Job-aware Scheduling in Eagle: Divide and Stick to Your Probes

Pamela Delgado, Diego Didona, Florin Dinu and Willy Zwaenepoel
EPFL
first.last@epfl.ch

Abstract

We present Eagle, a new hybrid data center scheduler for data-parallel programs. Eagle dynamically divides the nodes of the data center in partitions for the execution of long and short jobs, thereby avoiding head-of-line blocking. Furthermore, it provides job awareness and avoids stragglers by a new technique, called Sticky Batch Probing (SBP).

The dynamic partitioning of the data center nodes is accomplished by a technique called Succinct State Sharing (SSS), in which the distributed schedulers are informed of the locations where long jobs are executing. SSS is particularly easy to implement with a hybrid scheduler, in which the centralized scheduler places long jobs.

With SBP, when a distributed scheduler places a probe for a job on a node, the probe stays there until all tasks of the job have been completed. When finishing the execution of a task corresponding to probe *P*, rather than executing a task corresponding to the next probe P' in its queue, the node may choose to execute another task corresponding to *P*. We use SBP in combination with a distributed approximation of Shortest Remaining Processing Time (SRPT) with starvation prevention.

We have implemented Eagle as a Spark plugin, and we have measured job completion times for a subset of the Google trace on a 100-node cluster for a variety of cluster loads. We provide simulation results for larger clusters, different traces, and for comparison with other scheduling disciplines. We show that Eagle outperforms other state-of-the-art scheduling solutions at most percentiles, and is more robust against mis-estimation of task duration.

Categories and Subject Descriptors 4.1 [*Process Management*]: Scheduling

Keywords Cloud computing, Data center, Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987563

1. Introduction

Data center (cluster) scheduling is a challenging problem for a variety of reasons. The first issue is the heterogeneity of the workload. A typical workload consists of long and short jobs. The long jobs tend to be latency-insensitive and, while small in number, they consume the bulk of the resources. Vice versa, there are many short jobs, they are latency sensitive, but consume only limited resources [5, 22]. Therefore, the scheduler has to take care to avoid head-of-line blocking, i.e., placing a short job behind a long one, especially under high load. The second issue is the parallel nature of the jobs: the overall completion time of a job is equal to that of its slowest task. Therefore, the scheduler has to be job-aware, considering all tasks of a job rather than individual tasks in isolation. The final issue stems from the scale of the data center. A very large number of jobs must be scheduled on a very large number of nodes. At this scale, centralized schedulers can exhibit high scheduling latency [25], and as a result distributed [20, 21] or hybrid centralized/distributed [5, 14] schedulers have been developed.

This paper introduces a new hybrid scheduler, called Eagle. Eagle divides the data center's nodes in partitions for the execution of short and long jobs to avoid head-of-line blocking, and introduces sticky batch probing to achieve better job-awareness. We describe these techniques next, motivating them by fundamental results from queueing theory.

The Pollaczek-Khinchine formula states, under rather general conditions, that the expected completion time of jobs served by a node is proportional to the variance of the job execution times [16]. This observation has led to so-called Size Interval Task Assignment (SITA) scheduling policies, that statically divide compute nodes into different partitions for executing jobs of different lengths [8]. SITA as such is not practical in a data center because variations over time in the resource demands of long and short jobs make a static division inefficient. Instead, we develop a distributed and dynamic variant of SITA, by providing the schedulers for short jobs with information about where long jobs are currently executing, through a technique we call Succinct State Sharing (SSS).

Furthermore, Little's law states, again under rather general conditions, that the expected completion time of a job is inversely proportional to the number of jobs present in the

system [18]. To optimize job completion times, one must therefore optimize the rate at which entire jobs leave the system, and avoid that straggler tasks delay job completion. Eagle introduces Sticky Batch Probing (SBP) to deal with this problem. In contrast to conventional probe-based schedulers [5, 20], in SBP a probe does not represent a single task, but rather the whole job. A probe can trigger the execution of as many tasks of a job as required to prevent stragglers. In combination with SBP, Eagle implements a variant of the Shortest Remaining Processing Time (SRPT) scheduling policy [8], which further improves job-awareness and job completion times.

We evaluate Eagle through simulation on data center traces from Cloudera, Facebook and Google, and through implementation and measurement on a cluster with 100 nodes. Eagle compares favorably to earlier data center schedulers. In particular, we demonstrate that it does better at avoiding head-of-line blocking than other probe-based schedulers that rely on work stealing [5]. Furthermore, we show that Eagle's distributed job-awareness provides better completion times than schedulers relying on local job-awareness [21]. Finally, we quantify Eagle's superior robustness against mis-estimations of job execution times.

The key contributions of this paper are:

1. A technique for dividing a data center, dynamically and in a distributed fashion, into partitions for executing long and short jobs, thereby reducing head-of-line blocking.
2. A technique for bringing job-awareness to distributed scheduling in a data center.
3. The implementation and evaluation, through simulation and implementation, of a hybrid data center scheduler, Eagle, that embodies these techniques.

The outline of the rest of this paper is as follows. Section 2 introduces the system and the workloads targeted by Eagle. Section 3 shows how Eagle avoids head-of-line blocking using SSS. Section 4 shows how SBP provides job-awareness in Eagle. Section 5 describes the evaluation methodology. Section 6 presents experimental results obtained by means of extensive trace-driven simulations. Section 7 presents experimental results obtained by deploying and running a prototype of Eagle. Section 8 discusses related work. Section 9 concludes the paper.

2. System Model

We consider a data center composed of worker nodes. A job consists of a set of tasks that can run in parallel on different workers. Scheduling a job requires assigning every task of a job to a worker. When a new task is scheduled on an idle worker, the task starts executing immediately. When there is already a task running on the worker, the new task is appended to the queue on the worker.

The completion time of a task is the time from the submission of the job that contains the task to the time when the task finishes execution. A job completes when all of its tasks finish. The job completion time is then the maximum of the task completion times of all its tasks. The scheduling time of a task is the time between the submission of the job of which that task is part until the time the task is queued at a worker. The queueing time of a task is the time the task spends in the queue (zero if the worker is idle when the task is assigned). The execution time of a task is the time the job spends running. The task completion time is the sum of the scheduling time, the queueing time and the execution time. The execution time of a job is the sum of the execution times of its tasks.

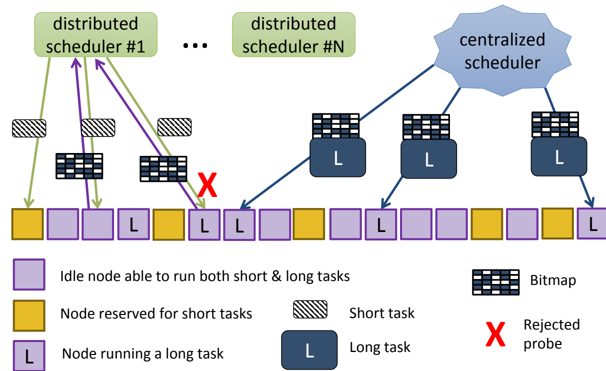


Figure 1: Overview of Eagle and Succinct State Sharing.

