Proceedings Paper:
Markovic, Ana, Kolovos, Dimitris orcid.org/0000-0002-1724-6563 and Soares Indrusiak, Leandro orcid.org/0000-0002-9938-2920 (2023) Distributed Data Locality-Aware Job Allocation. In: Proceedings of 2023 SC Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC Workshops 2023. 2023 International Conference on High Performance Computing, Network, Storage, and Analysis, SC Workshops 2023, 12-17 Nov 2023 ACM International Conference Proceeding Series . Association for Computing Machinery, Inc , USA , pp. 2089-2096.

https://doi.org/10.1145/3624062.3624287

Distributed Data Locality-Aware Job Allocation

Ana Markovic
am2813@york.ac.uk
University of York
York, United Kingdom

Dimitris Kolovos
dimitris.kolovos@york.ac.uk
University of York
York, United Kingdom

Leandro Soares Indrusiak
leandro.indrusiak@york.ac.uk
University of York
York, United Kingdom

ABSTRACT

Scheduling tasks close to their associated data is crucial in distributed systems to minimize network traffic and latency. Some Big Data frameworks like Apache Spark employ locality functions and job allocation algorithms to minimize network traffic and execution times. However, these frameworks rely on centralized mechanisms, where the master node determines data locality by allocating tasks to available workers with minimal data transfer time, ignoring variances in worker configurations and availability. To address these limitations, we propose a decentralized approach to locality-driven scheduling that grants workers autonomy in the job allocation process while factoring in workers' configurations, such as network and CPU speed differences. Our approach is developed and evaluated on Crossflow, a distributed stream processing platform with data-aware independent worker nodes. Preliminary evaluation experiments indicate that our approach can yield up to 3.57x faster execution times when compared to the baseline centralized approach where the master controls data locality.

CCS CONCEPTS

• **Computing methodologies** → *Distributed algorithms*; • **Software and its engineering** → *Development frameworks and environments.*

KEYWORDS

Big Data processing, Data-aware job scheduling, Distributed processing, Locality scheduler.

1 INTRODUCTION

Data locality, also termed spatial locality, refers to the data processing occurring at the location of the data storage [6]. Increasing data locality when scheduling tasks in a distributed environment can lead to lower execution times, mainly because network bandwidth is a scarce resource compared to CPU speed, and transferring data to the nodes that need it can often last longer than waiting for a node which has the data in close proximity to become available. This means that in distributed and data-intensive environments, such as MapReduce [7], we could witness a speed improvement by avoiding the equal distribution of the work among all nodes and introducing schedulers that will allocate tasks based on the placement of data.

The objective of this paper is to propose a novel approach to data-aware scheduling that will allow distributed worker nodes a degree of independence and responsibility in the task allocation process. Our approach is implemented on top of Crossflow [12], which is a distributed stream-processing engine. The next section explains the motivation behind locality-aware scheduling, whereas in Section 3 we analyze different techniques proposed in the literature for placing the work close to the data. Section 4 explains how the framework we chose as the baseline works with regard to job allocation, which is followed by a detailed description of our algorithm in Section 5. The set of key metrics we use for assessment is outlined in Section 6, together with the discussion about the results obtained from multiple workflow runs. Lastly, we provide a summary of the achievements and limitations of our implementation and we present a plan for future work in Section 7.

2 MOTIVATING EXAMPLE

In this section we outline an application scenario that motivates the need for processing data with a high degree of locality. Our motivating use case revolves around software repository mining (MSR), which is the analysis of source code repositories and other development artefacts to extract insights of interest. More specifically, we consider a pipeline which queries GitHub for Git repositories that have source code dependent on particular libraries and we investigate how often these libraries are used together. More precisely, we consider how often popular NPM libraries [1] for JavaScript co-occur in favoured large-scale projects on GitHub (e.g. repositories with over 500MB in size with at least 5000 stars and forks), by looking for *package.json* files present in the repository and inspecting their dependencies, in case there are any. To achieve this, we need to define a simple protocol to measure the co-occurrences of various libraries:

(1) Capture the libraries to look for in a structured format (e.g. JSON, CSV)
(2) Search GitHub for favoured large-scale repositories (e.g. repositories larger than 500MB with at least 5000 stars and forks)
(3) Clone repositories found in the previous step, query them for *package.json* files and look into their dependencies
(4) Calculate the number of times libraries appear together and store the results in a CSV file

Figure 1 showcases the described pipeline. The rounded boxes (e.g. Repository and RepositorySearchResult) present different types of jobs in the pipeline, where each job is defined as a piece of data required to process a task. A task example would be RepositorySearcher, which is shown in a rectangular shape. The cylinders are used as communication channels, allowing for different types of jobs to be input or output for connecting tasks.

