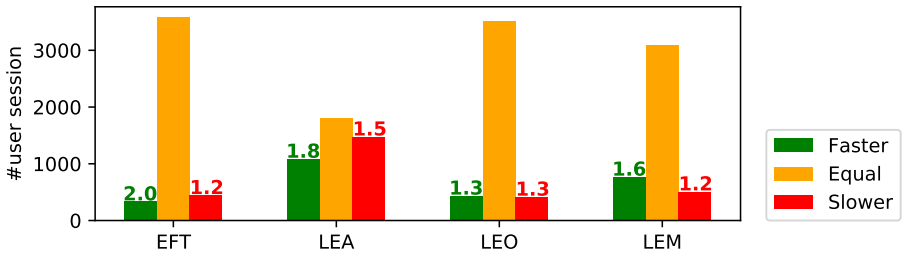

Fig. 1: User session stretch compared to FCFS at week 40.

25th 2022. To target different scenarios but avoid scheduling the whole year, we randomly selected 12 weeks and extracted the jobs submitted within these weeks from the logs. Over these 12 weeks of workloads, we scheduled 1 986 496 jobs. The most important for a user is that all the jobs that have been submitted at once are finished as soon as possible. Thus we introduce the notion of *user session*. We define the stretch of a *user session* as the sum of the stretch of jobs submitted by a user in a 5-minute window, and we concentrate on this value when reporting results. There have been 1083 distinct users and 136 404 *user sessions*.

To simulate these jobs in a realistic, steady-state operation of the platform, we consider both jobs submitted before and after the week under consideration.

Our evaluation is based on three metrics: the stretch of each *user session*, the total time spent waiting for a file to become available, as well as the total core time. We evaluate our proposed schedulers on these metrics and compare them to FCFS and EFT, with and without backfilling.

5.3 Simulator description

All strategies as well as the two baselines have been implemented on a simulator that we developed[2]. The simulator is an event-based simulator made in Python and focusing on data locality. As in real systems, the schedule is re-computed entirely at each unexpected event (job submission or job termination before the walltime). The scheduler is aware only of jobs that have been submitted before the current time and does not know the real duration of a job.

5.4 Results on an underutilized cluster

Among the 12 randomly-selected weeks, different workload conditions arise. We first concentrate on a low utilization case. Figure 1 shows the number of *user session* stretches that were faster, slower, or equal (+/- 1%) compared to FCFS. The number above the bars is the average speed-up (or slow-down) of

[2] Code, anonymized logs and the methodology followed to randomly select our evaluated weeks are available for reproducibility: `https://github.com/userdoubleblind/Locality-aware-batch-scheduling`

the faster (or slower) *user sessions*. We observe that our schedulers mostly do not bring any improvement for most of the *user sessions*, except for LEA, which degrades the performance for more than a third of its *user sessions*.

With a workload that does not fully utilize the cluster, the job queue is empty most of the time. Consequently, FCFS and EFT are very efficient. The earliest available node is in most cases a node that can start the job immediately, explaining why most *user sessions* get no improvement in Figure 1. LEO is 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 100% more than half the time, so LEM behaves similarly to EFT. On the contrary, LEA favors data re-use over an early start time for a job.

Schedulers	EFT	**LEA**	LEO	LEM
Reduction from FCFS	0.0%	**14.8%**	0.8%	3.8%

Table 1: Percentage of reduction in data transfer time relative to FCFS at week 40.

Table 1 shows the reduction of the amount of time spent waiting for a file to be ready before starting the computation, relative to the total waiting time of FCFS. In this case, LEA is the only strategy that decreases this waiting time by re-using input files. It will create a queue of jobs that already have a valid copy of their file loaded on a node, waiting to be able to re-use their input. On an underutilized cluster, this creates situations where some nodes are idle while submitted jobs wait in a queue, resulting in increased queue times. 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.

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

5.5 Results on a saturated cluster

We concentrate here on week 43, which we identify as a workload saturating the cluster: there are queues of several thousands requested cores for the whole duration of the evaluated week.