Consistent with observations made in many papers about data center workloads [5, 15, 22], we assume that the workload consists of a small number of long jobs that consume a large fraction of the data center's resources, and a large number of short jobs that consume only a small fraction of the data center's resources. We refer to long (short) tasks as the tasks of a long (short) job.

Similarly to other schedulers [1, 5, 21], Eagle leverages the availability of estimated tasks execution times for an incoming job. The estimated task execution time for a given job is computed as the average execution time across all the tasks in a given job. A job is classified as long (short) if the average execution time for its tasks falls above (below) a given threshold. The rationale underlying this approach for identifying long and short jobs is grounded in the fact that jobs in modern data centers are typically recurring [4, 6], and execute against similar input data. This allows Eagle to compute task runtime estimates looking at previous executions of the same job. Similarly, the relative proportion of short and long jobs is expected to remain stable over time. This enables the implementation of a simple yet accurate threshold-based classification of short vs long jobs.

3. Divide

3.1 Design

The problematic situation for short tasks in a data center highly loaded with long tasks is the so-called head-of-line blocking: a short task is enqueued behind a long task (either in the queue or running) and has to wait a long time to run. Since the majority of resources in typical data center workloads is taken up by long jobs, head-of-line blocking is one of the main causes of poor performance for short, latency-sensitive tasks [5].

Eagle provides a solution to the head-of-line blocking problem by means of a novel approach grounded in queueing theory. The Pollaczek-Khinchine formula states that the average completion time of the jobs executed by a node is proportional to the variance in the job execution times [16]. This result implies that, in a data center, reducing the variability of the execution times of jobs assigned to a single node yields a reduction in their average completion time.

SSS reduces the variability in the execution times of tasks assigned to nodes by enforcing that a short task is never enqueued behind a long one. SSS has a static and a dynamic component. The data center is statically split into two partitions. The smaller of the two, referred to as the *short* partition, is reserved for short jobs. The bigger one, referred to as the *general* partition, is primarily dedicated to long jobs, but may on occasion execute short jobs, guided by the dynamic component of SSS, as follows. SSS informs the short jobs schedulers in a low-overhead fashion about the placement of long jobs in the general partition, allowing them to opportunistically place short tasks on nodes in the general partition that are not currently serving a long job.

SSS achieves two principal goals: (*i*) it completely eradicates the head-of-line blocking problem, by avoiding short tasks to be enqueued behind long ones, and (*ii*) it achieves high resource utilization by dynamically allowing short jobs to run in the general partition.

3.2 Benefits Over Previous Designs

Head-of-line blocking is a primary concern for data-center schedulers, and a number of solutions have been proposed to address it.

The Hawk scheduler [5] has separate schedulers for short and long jobs and reserves a small portion of the data center for short jobs. Hawk, however, allows short tasks to be enqueued behind long ones, to prevent resource under-utilization. To compensate for the resulting head-of-line blocking, Hawk implements randomized work stealing. When a node is idle, it contacts some nodes at random to steal probes that are enqueued behind a long task. As we shall show in Section 6, work stealing, as implemented in Hawk, only partially removes head-of-line blocking.

Mercury [14] mitigates head-of-line blocking by means of load shedding. In more detail, tasks from overloaded nodes are periodically relocated to underloaded nodes. Similarly to work stealing, load shedding does not operate at the scheduling level, but it is a runtime correction mechanism.

Hawk and Mercury only partially avoid head-of-line blocking. Some tasks may execute on a node after having experienced head-of-line blocking. Even if they are relocated, they may have waited behind a long task. SSS, instead, pro-actively eradicates the head-of-line blocking problem by avoiding that a short task ever gets enqueued behind a long one.

The Pollaczek-Khinchine formula is also at the basis of the so called Size Interval Task Assignment (SITA) scheduling policies [8, 10]. According to such policies, each node i in a data center serves only jobs whose estimated execution time falls in a specific range $[S_i, S_i']$. A limitation of SITA policies, however, is their *static nature*. It is possible for a subset of the nodes to be temporarily (over)loaded, while having other nodes, assigned to other execution time ranges, idle or under-loaded [8–10]. SSS leverages the Pollaczek-Khinchine formula in a similar fashion to SITA policies. Unlike SITA policies, however, SSS preserves high resource utilization by dynamically and opportunistically allowing short tasks to occupy worker nodes in the general partition.

3.3 Implementation

Eagle is a hybrid scheduler, in which a centralized scheduler handles long jobs and distributed schedulers handle short jobs [5]. The rationale for this design is the following. Long jobs are relatively few, but consume the bulk of the resources, so their centralized scheduling allows for a good placement of the most demanding jobs, while not introducing a scalability bottleneck. The distributed scheduling of short jobs enables their placement on worker nodes with low latency and in a scalable fashion. Figure 1 shows the hybrid nature of Eagle and provides an overview of SSS.

The centralized scheduler implements the Least Work Left (LWL) scheduling policy [8] to place long jobs on nodes in the general partition, as in Hawk [5]. This policy places each task on the node corresponding to the smallest expected queueing time for such task. When the centralized scheduler assigns a (long) task to a worker node, it piggybacks on the message a timestamp and a succinct copy of its state, consisting of a bitvector of length equal to the number of workers, with bit i indicating whether or not worker i currently has a long task assigned to it (either running or enqueued). The worker node stores this bitvector, together with the timestamp received in the message. The arrival of a new long task causes the old bitvector and timestamp to be replaced with the newly received values.

The distributed schedulers are based on probing [19, 20]. For a job with t tasks, a distributed scheduler sends probes to $max\{K, 2t\}$ worker nodes, with K being a tunable parameter. A distributed scheduler sends a minimum of K probes to improve the completion times of short jobs with very few tasks. Otherwise, such jobs would result in a very small

number of probes being sent, reducing the likelihood that at least one probe lands on an unloaded node.

A distributed scheduler selects the targets of probes for a given job uniformly at random among all nodes. A probe can reach a node N to which a long task is currently assigned. In this case, N rejects the probe and responds to the distributed scheduler with its bitvector and corresponding timestamp.

The distributed scheduler then re-schedules the rejected probes. To do so, it uses the freshest available bitvector to identify the set of nodes to which currently no long jobs are assigned. For the re-scheduling phase, target nodes are drawn uniformly at random from the set of nodes that are not currently serving a long job according to the selected bitvector. Some probes might be rejected again, due to stale bitvectors or the concurrent arrival of long jobs. Probes rejected during re-scheduling are assigned uniformly at random to nodes in the short partition.