From an implementation perspective, to deal with the size of repositories to be cloned (which could be in the order of GBs) and minimise the number of expensive repository cloning operations, it would be helpful to ensure that cloned repositories could be saved for later use without the need to re-download their contents. To reuse the downloaded resources, repeated computations involving the same files would be required to be allocated to the same worker

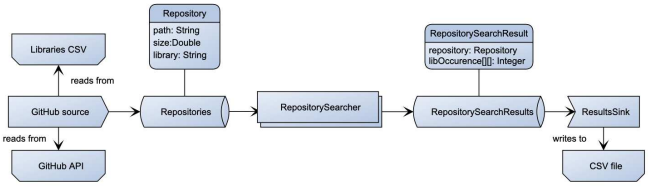


Figure 1: MSR pipeline specified in Crossflow [12]

nodes, namely the ones that already possess them. By reducing the download costs, especially for large resources such as GitHub repositories, we could see a significant increase in the speed of workflow execution.

To illustrate the rationale behind leveraging data locality in this specific scenario, consider an incoming stream of libraries l_i to be searched and a workflow consisting of two workers w_1 and w_2. The large-scale projects dataset obtained from GitHub's API includes repositories r_1 and r_2. Let us assume the presence of a library denoted as l_1, resulting in two related jobs for the RepositorySearcher task: specifically (l_1, r_1) and (l_1, r_2).

Examining the job (l_1, r_1) reveals that one of the workers needs to clone the repository r_1 and examine it for the presence of the library l_1, such as worker w_1. Subsequently, after an arbitrary amount of time, a library l_2 may emerge, giving rise to a job such as (l_2, r_1). In order to minimize the download costs by eliminating the time required for repository cloning, it would be advantageous to pair this job with the worker node that has the repository contents already saved locally. In this instance, that worker node would be w_1, which possesses resources saved from processing a previous job.

The following section aims to outline different approaches for locality-aware scheduling that were published over time, and to identify a comparative baseline for the work we propose in this paper.

3 RELATED WORK

In the current stage of distributed Big Data processing, several research attempts use data resources to improve task allocation. Some approaches attempt to delay job assignment until an appropriate node is available [13, 14]. If that node is unavailable, the allocation will be postponed, which can occur a fixed number of times. A downside to delayed scheduling is that workers do not always become free as quickly as expected. Therefore, if the system is overloaded, maintaining a high data locality could lead to wasting time by waiting. The Matchmaking [9] technique for MapReduce [8] presents a similar mechanism but avoids wasting time by allowing nodes to request jobs rather than receive them. Only when a node becomes available will it try to pull a task for which it has data locally. The node will remain idle for a single heartbeat if no such task is present. On the second attempt, it is bound to accept a task even if it does not have data locally.

More complex algorithms have also been proposed to improve systems' performance by increasing data awareness. For example, in BAR [11], the authors introduce a function that calculates completion time with respect to data locality. Their algorithm comprises two phases: at first, they attempt to assign all the tasks so they are entirely local, only to iteratively produce alternative execution scenarios which reduce completion time on account of the locality. A more recent research effort proposed a novel scheduler for cloud-based systems that utilises machine learning to decide when scheduling tasks away from their data is less expensive than moving data to achieve the maximum locality (e.g. when the network bandwidth is unstable) [10].

When it comes to Apache Spark [5], it introduces five levels of locality, and it attempts to schedule tasks so that the maximum degree of locality is obtained. If that is not possible, it will wait a threshold period of time before reducing the level of locality for that particular task [2]. Each data piece can be viewed as local based on an IP or the set of IP addresses presenting servers containing it, and this information can be retrieved from Spark's partition [3].

To summarise, data awareness aims to reduce the end-to-end execution time by avoiding unnecessary data transfers and is achieved through compromising the fairness of task allocation. This section described some of the techniques in the literature, whereas the next one will introduce Crossflow [12], another framework that tackles data locality, which will be used for implementing and evaluating the new data-aware scheduler.

4 CROSSFLOW

Crossflow [12] is a data processing engine which, like Apache Spark [5] and Flink [4], relies on the master/worker paradigm. However, it is designed specifically to cope with resource-intensive workflows, such as the one discussed in Section 2. It is aimed at processing streams of data and tailored to deal with potentially expensive but not as numerous jobs compared to streams of events suitable for Spark and Flink (e.g. tweets or web server log events). For instance, in the example of GitHub mining, searching a repository for files of specific types can be viewed as a single job and considered expensive due to the need to download contents of the repository in order to identify co-occurrences of technologies of interest.

While adhering to the master/worker architecture, Crossflow distinguishes itself from other technologies in terms of its component functions. Notably, Crossflow employs "opinionated" nodes, a unique feature that enables workers to decide on whether to accept or decline a job based on their preferences, without direction from the master node. This means that applications built on top of this framework can be implemented so they schedule work based on data locality, since workers could be instructed to store used resources locally and decide to work on tasks they already possess the data for. In our example of repository mining, this could be one way of mitigating the issue of redundant downloads, i.e. the allocation of jobs could be dependent on the contents of workers' local filesystems, that is, whether they have already processed the same GitHub repository and saved it.