On a saturated cluster, filling all cores with the first jobs in the queue, as FCFS does, is not critical. It is more beneficial to group jobs that share the

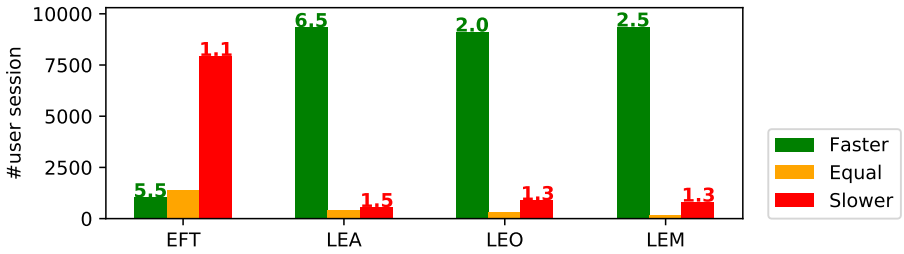

Fig. 2: User session stretch compared to FCFS at week 43.

same file. This way, the first few jobs will have a longer queue time than with FCFS, but over time, file reuse creates a snowball effect that reduces the queue time of all subsequent jobs. Moreover, the queue contains enough jobs to fill all the nodes, even when grouped by input file. We thus avoid the pathological case outlined in Section 5.4 for LEA. LEA's strategy, also found in LEO and LEM, allows to group jobs that share an input file, thus greatly reducing the stretch of each *user session*. LEA focuses only on locality, while LEO and LEM are mixed strategy, which explains the higher average improvement for LEA compared to LEO and LEM in this very locality-favoring situation.

5.6 Complete results

We report here the aggregated results of the 5 schedulers over the 12 weeks evaluated, after scheduling nearly 2 million jobs. Figure 3 and 4 depict the distribution of the improvement of each of our *user session* stretch, compared to FCFS. In other words we represent $Stretch\_FCFS/Stretch\_Scheduler$ for each *user session* and each scheduler. The horizontal black dotted line corresponds to no improvements from FCFS, which is the situation where the sum of the queue time and transfer time of the *user session* is the same as with FCFS. The solid lines are the median and the triangles are the average improvements. Because outliers get a huge improvement, the averages are usually higher than the medians. Because of such extreme outliers, we are more interested in median values. An improvement above 1 is a speed-up. An improvement at 0.5 means that the sum of the queue times and transfer times for this *user session* was two times larger with the scheduler than with FCFS. For all boxplots shown in this paper, the box contains results within the [25%,75%] range, while whiskers are drawn at 12.5% and 87.5%.

Results without backfilling We observe on Figure 3 that EFT does not bring any real improvement compared to FCFS. EFT takes into account file transfers when scheduling jobs but is a less aggressive strategy than LEA or LEM, and the savings in data transfers are only 0.9% (see Table 2). EFT is not able to see that a large number of jobs using the same file should be scheduled on the

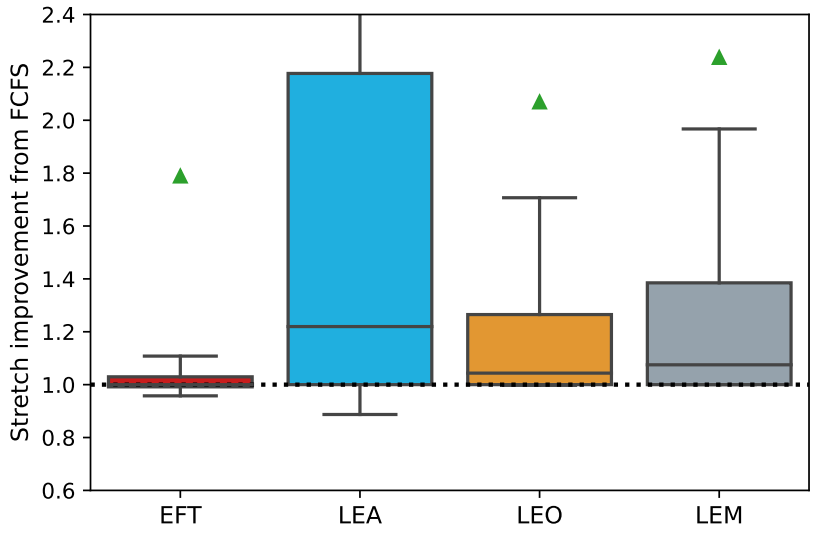

Fig. 3: Stretch's improvement from FCFS on all evaluated weeks. The upper whisker of LEA extends up to 5.

same node: even if it would generate more queue time, the overall execution time would be lowered thanks to data re-use.