Probes are scheduled at first by contacting nodes uniformly at random, even if the distributed scheduler already has a bitvector available. We have experimentally verified that this design leads to better results than using an old bitvector in the first scheduling attempt, because sampling nodes at random gives a better approximation of the current utilization of the data center than a possibly stale bitvector.

4. Stick to Your Probes

4.1 Design

A key characteristic of data-parallel jobs is that a job completes when all its tasks finish. The overall completion time of a job is therefore equal to that of its slowest task.

Little's law, a fundamental result in queueing theory [18], states that, given an arrival rate of jobs to a system, the average job completion time is inversely proportional to the number of jobs in the system.

Applied to the data-parallel jobs case, Little's law indicates that a scheduler needs to optimize the completion time of jobs as a whole, and not the completion time of their individual tasks. In other words, a good scheduling policy must be job-aware.

Eagle uses LWL to schedule long jobs. Therefore, the centralized scheduler in Eagle places long tasks on nodes aiming to optimize the completion time of the whole job.

Eagle introduces SBP to provide job-awareness for the distributed scheduling of short jobs.

In SBP, a probe does not represent a single task of a job but a whole job. In other words, a single probe can lead to the execution of multiple tasks of the corresponding job.

When a task of a job J finishes at a given worker node, rather than relinquishing the node to the next task in the local queue, the worker may contact the distributed scheduler of J to request another task of J. In this way worker nodes are allowed to quickly remove all tasks of a job from the system once that job starts, thus avoiding stragglers.

SBP implements the latest possible form of task-to-node binding, by assigning a task to a node only when the node has available resources. If more nodes storing a probe for J have available resources, more tasks of J can be executed in parallel. Thus, SBP gracefully adapts the degree of parallelism of jobs execution to resources availability.

The following example shows how SBP augments probing with job-awareness. Suppose we have a data center with 4 nodes, with queue lengths of 100 at nodes $n1$, $n2$, and queue lengths of 10 at nodes $n3$ and $n4$. We have to schedule a job with four tasks, each with duration 10. One probe lands on each node, making the expected execution time of the overall job 110. With SBP, instead, at time 20 node $n3$ and $n4$ are able to pull from the distributed scheduler another task each, thus achieving a job completion time of 30.

SBP does not restrict worker nodes to process probes in FIFO order. It is well known that SRPT achieves optimal average completion time by executing the job with smallest remaining execution time first (in the single-task job scenario with preemption) [8, 17]. Aiming to further reduce short job completion times, Eagle implements an approximate variant of the SRPT scheduling policy on top of SBP. This variant does not need support for preemption, works for data-parallel jobs, and is augmented with an anti-starvation measure.

SBP can only implement the approximated variant of SRPT, as it would be implemented by a centralized scheduler, because of its probing-oriented nature. If J is the job to run next according to SRPT, J can only be executed on nodes which host a probe for it. Therefore, even if some other node has available resources, J cannot be run there.

4.2 Benefits Over Previous Designs

We now show the limitations of existing scheduling systems in implementing job-awareness.

The work stealing implemented by Hawk, being randomized, is not job-aware. With reference to the previous example, node $n4$, once idle, could in vain try to steal a probe from $n3$, and vice versa. In general, a stealing attempt can fail, or it can target a job with no straggler tasks. Mercury is job-unaware as well, and its load shedding technique rebalances queues based only on their backlogs and not on the job status of enqueued tasks.

Yaq [21] implements job-awareness by supporting different local queue reordering policies. Yaq performs early binding of tasks to nodes, thus re-introducing the issue of straggler tasks. We refer to the example used in 4.1, where nodes $n1$, $n2$, $n3$ and $n4$ have queue lengths of 100, 100, 10 and 10 respectively. Supposing that the queue length for $n1$ and $n2$ is mis-estimated to be 10 instead of 100, a scheduler in Yaq would place the four tasks of the example job, one in every node. With a task execution time of 10, this would result in an actual completion time of 110. In Eagle, assuming the initial scheduling is the same, nodes $n3$ and $n4$ will ex-

Algorithm 1 Sticky Batch Probing + SRPT

```
1:  procedure MAINLOOP
2:      while (true) do
3:          task ← GetNextTaskToExecute()
4:          ExecuteTask(task)
5:          Send(task.scheduler, finishedTask, task.id)
6:  procedure GETNEXTTASKTOEXECUTE
7:      p ← GetProbeFromQueue()
8:      if (p.isLong) then
9:          queue.pop(p)
10:         return p.task
11:     else
12:         reply ← Send(p.scheduler, getTask)
13:         return reply.task
14: procedure GETPROBEFROMQUEUE
15:     shortest ← infinite
16:     chosen ← void
17:     for p in queue do
18:         if (p.isLong) then
19:             break
20:         if (p.estJobLeftRuntime < shortest) then
21:             if (CanBypass(p.estTaskRuntime)) then
22:                 shortest ← p.estJobLeftRuntime
23:                 chosen ← p
24:     if (chosen == void ∧ queue.size() > 0) then
25:         return queue.first
26:     else
27:         return chosen
```

ecute the two straggler tasks enqueued at $n1$ and $n2$ at time 20, resulting in an actual completion time of 30 instead.

In general, Eagle's SBP is able to pull tasks to the "best" node on which a probe is located, while Yaq may, as a result of mis-estimation, locate a task on a less desirable node, without any possibility of recovering from that choice other than performing job-unaware load shedding.

4.3 Implementation

We only discuss the scheduling of short jobs, as they are the only jobs affected by SBP. To further simplify the presentation, we first explain an SBP implementation in connection with FIFO scheduling. We then augment that implementation with Eagle's variant of SRPT, and complete the description with the anti-starvation measure.

In the case of FIFO, when a probe P arrives at the head of the queue on node N, N contacts the distributed scheduler of P. The scheduler replies to N with one task T of the job J corresponding to P. N does not remove the selected probe from the queue. Once T terminates its execution, N requests another task of J, until all tasks of J are executed.

In the case of SRPT, N selects from its queue the probe to run according to SRPT instead of according to FIFO. In order to make this selection, N needs to know the remaining execution times for the jobs for which it has probes in its queue. To this end, when a distributed scheduler sends a probe P to N, it also communicates the number of tasks composing the job and their expected average execution time. Upon assigning a new task to a node, the distributed scheduler updates the number of remaining tasks at all nodes where it has located probes. This information suffices for N to decide which probe to select according to SRPT.