In more detail, instead of the master pushing jobs to the workers it finds appropriate, Crossflow currently deals with scheduling by enabling worker nodes to pull jobs from the master. Before being executed, each pulled job is internally evaluated by the worker to check if it conforms to that worker's *acceptance criteria*. If it does, the job is processed, otherwise, it is returned to the master so another worker can consider it. The intelligence of workers lies in this evaluation process, where they proceed to accept an assignment

or decline it based on their internal state, i.e. their opinions. In order to ensure the completion of all incoming jobs, workers are required to keep track of any jobs they have previously declined. This enables them to accept such jobs upon a second attempt, provided that no other worker possesses the necessary capabilities to carry out the task.

The concrete *acceptance criteria* are application-specific and left to the developer to define. For example, it could be related to data locality if the workers are instructed to scan the contents of local cache memory. However, it could also be any other logical expression, such as available CPU/RAM capacity or an attribute-based preference. Hence, for the mining software repositories example, exploiting data locality is limited to the functioning of worker nodes that trigger and control necessary data transfer through a method the programmer implemented to suit this specific use case.

Crossflow demonstrates promising performance in executing pipelines in which transferring data to workers is expensive, compared to state-of-the-art frameworks such as Apache Spark. For instance, in the MSR example, Crossflow achieves up to 8.2x faster processing due to differences in scheduling techniques. Unlike Spark, where all task allocation occurs in advance and without considering the resources that become local during execution, Crossflow performs impromptu task allocation as jobs arrive. Additionally, Crossflow's "opinionated" nodes enable worker preferences to be considered when allocating tasks, which is not the case with Spark, where the master produces all assignments and considers all workers equal. This can potentially result in slower workers having to download and process larger repositories or complete assignments later than other workers, leading to longer execution times. Figure 2 presents the end-to-end execution time comparison of the MSR example between Crossflow and Apache Spark. The chart is organized into different column groups, which showcase various worker and job configurations used to execute the workflow. In the first group, where one worker's internet and read/write speeds are significantly faster, and one worker is slower than the others, while repositories to analyze are large (e.g., larger than 500MB), Spark takes 7.94x longer to complete the workflow than Crossflow. Similarly, in the second column group, where all workers have equivalent characteristics, and small repositories are processed (e.g., smaller than 50MB), Crossflow is 2.3x faster than Spark. The third and fourth column groups demonstrate the workflow's runtime when all workers have the same characteristics computing on non-repetitive dataset, and computing on varying network and read/write speeds while processing a repetitive dataset (80% of jobs in column group four required the same repository). The test case scenarios used to compare the two frameworks are explained in more detail in Section 6.3.1.

Although Crossflow's scheduler is highly local, it is likely there will be redundant clones of the same repository if a node is offered a job it has previously seen, even though some other node has that resource locally but is currently occupied. If this happens, even though the distributed allocation happening in Crossflow can incorporate workers' configurations, it is not guaranteed that the redundant clone will be assigned to a fast worker who can facilitate efficient execution. Even so, the preliminary evaluation indicates that Crossflow's scheduling technique can aid the performance of

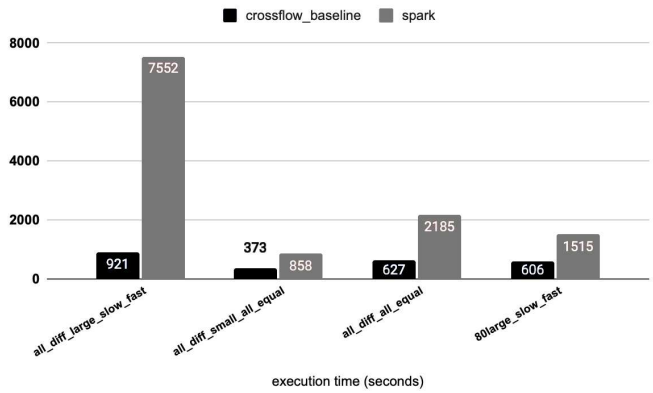


Figure 2: Execution times of MSR in Spark compared to Crossflow Baseline

long-running resource-intensive workflows which establishes a starting point for investigating distributed job allocation further.

That being said, we can identify a couple of constraints of the described decentralized scheduling mechanism to be considered in our research:

- when executing the pipeline for the first time, all worker nodes will end up rejecting repository-related jobs as they do not possess any clones locally,
- there is no assurance that performant workers will end up with compute-intensive jobs and vice versa, potentially increasing execution times.

Altogether, this creates a baseline for developing a new approach to job allocation in Big Data frameworks that will exploit opinionated nodes and attempt to overcome existing disadvantages depicted above.

5 BIDDING SCHEDULER: THE DISTRIBUTED LOCALITY-AWARE JOB SCHEDULER

The goal of this section is to present a new scheduling mechanism that both takes data locality into account and relies on opinionated nodes. It is proposed as an improvement to the Crossflow framework discussed in Chapter 4, however, it presents a general solution that could be integrated with other data processing engines.