Schedulers	EFT	**LEA**	LEO	LEM
Reduction from FCFS	0.9%	**17.1**%	0.9%	7.1%

Table 2: Percentage of reduction in data transfer time relative to FCFS on all workloads.

On the contrary, LEA has the largest median improvement. We can explain the larger median value for LEA from the good performance on heavily-saturated clusters (see Section 5.5). 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 (see Table 2) in the time spent waiting for a file greatly reduces the stretch of each job.

LEM has a lower median. However, as can be seen on Figure 3, at least 87.5% of its results are above 1, i.e., an improvement, whereas for LEA, only approximately 75% 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.

Schedulers	EFT	**LEA**	LEO	LEM
Reduction from FCFS	0.01%	**0.37**%	0.02%	0.10%

Table 3: Percentage of reduction of the total core time used on all workloads relative to FCFS.

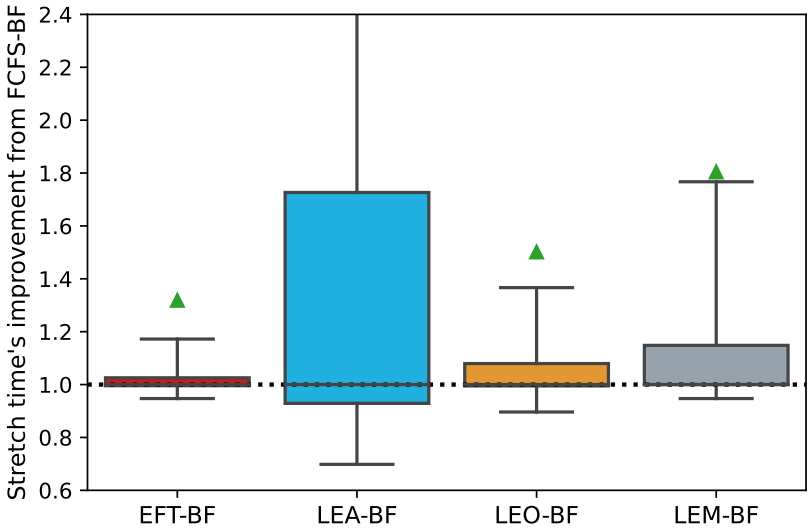

Fig. 4: Stretch's improvement from FCFS with backfilling on all evaluated weeks. The upper whisker of LEA extends up to 4.

Table 3 shows the reduction in terms of total core time (i.e. the core time on each core of each node over the 12 weeks). LEA reduces the total core time used by 49 000 core.hours over the 12 weeks (approximately 5h of the whole platform). Although it is a small percentage reduction, it can be of interest on large scale clusters that are highly demanding in terms of electrical resources.

Thus, LEA is the strategy that leads to more significant improvements: it is able to compute jobs between 1 and 2 times faster in 50% of the cases and between 2 and 6 times faster in 25%. 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. LEM is more versatile. It leads to an improvement in 87.5% of the results, with a speed-up of at least 7.5% in 50% of the results.

Results with backfilling Figure 4 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,

because FCFS-BF already performs much better than FCFS. The reason is that our proposed strategies do not benefit from backfilling as much as FCFS-BF does for two reasons.

Firstly, even if LEA, LEO and LEM 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.

Secondly, 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, FCFS-BF and EFT-BF already naturally benefit from increased data locality thanks to backfilling, leading to a reduced benefit in using LEA-BF, LEO-BF or LEM-BF.

Compared to FCFS-BF, our strategies still reduce the total queue time with backfilling but the difference is less significant.

Figure 4 shows that the improvement of EFT-BF compared to FCFS-BF is not significant. Out of our four heuristics, LEM-BF is the best compromise here. It is better than FCFS-BF in more than 75% of the cases, with 12.5% of those results above an improvement of 1.8. Among the slow-downs, only 12.5% are worse than 0.95.