Because SRPT does not respect FIFO order, starvation can occur: a probe that has joined N's queue at time t can be bypassed indefinitely often by tasks whose probes have joined N's queue at a time $t' > t$. To prevent starvation, whenever a probe P is about to be bypassed by a task T, P's "bypass counter" is incremented by the estimated execution time of T.

Eagle does not allow T to bypass P if the estimated execution time of T would make the bypass counter of P exceed a threshold value.

In summary, when choosing the next short task to execute, N picks the probe corresponding to the enqueued job with the shortest remaining execution time that is allowed to bypass all the probes in front of it. If there is no such probe, then N selects the next probe in FIFO order.

The pseudocode in Algorithm 1 depicts this protocol, reporting the operations performed by a worker node. For simplicity, the initial scheduling of probes and the message exchanges to update the remaining execution time of the jobs are not shown.

The main loop ran by a worker node is embedded in the function `MainLoop`, which performs three main steps: (*i*) retrieval of the next task to run (Line 3), (*ii*) execution of the task (Line 4) and (*iii*) notification of task completion to the distributed scheduler (Line 5).

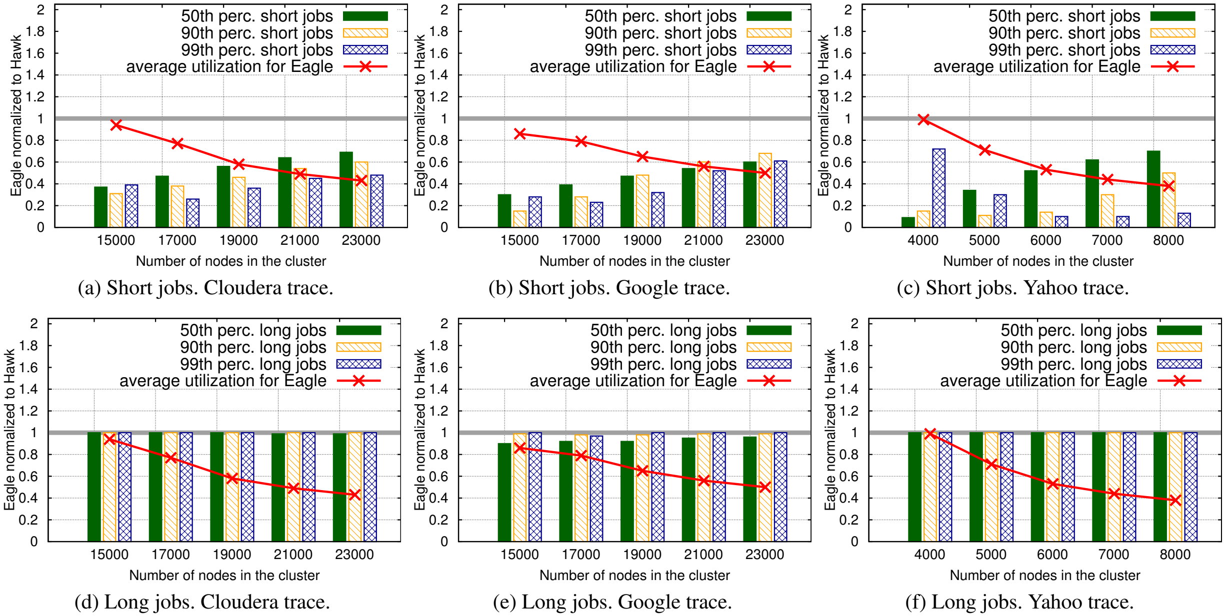
The function `GetNextTaskToExecute` implements SBP. It first invokes the `GetProbeFromQueue` function to determine whether the next task to execute belongs to a long or to a short job. In the former case, `GetNextTaskToExecute` removes the corresponding placeholder from the queue (Line 9). In the latter case, the function contacts the distributed scheduler corresponding to the returned probe to pull a new task (Line 12).

The `GetProbeFromQueue` function embeds the logic of Eagle's job-aware policy. If the worker is in the general partition and there are no probes, the function picks a long task from the queue in FIFO order. Otherwise, the function implements Eagle's variant of SRPT described before and returns the next short task to be executed.

5. Evaluation Methodology

We evaluate Eagle using the Google workload trace as the primary workload [22], and we additionally use traces from Cloudera [3] and Yahoo [2]. A detailed description of these traces can be found in [5]. For completeness, Table 1 shows for each of the three traces the total number of jobs, the percentage of long jobs and the percentage of task-seconds for long jobs.

We assess Eagle's effectiveness by means of a twofold methodology. We evaluate, against Hawk, a prototype integrated in Spark [27], on a 100-node deployment running a subset of the Google trace. In addition, we provide simulation results for larger clusters and for all the traces, comparing with other scheduling policies.



(a) Short jobs. Cloudera trace. (b) Short jobs. Google trace. (c) Short jobs. Yahoo trace.

(d) Long jobs. Cloudera trace. (e) Long jobs. Google trace. (f) Long jobs. Yahoo trace.

Figure 2: 50/90/99th percentiles of job completion times. Eagle normalized to Hawk.

Trace	Total # jobs	% Long jobs	% Task-Seconds long jobs
Cloudera [3]	21030	5.02	91
Yahoo [2]	24262	9.41	98
Google [22]	506460	10.00	83

Table 1: Job heterogeneity in the traces. *% Task-seconds long jobs* is the sum of the execution times of all long tasks divided by the sum of the execution times of all tasks.

We vary the number of worker nodes to simulate both high and low load conditions. Each worker node has one queue. The size of the small partition is 17, 9 and 2 percent of the nodes, for the Google, Cloudera and Yahoo traces, respectively. These numbers correspond to the percentage of the execution times (task-seconds) of all short jobs in the respective traces. The choice of these values is aimed at allocating an amount of worker nodes to short jobs that is proportional to their computational demands.

We set the network delay to 0.5 milliseconds, and we do not assign any cost to making scheduling decisions. To prevent starvation, as specified in Section 4.3, we set the starvation threshold value to 5 times the estimated duration of a single task in a job, for all experiments involving SRPT. In all the Eagle experiments, unless otherwise stated, the minimum number of probes sent per job is 20.

We use as main metrics the 50th, 90th and 99th percentiles of the job completion time distribution. The results are averages over 5 runs. We do not plot error bars, since the results of different simulations are consistent across runs.

6. Simulation Results

In Section 6.1 we evaluate the benefits of Eagle against Hawk, a state-of-the-art hybrid scheduler. Next, in Section 6.2, we compare Eagle against a Distributed Least Work Left (DLWL) scheduler with node-local Shortest Remaining Processing Time (SRPT) queue reordering. In Section 6.3 we show a breakdown of Eagle's components. A comparison against an omniscient centralized scheduler is shown in Section 6.4. Finally, Section 6.5 includes an analysis of Eagle's robustness to mis-estimations.