In this approach, which we will refer to as the Bidding Scheduler in the remainder of the paper, worker nodes are not responsible for accepting/rejecting jobs, but they enhance the traditional master/worker architecture by participating in the job allocation process and making scheduling a distributed decision-making activity. The master node still broadcasts incoming jobs, however, in this algorithm, the workers create offers and bid for work. Their bids include estimates on when they *estimate* they can get that job done, i.e. how much time they think they need to complete a particular task. Since worker nodes schedule tasks in FIFO order, this estimate must include the time to obtain any necessary resources and execute the new job, as well as the time to download resources and execute all unfinished jobs that have been previously allocated (i.e. bids won previously). The master waits for workers to make

submissions within one second and looks into all the received bids before allocating the job to the worker who made a submission with the lowest estimate. The same applies to the situation when not all the workers sent their bids on time: the master chooses the bid with the lowest estimate of all received bids or assigns the job to an arbitrary node in case none of the workers submitted their estimates before the threshold period exceeded. Therefore, although the communication process is asynchronous, we rely on time frames to group the messages and ensure that the execution does not break in case workers follow different timings from the master. However, in the initial concept of the Bidding Scheduler, we did not address the issue of fault tolerance. As a result, there are currently no specific policies in place to handle situations such as a worker dying after winning a bid or redistributing the remaining jobs if a worker becomes unavailable. Moreover, as we envision the bidding process to be handled by a separate thread, we do not anticipate workers being late to make a submission. The main focus of the approach is that the bids the master receives provide it with insight regarding both data locality and previously committed workload and allow it to distribute work in a manner that will reduce the total execution time and data transfer costs.

Listings 1 and 2 contain pseudo-codes for the Bidding technique from the master's and the worker's perspective. Lines 2-4 in Listing 1 encompass the initial steps responsible for initiating a bidding contest when a new job arrives: the master publishes it for bidding in line 3 and sets its status to *open* in line 4, meaning that the bidding contest for that job is ongoing. The *receiveBid* function, defined in lines 7-15, is invoked upon receiving a bid. Specifically, line 7 verifies which job the bid corresponds to, while lines 8 and 9 store the bid information in the master record. Line 10 evaluates whether the bidding contest should be concluded, and if this condition is met, it triggers a sequence of operations: changing the bidding status for the job (line 11), identifying the winner (line 12), and eventually assigning the job to the winning worker (line 13). The *getPrefferedWorker* function determines the worker with the lowest estimate by sorting the received bids (line 19) and returning the winner (line 20). Concerning the job allocation process, lines 24 and 25 update the job status and record the start time, subsequently instructing the worker to queue the job for processing (line 26). Finally, lines 29-31 outline the existing decision-making logic for closing contests, verifying whether all workers have submitted their bids or the bidding contest has been open for over a threshold period of time (in this case, 1 second).

The worker functionalities in this algorithm are delineated into two main functions. The first function, as depicted in lines 1 to 8, is responsible for estimating the workload and submitting a bid. In line 2, the worker estimates the time required to complete the current workload by aggregating the costs of previously won queued jobs. This estimation serves as the baseline for creating a new bid, as stated in line 3. The algorithm exhibits data awareness in line 4, wherein each worker is mandated to calculate the cost of acquiring the required data. The data transfer time could be determined by dividing the size of the repository by the current network speed, however the concrete formula is application specific and should be left to the developer to define. Minimum expenses are incurred when the worker possesses the data stored locally, which leads to lower time estimates and subsequently increases the chances of winning the bid. The bid amount is increased in line 5, according to the current read/write speed's job processing time estimation. Similarly to the data transfer time, the processing time in the MSR example could be computed by dividing the repository size by the current read/write speed, yet this is specific to different types of jobs and should be defined at the application level. Finally, the bid is transmitted to the master, as specified in line 6.

The algorithm's second function, described in lines 9 to 15, comes into play when the worker wins the bidding and is required to process the job. In line 10, the worker initiates the job processing by updating its status to *started*. The actual work is executed in line 11. The worker updates the current job status in lines 12 and 13 and submits the result as the new job to be processed downstream in line 14 (or to be recorded as a result in case the performing task is the final one in the pipeline).

Upon closer examination of the bidding algorithm, it becomes clear that no job needs to be rejected by all workers before being processed. This is because, unlike the original *acceptance criteria*, the "opinionated" nodes offer job completion estimates that factor in details, such as workers' network and read/write speeds in this example, when allocating work. Moreover, this approach to scheduling where the tasks are allocated as they arrive also allows for volatile environments, as workers' performance metrics can fluctuate over time and still be leveraged to reduce execution time and data load. Additionally, because the estimates are based on worker configurations, the Bidding Scheduler ensures that redundant resources (i.e. copies of the same repository cloned in different workers in this example) occur only to accelerate overall execution. This is the case if, for example, the worker that has resources locally has too many queued jobs, therefore the cost of waiting for it to become available might be greater than the cost of assigning the job to another worker that needs to download the resources for itself. The next section will compare the two distributed scheduling approaches previously portrayed and discuss the results obtained from experiments.

6 EVALUATION SETUP, METRICS AND RESULTS

To examine the differences between the schedulers, we configured the Crossflow framework to support both versions of job allocation, the Bidding Scheduler and its Baseline scheduler where workers can reject incoming jobs once in case they do not possess the necessary data locally. The remainder of this section aims to present metrics we used to evaluate our approach and discuss the results we obtained from conducting the experiments.