6 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 of 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 LEA significantly improves the mean waiting time of a job, especially when the cluster is under a high computing demand. Without backfilling, LEA is better than our baseline in 75% of the cases (50% of the cases with backfilling). Our strategy called LEM is the best compromise. LEM is better than the baseline in more than 75% of the cases with or without backfilling (with a median improvement of 7.5% compared to our baseline without backfilling).

This work opens several exciting future directions. 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. An extension towards workflow scheduling where tasks depend on the output of a previous task would

be very useful. Indeed the output file in memory would be re-used for other jobs, which is a new opportunity for locality-aware schedulers. In the long run, our objective is to consider other issues raised by batch scheduling like fairness between users, or dealing with advance reservations. Lastly, we want to integrate these strategies into real cluster schedulers to test their robustness in real-world situations.

References

1. Slurm workload manager. https://slurm.schedmd.com/sched_config.html, accessed: 2022-12-06
2. Agrawal, P., Kifer, D., Olston, C.: Scheduling shared scans of large data files. Proc. VLDB Endow. (aug 2008). https://doi.org/10.14778/1453856.1453960
3. Bent, J., Thain, D., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Livny, M.: Explicit control in the batch-aware distributed file system. In: NSDI. vol. 4 (2004)
4. Borthakur, D., et al.: Hdfs architecture guide. Hadoop apache project (2008)
5. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounie, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005. vol. 2, pp. 776–783 Vol. 2 (2005). https://doi.org/10.1109/CCGRID.2005.1558641
6. Delavar, A.G., Javanmard, M., Shabestari, M.B., Talebi, M.K.: Rsdc (reliable scheduling distributed in cloud computing). International Journal of Computer Science, Engineering and Applications (2012)
7. Etsion, Y., Tsafrir, D.: A short survey of commercial cluster batch schedulers. School of Computer Science and Engineering, The Hebrew University of Jerusalem **44221** (2005)
8. Feitelson, D.G., Jettee, M.A.: Improved utilization and responsiveness with gang scheduling. In: Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
9. Gentzsch, W.: Sun grid engine: towards creating a compute power grid. In: Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid (2001). https://doi.org/10.1109/CCGRID.2001.923173
10. Ghanbari, S., Othman, M.: A priority based job scheduling algorithm in cloud computing. Procedia Engineering **50**(0) (2012)
11. Henderson, R.L.: Job scheduling under the portable batch system. In: Workshop on Job Scheduling Strategies for Parallel Processing. Springer (1995)
12. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Job Scheduling Strategies for Parallel Processing. pp. 87–102. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
13. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
14. Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.F.: Workload management with loadleveler. IBM Redbooks (2001)
15. Leonenkov, S., Zhumatiy, S.: Introducing new backfill-based scheduler for slurm resource manager. Procedia Computer Science (2015). https://doi.org/https://doi.org/10.1016/j.procs.2015.11.075, 4th International Young Scientist Conference on Computational Science

16. Mishra, P., Agrawal, T., Malakar, P.: Communication-aware job scheduling using slurm. In: 49th International Conference on Parallel Processing-ICPP: Workshops (2020)
17. Nikolopoulos, D.S., Polychronopoulos, C.D.: Adaptive scheduling under memory constraints on non-dedicated computationalfarms. Future Gener. Comput. Syst. **19** (2003)
18. Parsa, S., Entezari-Maleki, R.: Rasa: a new grid task scheduling algorithm. JDCTA **3** (01 2009). https://doi.org/10.4156/jdcta.vol3.issue4.10
19. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling (02 2002). https://doi.org/10.1109/ICPPW.2002.1039773
20. Staples, G.: Torque resource manager. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. SC '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1188455.1188464
21. Weets, J.F., Kakhani, M.K., Kumar, A.: Limitations and challenges of hdfs and mapreduce. In: 2015 International Conference on Green Computing and Internet of Things (ICGCIoT) (2015). https://doi.org/10.1109/ICGCIoT.2015.7380524
22. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)