6.1 Comparing Eagle Against Hawk

We compare the job completion time distributions for short and long jobs between Hawk and Eagle for the three considered traces. Work stealing in Hawk is configured according to the default settings [5]: idle nodes in the general partition perform ten attempts to steal probes from other nodes in that partition. If successful, they steal the first batch of short tasks that are enqueued behind a long one.

Figure 2 (top) shows, as a function of the number of worker nodes, the 50th, 90th, and 99th percentiles of the job completion time distribution for short jobs for Eagle, normalized to those same values in Hawk. In addition, we report the average node utilization with the given number of nodes. Figure 2 (bottom) reports the same results for long jobs.

From Figure 2 we see that for short jobs Eagle achieves better job completion times than Hawk at all percentiles, for all load conditions and in all traces. The improvements brought about by Eagle are more evident at higher loads,

Trace	# Probes behind long task		# Short tasks exec after long		# Re-scheduled probes	
	Hawk	Eagle SSS	Hawk	Eagle SSS	Hawk (Steal)	Eagle SSS
Google 15K nodes	17.30M	0	0.63M	0	15.70M	16.98M
Yahoo 4K nodes	0.87M	0	18K	0	0.80M	0.86M
Cloudera 15K nodes	5.60M	0	55K	0	5.50M	5.65M

Table 2: Head-of-line blocking statistics: Eagle vs Hawk.

since the impact of the better resource utilization achieved by SSS and SBP is higher when resources are scarce. At the highest load, Eagle achieves a speedup that ranges between a minimum factor of 3 and a maximum factor of 10 (Figure 3c). As the load goes down, the benefits of Eagle tend to decrease, as the abundance of available computational resources makes scheduling decisions less important.

Figure 2 shows that long jobs in Eagle are not negatively affected by the execution of short jobs in the general partition. On the contrary, in some cases, the completion times of long jobs are better in Eagle than in Hawk. This result showcases Eagle's ability to opportunistically allow short jobs to take advantage of computational resources in the general partition without impairing the completion times of long jobs.

We now provide some detailed data to compare the effectiveness and the efficiency of the strategies that Eagle and Hawk implement to combat head-of-line blocking, i.e., SSS and work stealing, respectively. To this end, we show the frequency of head-of-line blocking in both systems, and the message overhead they occur to avoid it. We focus on this aspect of the two systems because Hawk does not provide support for job-awareness.

Table 2 reports the following metrics: *i*) number of probes that are initially scheduled behind a long task; *ii*) number of short tasks that execute after experiencing head-of-line blocking, i.e., that are not “rescued” by stealing; *iii*) number of re-scheduled probes, namely stolen probes for Hawk and probes re-assigned by Eagle during its re-scheduling phase. The data reported in Table 2 are collected under the highest load condition for all three traces. We are going to analyze the Table by looking at its columns from left to right.

The first result that the Table reveals is that, thanks to SSS, Eagle never schedules a probe behind a long task and, as a consequence, totally avoids head-of-line blocking. In contrast, Hawk initially schedules, depending on the trace, from 0.87 to 17.3 million probes behind long jobs, which correspond to 84%, respectively, 67% of the total number of probes sent. Among these probes, 18 thousand to 0.63 million, again depending on the trace, experience head-of-line blocking before they get to the head of their queue and are able to execute a task. For the Google trace, these tasks constitute roughly 5% of the total number of short tasks.

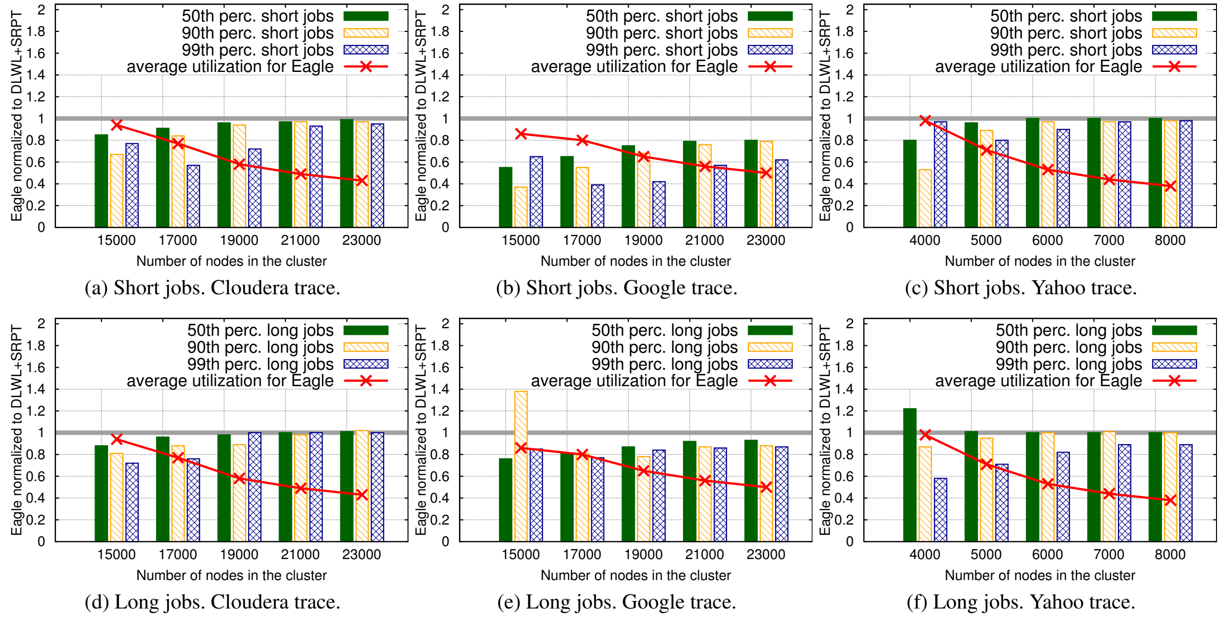
The following column reports the number of probes that are re-scheduled in the two systems, either by work stealing (in Hawk) or by retrying the probe placement (in Eagle). We see that a large fraction of the probes initially scheduled behind a long task is successfully stolen in Hawk (from 90 to 98% depending on the trace), thus mitigating the impact of head-of-line blocking. The number of probes that are re-scheduled with SSS is only marginally higher than the number of probes stolen in Hawk (up to 8% more).

Despite this marginal increase in number of re-scheduled probes, Figure 2 demonstrates that Eagle's scheduling design is more effective than Hawk's. The work stealing in Hawk is a reactive scheme, that is triggered whenever a node becomes idle. This scheme has two main drawbacks. On the one hand, the likelihood of a node being idle is inversely proportional to the load, thus reducing the number of times that work stealing is triggered under high load. On the other hand, whenever a task is stolen, it has probably already experienced some head-of-line blocking, negatively impacting the corresponding job's completion time. In contrast, the proactive nature of Eagle's probe re-scheduling avoids scheduling short tasks behind long ones altogether. This is crucial for task-parallel jobs, especially looking at high percentiles in their completion time distribution, because even failing to steal one of the probes of a job might have a huge impact on the job's completion time.