6.1 Metrics

Listed below is a set of metrics we defined for comparing various job allocation techniques with respect to data locality. To help assess the aspects the new scheduler and data locality function are expected to improve, this set could comprise the following list of attributes to be quantitatively measured:

(1) **End-to-end execution time.** The amount of time required to execute a workflow, from starting the master and worker nodes to terminating their processes.

Listing 1 Running on master

```
1: function sendJob(job)
2:     publishForBidding(job);
3:     Bids[job.id].status = open;
4: end function
5:
6: function receiveBid(bid)
7:     job_id = bid.job_id;
8:     bids_for_job = bidsMap[job_id];
9:     bids_for_job.add(bid);
10:    if biddingFinished(job_id) then
11:        Bids[job_id].status = closed;
12:        w = getPreferredWorker(job);
13:        sendToWorker(w, job);
14:    end if
15: end function
16:
17: function getPreferredWorker(job)
18:     receivedBids = bidsMap[job_id];
19:     receivedBids.sort(bid as bid.cost_in_sec, ASC);
20:     return receivedBids[0].workerID;
21: end function
22:
23: function sendToWorker(job, worker)
24:     JobStatus[workerID, jobID].status = queued;
25:     JobStatus[workerID, jobID].timestamp_started = now;
26:     worker.consumeJob(job);
27: end function
28:
29: function biddingFinished(job_id)
30:     return bids[job_id].length = activeWorkers.length OR bidding_lasted_for > 1s;
31: end function
```

Listing 2 Running on worker

```
1: function sendBid(job)
2:     currentWorkloadCost = totalCostOfUnfinishedJobs();
3:     bid = currentWorkloadCost;
4:     bid += estimateDataTransferTime(job);
5:     bid += estimateProcessingTime(job);
6:     bidForJob(job, bid);
7: end function
8:
9: function consumeJob(job)
10:    JobStatus[this.workerID, job.jobID].status = started;
11:    newJob = process(job);
12:    JobStatus[this.workerID, job.jobID].status = finished;
13:    JobStatus[this.workerID, job.jobID].timestamp_finished = now;
14:    master.sendJob(newJob);
15: end function
```

(2) **Data load.** The volume (in megabytes) of data that is not local and must be transferred to the worker nodes during execution.
(3) **Cache miss.** The number of times workers did not have the necessary data locally and were required to either download it or relocate it in order to perform computations.

The novel scheduling with locality-driven workers' decisions should decrease the end-to-end execution and data load for pipelines using this scheduling mechanism for allocating work. Therefore, these metrics will be considered the most important when assessing various job allocation techniques.

6.2 Experimental Setup

To evaluate the performance of the two scheduling mechanisms, we conducted experiments on a distributed AWS infrastructure using t3.micro instances. We ran the MSR workflow with a total of five workers and the master node. The infrastructure consisted of a total of 7 instances, with one assigned to each worker, one for the master, and one for the messaging infrastructure. The instances were geographically distributed, and their locations were randomly determined during configuration startup.

6.3 Simulation results

6.3.1 Configuration. Initially, we defined a set of different workers and jobs configuration to be used for testing our approach in a controlled environment. The evaluation encompassed multiple configurations for both the workers and the incoming jobs, allowing us to observe the behaviour of the algorithms across different setups. Namely, for workers we prepared four different configurations, all comprising five workers in total:

- **All-equal.** A configuration where all workers have the same, or nearly the same, network and read/write speeds as well as storage resources.
- **One-fast.** A configuration where one worker is significantly faster than the others, in terms of network and computation speed.
- **One-slow.** A configuration where one worker is significantly slower than the others, in terms of network and computation speed.
- **Fast-slow.** A configuration which has one slow and one fast worker, while the remaining three have average download and processing speeds.

Similarly, we created five different job configurations with 120 jobs each, to emulate the real-world assignment patterns. In these configurations, repositories can vary in sizes (be small, medium or large, ranging between 1MB and 1GB), and the jobs can be all different or repetitive (depending on whether the jobs requiring the same repository occur again in the same pipeline run). Details of the configurations are listed below:

- **All_diff_equal.** Equal distribution of repository sizes, with all jobs in the test case scenario using different repositories.
- **All_diff_large.** Mostly large repositories, with all jobs in the test case scenario using different repositories.
- **All_diff_small.** Mostly small repositories, with all jobs in the test case scenario using different repositories.
- **80%_large.** Repetitive pattern with mostly large repositories. Within the set of large-scale jobs, 80% require the same large repository.
- **80%_small.** Repetitive pattern with mostly small repositories. Within the set of small-scale jobs, 80% require the same repository.