6.2 Comparing Eagle Against DLWL+SRPT

We compare Eagle to the distributed version of the recent Yaq scheduler [21]. We refer to this design as DLWL+SRPT. Queue reordering in DLWL+SRPT is implemented on top of a distributed version of an LWL-like scheduler, in which the estimated queue waiting times are made available to the distributed schedulers through periodic updates (hereby referred to as heartbeats). For the Google trace, with an average job inter-arrival time of 1s, we use a heartbeat interval of 3s, a value commonly used in the industry. For the Yahoo and Cloudera traces the heartbeat intervals are 7s and 12s. To reduce the chance of conflicts by several distributed schedulers picking the same workers with a low advertised load, we add a small random number (smaller than the heartbeat interval) to each queue waiting time, as done in Apollo [1]. We use the same relative threshold as in Eagle to prevent starvation, and we also reserve the same small partition of the data center for short jobs. Doing so improves DLWL+SRPT's performance, although Yaq's design does not include it. Finally, we use SRPT as a common queue reordering policy for both systems.

Figure 3 (top) shows, as a function of the number of worker nodes, the 50th, 90th, and 99th percentiles of the job completion time distribution for short jobs for Eagle, normalized to those same values in DLWL+SRPT. In addition, we report the average node utilization with the given number of nodes. Figure 3 (bottom) reports the same results for long jobs.



(a) Short jobs. Cloudera trace. (b) Short jobs. Google trace. (c) Short jobs. Yahoo trace.

(d) Long jobs. Cloudera trace. (e) Long jobs. Google trace. (f) Long jobs. Yahoo trace.

Figure 3: 50/90/99th percentiles of job completion times. Eagle normalized to DLWL+SRPT.

For short jobs, Eagle performs better than DLWL+SRPT, because SBP improves SRPT's ability to quickly remove jobs from the system. Moreover, Figure 3 (bottom) shows that Eagle also improves the completion times of long jobs in the vast majority of the cases. The reason is that Eagle does not enqueue short tasks after long ones, so a long task can start executing after all short probes in front of it are serviced. In contrast, in DLWL+SRPT, short tasks can queue after long ones and bypass them due to SRPT, resulting in an additional delay for the long jobs.

DLWL+SRPT is sensitive to the value of heartbeat interval. For the Google trace, Figure 4 shows on a logarithmic scale how DLWL+SRPT is affected by an increase in the heartbeat interval. Increasing the heartbeat interval from 3s to 5s results in a 4.8% increase for the 90th percentile of the short job completion time, and a 2.2% increase for the 99th percentile. When increasing the interval from 3s to 7s, the increases in the 90th and 99th percentiles are 6.8% and 4.2%.

6.3 Breakdown of Eagle's Benefits

In this Section we evaluate the benefits of each of Eagle's components separately. To do so, we compare Eagle to three variants of Eagle, each being stripped of one of Eagle's features: SSS, SBP+SRPT and the minimum number of probes per job.

Figures 5 and 6 report, for the Google and Yahoo traces, respectively, the 50th, 90th and 99th percentiles of short job completion times for the full-fledged Eagle implementation, normalized to the same values for Eagle's variants.

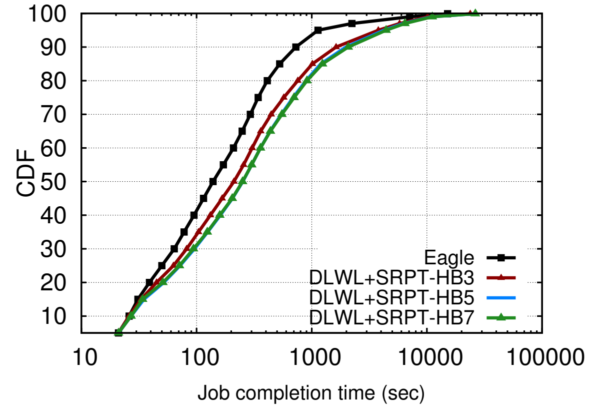
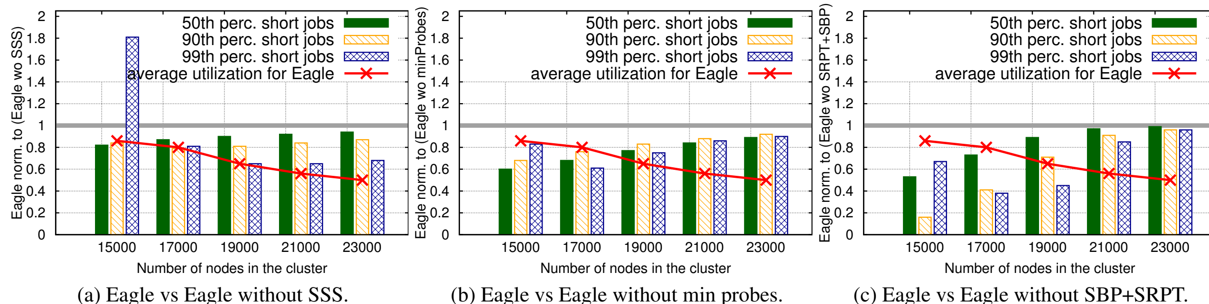


Figure 4: DLWL+SRPT with 3s, 5s and 7s heartbeat interval compared to Eagle. Short jobs, Google trace, 15000 nodes.

The lower the value reported in the graph for a variant without a given feature, the more that feature contributes to Eagle's overall performance. Figures 5 and 6 also report the data center utilization corresponding to each simulated deployment. We do not report results for long jobs, because Eagle's features are largely aimed at improving short job response times.

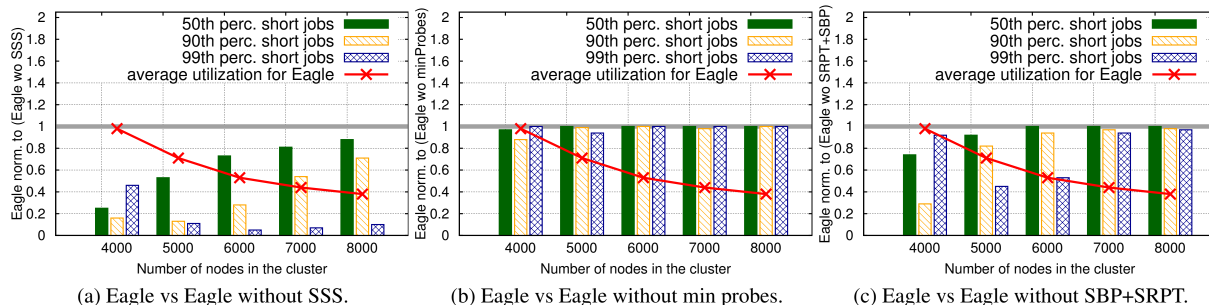
6.3.1 Divide

Figures 5a and 6a showcase the performance of Eagle against Eagle without its SSS component. We see that SSS is a key ingredient to Eagle's success. SSS shows greater benefit for the Yahoo trace, because in that trace there are



(a) Eagle vs Eagle without SSS. (b) Eagle vs Eagle without min probes. (c) Eagle vs Eagle without SBP+SRPT.

Figure 5: Breakdown of Eagle's benefits. Eagle normalized to Eagle without one of its components. Short jobs. Google trace.



(a) Eagle vs Eagle without SSS. (b) Eagle vs Eagle without min probes. (c) Eagle vs Eagle without SBP+SRPT.

Figure 6: Breakdown of Eagle's benefits. Eagle normalized to Eagle without one of its components. Short jobs. Yahoo trace.

more tasks in a job, and so without SSS, there is a higher chance that at least one task of a job is affected by head-of-line blocking. Such blocking is more likely to occur at high load, so for the Yahoo trace that is precisely where SSS shows the greatest benefit. The benefit of SSS is more pronounced for the higher percentiles, as that is where the impact of stragglers is best visible.

For the Google trace, the addition of SSS leads to improvements in all cases, except for the 99th percentile for the highest load. In this extreme case the short partition becomes highly loaded. It thus becomes better to enqueue a probe after a long task in the general partition than after many short tasks in the short partition. We verify this hypothesis by increasing the size of the short partition from 17% to 20% of the workers. This lessens the load in the short partition, and as a result the 99th percentile becomes better for Eagle with SSS.

6.3.2 Stick to Your Probes

Figure 5c shows that SBP+SRPT significantly helps Eagle for the Google trace. Comparing to Figures 5a and 5b we see that SBP+SRPT is the feature that contributes the most to Eagle's performance, because one-task short jobs are abundant in the Google trace and SRPT allows them to bypass other tasks in the queue. SBP+SRPT is also effective for the Yahoo trace, as depicted in Figure 6c. As expected, the benefits of SRPT+SBP diminish at lower loads, when enough workers are idle.

6.3.3 Minimum Number of Probes per Job

Figure 5b and Figure 6b show the effectiveness of sending a minimum number of probes to help jobs with very few tasks. As expected, this feature has a much higher impact in the Google trace than in the Yahoo trace, because of the much higher number of jobs with few tasks in the Google trace. For the Google trace, at high load (15000 and 17000 nodes), setting a minimum number of probes for short jobs achieves a speedup up to 30-40% at all the considered percentiles. Conversely, improvements for the Yahoo trace are more modest, reaching 10% in the higher percentiles. These performance gains come at the cost of a negligible amount of extra messaging overhead. Sending 20 probes, in fact, corresponds to contacting only 0.05% of the nodes in the system in the worst considered case (4000 nodes).

6.4 Comparison Against an Omniscient Scheduler

In order to assess the relative benefits of knowing only the location of long jobs, as provided by SSS, compared to having complete information about the estimated work left in each worker node queue, we compare SSS, combined with a minimum number of probes per job, to an omniscient scheduler that uses this complete information to schedule all

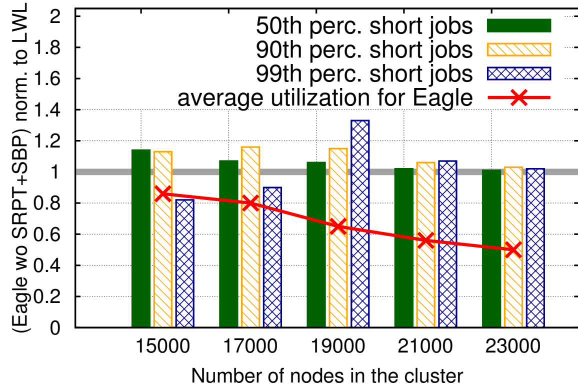


Figure 7: Eagle without SBP+SRPT against an omniscient scheduler. Google trace, 15000 nodes.

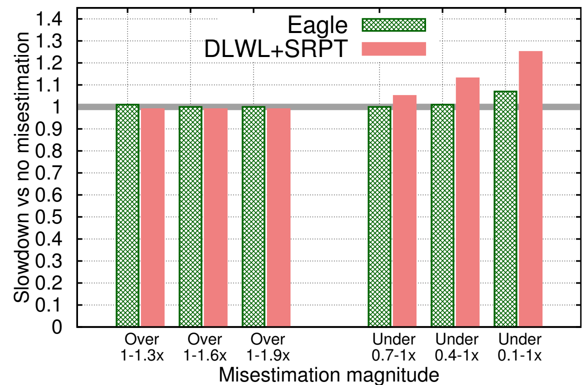


Figure 8: Sensitivity of Eagle and DLWL+SRPT to under and over-estimation of task execution time. Google trace, 15000 nodes, 99th percentile short job completion time.

tasks (short and long) in LWL fashion. The latter provides an upper bound on the performance that can be achieved by using this complete information.

To obtain this upper bound, the omniscient scheduler is enhanced with a (static) data center partitioning scheme with the same configuration as Eagle. Without such a partitioning, LWL is not able to match Eagle, because, especially at high load, several long jobs can occupy most of the data center's resources.

Figure 7 shows that Eagle delivers performance very close to that achieved by an omniscient LWL scheduler. In other words, with only the information about the location of long jobs, Eagle performs almost as well as having the up-to-date estimated queue lengths of all workers. In some cases Eagle is better than LWL, because of late binding as a result of SBP, and because the information on which Eagle relies, namely the location of tasks of long jobs, is smaller and easier to keep up-to-date.

6.5 Sensitivity to Mis-estimation

We analyze the impact of task length mis-estimation on the performance delivered by the job-aware scheduling policies of Eagle and DLWL+SRPT. To do so, we multiply the *estimated* task execution time of every job by a random value, chosen uniformly at random within the range $[0.x, 1]$ for under-estimation and $[1, 1.x]$ for over-estimation. The actual task execution times remain the same, only the estimate used by the scheduler changes. In other words, over-estimation (under-estimation) means that tasks complete faster (slower) than the scheduler expects. We set x to 3, 6 and 9.

Figure 8 shows the slowdown for Eagle and DLWL+SRPT, when mis-estimations occur. The results are for the 99th percentile of the short job completion time. For both systems, the impact of the mis-estimations on the 50th and the 90th percentiles is minimal.