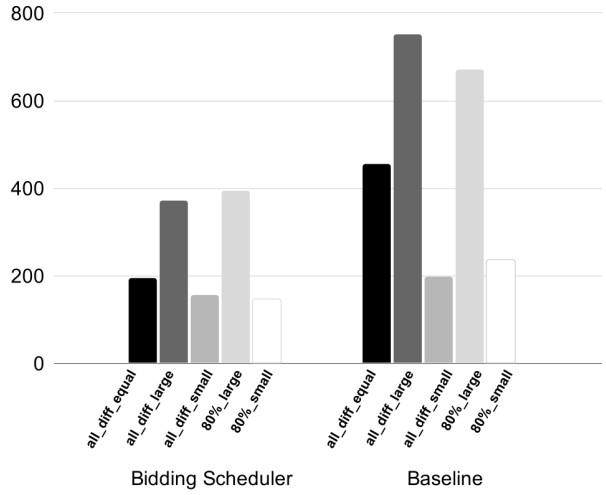
For testing purposes, we ran all combinations of worker and job configurations, in three iterations each. Multiple iterations are required to examine the data locality features, since, in the case of all jobs being different, we cannot see job allocation occurring with respect to data storage unless workers have files saved from previous executions. To control network and read/write speeds, workers were equipped with preconfigured speeds upon initiating the workflow. These speeds were used to determine bid values. However, to better replicate real-world network throttling scenarios and ensure bidding costs differed from actual execution times, the speeds were subjected to a noise scheme during job execution to simulate realistic variations in network conditions. The next section outlines core findings stemming from evaluating the behaviour of the two approaches in a simulated environment.

6.3.2 Experiment results. Performing tests with the configurations depicted above led to three crucial conclusions we aim to elaborate further in the remainder of this section:

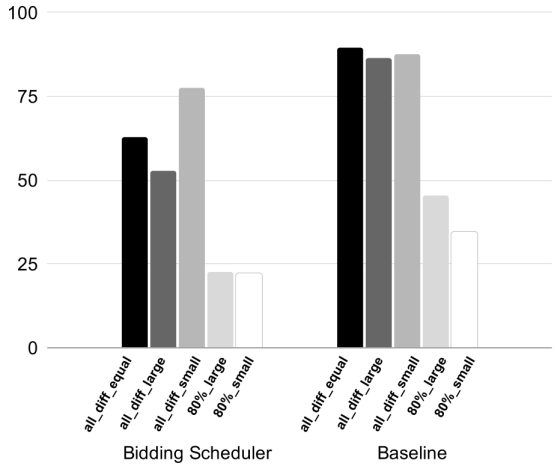
(1) Bidding Scheduler achieves a speedup of approximately 24.5% compared to the Baseline.
(2) Bidding Scheduler demonstrates improvements in local data utilisation, with approximately 49% fewer cache misses and approximately 45.3% reduction in data load per workflow run.
(3) The Bidding Scheduler exhibits an overhead that makes it more effective for large resources and long-running workflows. However, for small resources or short workflows, competing for jobs unnecessarily prolongs the execution, making it less advantageous compared to the Baseline.

Figure 3 displays three charts that provide additional insights into the performance differences between the Bidding Scheduler and the Baseline. The charts illustrate the average time required for execution, the average number of cache misses, and the average number of megabytes downloaded per algorithm per workload, respectively. These three factors are interrelated, as demonstrated by the results. For example, when 80% of repositories are large and repeating, the Bidding Scheduler records approximately 22.65 cache misses per workflow run, compared to the Baseline's 45.5. With reduced cache misses for the same test configuration, the Bidding Scheduler minimizes data load from 10786.88MB to 5270.87 MB on average, ultimately leading to a 51% reduction in data downloaded and a 41% increase in speed. Similarly, observing aggregated data for the test scenario of all_diff_equal we can conclude that, for the case of all repositories being different with their sizes equally distributed, the Bidding Scheduler downloads 9591.45 MB compared to Baseline's 17908.08 MB, resulting from 26.83 less cache misses on average, which altogether leads to approximately 57% speedup.

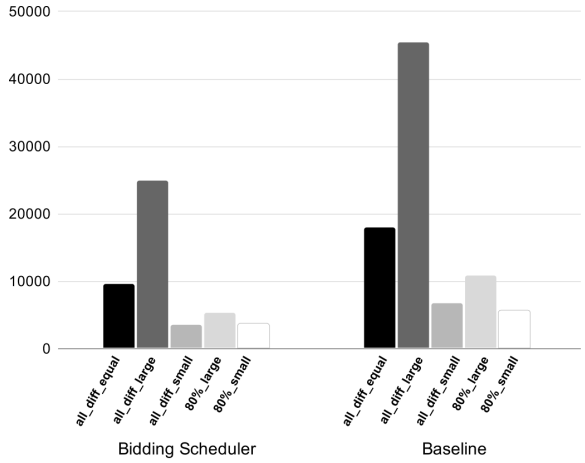
Figure 4 features a breakdown of the total workflow execution time, analyzing the measured number of seconds taken to complete the workflow with diverse job configurations and worker characteristics. Observing the cases for one_slow and one_fast for both the Bidding Scheduler and the Baseline, it can be argued that the figure offers evidence that the Bidding Scheduler is tailored to address only a specific subset of use cases, rather than being suitable for all. In our setup, the performance of the scheduler is impacted by two key factors: internet connectivity and resource size. In situations where high-performing machine instances exist and should compute over relatively small pieces of data, the cost of transferring data remains lower than the overhead involved in contesting an incoming job. As such, the Bidding Scheduler performs comparably to, or somewhat slower than, the Baseline when one worker is significantly more efficient than the others. Conversely, Bidding outperforms the Baseline when workers have restricted internet access or need to work with large resources. The advantages of Bidding can be attributed to the increased autonomy of workers and the utilisation of various worker characteristics in the estimation calculations facilitated by the Bidding Scheduler, which allows workers to inform the master about their current state with each submission. In turn, this enables the master to prioritize workers based on their capabilities, avoiding the prolongation of execution due to slower nodes carrying excessive workloads. Additionally,



(a) Average total execution time per workload



(b) Average cache miss count per workload



(c) Average data load per workload

Figure 3: Accumulated results per workload per algoritm

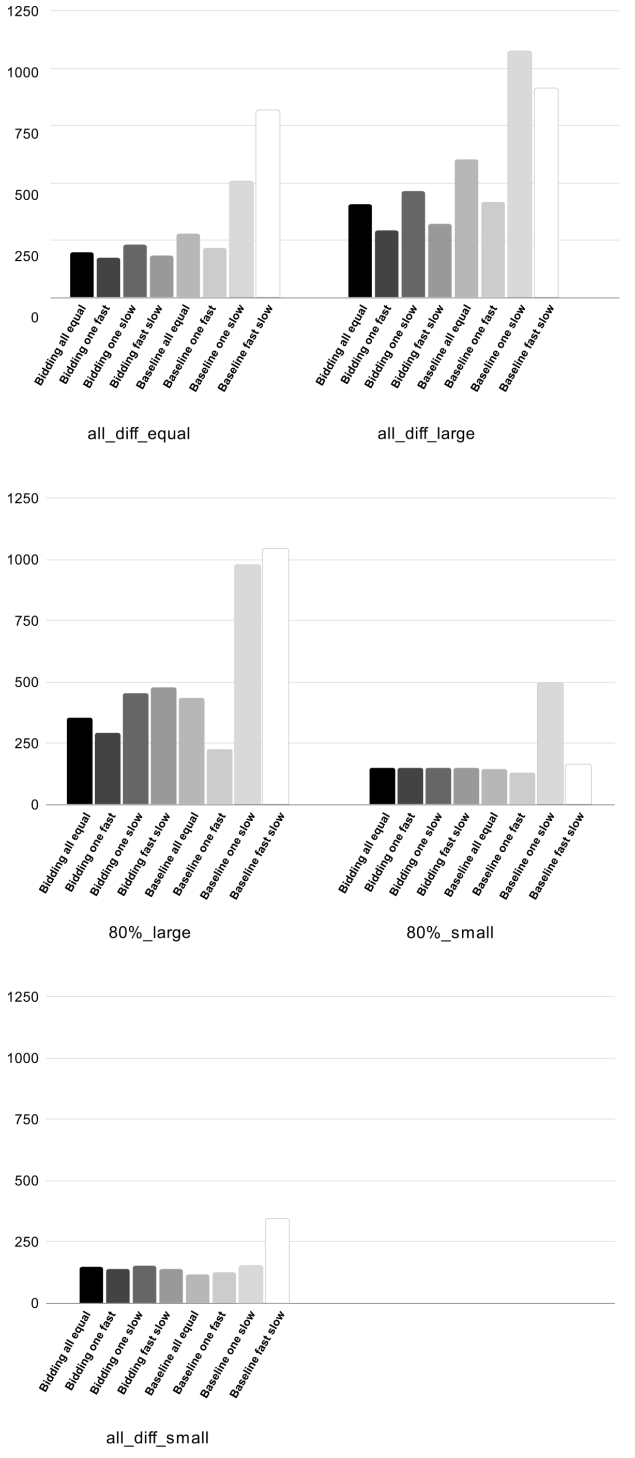


Figure 4: Average execution times per workload per algorithm

MSR	Bidding	Baseline
run 1	3204.5s	3575.55s
run 2	2918.5s	3544.45s
run 3	3116.52s	4183.5s

Table 1: MSR execution times

MSR	Bidding	Baseline
run 1	332935.90 MB	891165.59 MB
run 2	325461.08 MB	847802.57 MB
run 3	330048.70 MB	889594.77 MB

Table 2: Data load in MB

MSR	Bidding	Baseline
run 1	205	405
run 2	191	394
run 3	186	386

Table 3: Cache miss count

since scheduling occurs for every job as it arrives, the master can adapt its allocation decisions to fluctuations in workers' speeds over time.

6.4 Non-simulated experiment results

To evaluate the authenticity of our emulation, we conducted a series of experiments that executed the MSR example on the identical AWS infrastructure. These experiments relied on the bandwidth and read/write speeds available to the AWS instances, as well as the responsiveness of the GitHub API. We performed the pipeline with both schedulers three times each, to collect results and observe nuances between the executions.

In this particular evaluation scenario, workers were assigned pre-defined network and read/write speeds. These speeds were obtained by examining a repository of 100MB in advance. Upon completion of each job, workers were tasked with calculating their latest network and read/write speeds. The network speed was determined by dividing the size of the repository by the time taken to complete the download, while the read/write speed was computed by dividing the repository size by the time taken to examine its contents. These newly obtained speeds were then utilized to set the network and read/write speeds for the subsequent bid, by calculating the historic average for all speeds determined for previous jobs.

The results obtained from this set of experiments, organized according to the metrics described in Section 6.1, are presented in Tables 1, 2, and 3. In Table 1, we observe that, when none of the workers have any locally downloaded repositories, the Bidding Scheduler completes the execution with a 10.3%-25.5% reduction in time compared to the Baseline. For example, for *run*3 this difference is at its highest, given that the setup with the Bidding Scheduler finishes in 3116.52s whereas the one relying on the Baseline completes in 4183.5s. In contrast, the lowest acceleration was recorded for

*run*1 where the Bidding Scheduler completed in 3204.5s compared to the Baseline's 3575.55s.

To further analyze these findings, we should examine the data presented in Tables 2 and 3. In *run*1, the pipeline utilizing the Bidding Scheduler downloaded 332935.90 MB, while the same workflow employing the Baseline Scheduler transferred 891165.59 MB. This corresponds to 205 and 405 cache misses in Table 3, respectively, indicating a significant reduction of 62.6% in data load attributable to a reduction of 49.4% in cache misses. In *run*3, as shown in Table 3, the Bidding Scheduler resulted in 200 fewer repository clones, leading to a decrease in downloads of 559546.07 MB, as indicated in Table 2. Consequently, the Baseline Scheduler exhibited a 25.5% longer execution time due to higher data transfer rates by approximately 62.9% compared to the Bidding Scheduler. These findings collectively demonstrate the successful enhancement of data locality achieved by the Bidding Scheduler.

7 CONCLUSIONS AND FUTURE WORK

This paper presented a novel technique for scheduling work in Big Data frameworks, that relies on distributed decision-making, unlike the traditional master/worker architectures. This mechanism delegates more responsibility to the worker nodes through participating in the bidding contest to acquire jobs. Moreover, to increase their chances of getting an assignment, they are responsible for maintaining their cache memories and local resources and seeking work corresponding to the data they have locally. Preliminary evaluation shows promising results with regard to the bidding mechanism, that in most cases performs better than the baseline due to the fact that its scheduling is not strictly controlled by the data locality and takes into account workers' characteristics. Beyond larger-scale evaluation and comparing the approach to other locality scheduling techniques such as Matchmaking, future work includes minimizing the bidding overhead for highly local jobs as well as providing more intelligence for the worker nodes by enabling them to keep the historic data of their bids and completed work and use this data to learn from it and adjust their future bids.

REFERENCES

[1] [n. d.]. 30 Most Popular NPM Packages for Node JS Developers. https://www.turing.com/blog/top-npm-packages-for-node-js-developers/ Last visited: 08.08.2023..

[2] [n. d.]. *Spark Data Locality*. https://spark.apache.org/docs/latest/tuning.html#data-locality Last visited: 15.08.2023..

[3] [n. d.]. *Spark Data Locality - Code Reference*. https://github.com/apache/spark/blob/aba9492d25e285d00033c408e9bfdd543ee12f72/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L137 Last visited: 17.06.2023..

[4] [n.d.]. *Apache Flink*. https://flink.apache.org Last visited: 07.07.2023..

[5] [n.d.]. *Apache Spark*. https://spark.apache.org Last visited: 07.07.2023..

[6] Wo Chang and Nancy Grady. 2019. NIST Big Data Interoperability Framework: Volume 1, Definitions. https://doi.org/10.6028/NIST.SP.1500-1r2 Last visited: 30.01.2023..

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[9] Chen He, Ying Lu, and David Swanson. 2011. Matchmaking: A New MapReduce Scheduling Technique. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. 40–47. https://doi.org/10.1109/CloudCom.2011.16

[10] Ru Jia, Yun Yang, John Grundy, Jacky Keung, and Hao Li. 2019. A Highly Efficient Data Locality Aware Task Scheduler for Cloud-Based Systems. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 496–498. https://doi.org/10.1109/CLOUD.2019.00089

[11] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. 2011. BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 295–304. https://doi.org/10.1109/CCGrid.2011.55

[12] Dimitris S. Kolovos, Patrick Neubauer, Konstantinos Barmpis, Nicholas Matragkas, and Richard F. Paige. 2019. Crossflow: a framework for distributed mining of software repositories. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 155–159.

[13] Ioan Raicu, Ian Foster, Yong Zhao, Philip Little, Christopher Moretti, Amitabh Chaudhary, and Douglas Thain. 2009. The quest for scalable support of data-intensive workloads in distributed systems. *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, 207–216. https://doi.org/10.1145/1551609.1551642

[14] Matei Zaharia, Dhruba Borthakur, Joydeep Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. *EuroSys'10 - Proceedings of the EuroSys 2010 Conference*, 265–278. https://doi.org/10.1145/1755913.1755940