Figure 8 shows that both systems are similarly robust with respect to over-estimation. However, Eagle is more robust to under-estimation than DLWL+SRPT. SBP is the reason behind the robustness. The early binding performed by DLWL+SRPT leads to stragglers, while the late binding in SBP avoids stragglers. This effect is exacerbated by under-estimation.

6.6 Summary

We conclude this Section by showing that, although they may seem orthogonal techniques, SSS, SBP and minProbes are instead complementary, and represent synergistic building blocks of a unified, principled design.

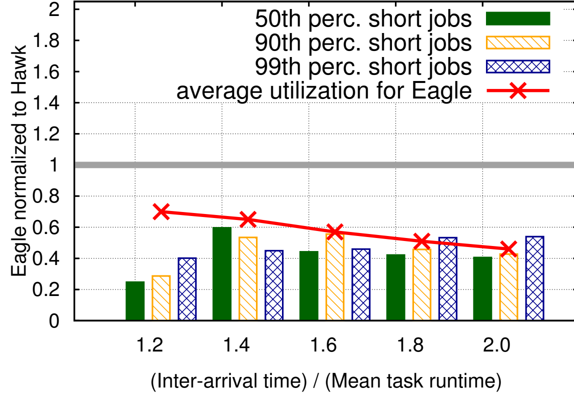
Applying standard SRPT (or any policy based on preemption) in Eagles target system model is not possible, because tasks cannot be preempted. Therefore, SBP uses the time when a task finishes as natural evaluation point to determine the next job to be executed and to pick a task from that job. If we were to allow the entire data center to become overwhelmed with long tasks, then such evaluation points would occur at low frequency, leaving only few opportunities to favor short jobs. Instead, with SSS, even in the case where the general partition is full, short tasks complete at a high rate in the short partition. The result is an implementation of SRPT without preemption, but at a granularity that is fine enough to benefit from it.

The minProbes enhancement further amplifies the gains brought about by the other two techniques. It increases the chances for a probe of a job with few tasks to land on an unloaded node. SSS plays a complementary role in the effectiveness of minProbes, since, in a data center crowded with long tasks, any number of probes would still likely suffer head-of-line blocking, if no nodes were reserved for short tasks.

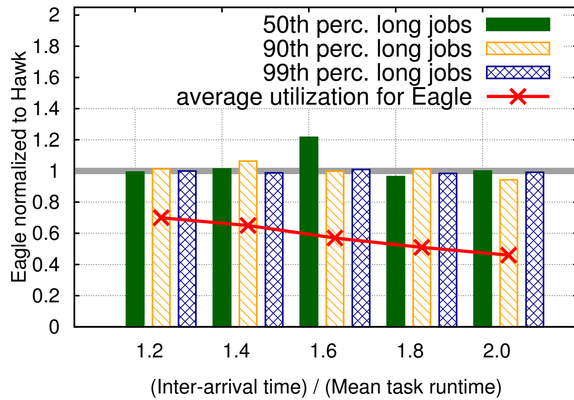
7. Implementation Results

We implement Eagle in the Spark framework [27]. We run an Eagle daemon that runs in each worker node to manage their queue, and a scheduler client as a plug-in for Spark.

We deploy this implementation of Eagle on a 100-node cluster, using a centralized scheduler and ten distributed schedulers. To evaluate its performance, we compare it against an implementation of Hawk, an earlier hybrid scheduler [5].



(a) Short jobs



(b) Long jobs

Figure 9: 50/90/99th percentiles of the job completion times in Eagle, normalized to Hawk for a 3300-job sample of the Google trace.

To keep the time to run experiments tractable, we use a scaled-down version of the Google trace. We use a 3,300-job sample of the trace, and we reduce the duration of each task in the sample by a factor of 1,000 (from seconds to milliseconds). We also reduce the number of tasks in a given job by the ratio between the number of nodes in the original trace and in the sampled-down trace. In order to keep the task-seconds ratio between long and short jobs the same as in the original trace, we increase the task duration in the affected jobs by the same ratio. Job arrival follows a Poisson process, and we vary the cluster load by varying the mean job inter-arrival time as a multiple of the mean task runtime.

Figure 9 presents the 50th, 90th and 99th percentiles of execution times when running with Eagle, normalized to the same values for Hawk, for various ratios of mean inter-arrival time to average job completion time. As this ratio get bigger, the cluster load decreases. In Figure 9a we see that Eagle is better across the board for short jobs. The higher the load, the more improvement we get. For example, the improvement reaches 80% for a ratio of 1.2. As the load decreases, the gains of Eagle over Hawk stabilize, but remain non-negligible, up to 60% for the 50th percentile.

Figure 9b presents the same results for long jobs and shows that the performance delivered by Eagle and Hawk are comparable. This result showcases the ability of Eagle to improve short job completion times without hurting the performance of long jobs.

8. Related Work

Initial schedulers for data centers were centralized [6, 13, 26], with scalability issues as a result [25]. These problems were partially addressed by two-level schedulers, such as Mesos [11] and YARN [25], which separate resource management from the scheduling logic. Scalability issues, however, remained with the centralized resource management entity. In contrast, Eagle schedules a large fraction of the jobs in a distributed manner and only the (usually few) long jobs centrally.

Distributed schedulers, such as Sparrow [20], address the scalability issues more effectively, but can produce poor scheduling decisions, especially in high load scenarios and with common heterogeneous workloads [5]. Instead, Eagle centrally schedules the long jobs, which occupy the vast majority of the resources, and uses the state of the central scheduler to improve the distributed scheduling of short jobs.

A number of shared-state schedulers have also been proposed, such as Apollo [1] and Omega [23]. Here, a separate centralized resource manager maintains shared scheduling state. This state must be updated either by the distributed schedulers [23] or by the worker nodes [1]. In Omega, the distributed schedulers make a scheduling decision based on the shared state, and then atomically update the state. If the state has changed by that time, one of the conflicting schedulers succeeds and the others retry later. In Apollo, the shared state consists of an estimate of the waiting times on all workers. The workers update this information periodically via heartbeats. A distributed scheduler reads the shared state, and assigns tasks to workers based on it. Eagle does not introduce an additional shared entity that needs to be contacted by the distributed schedulers and updated by either the workers or the distributed schedulers. Instead, it relies on an already existing centralized entity, the long job scheduler, that has a good approximation of the state of the cluster, and it uses piggybacking on existing communication in the system to distribute this information to the distributed schedulers. One of the conclusions of this paper is that the knowledge about the placement of only the long jobs and the lazy propagation of information is sufficient to obtain an accurate approximation of the worker nodes in the data center.

In recent work, hybrid schedulers were introduced, such as Hawk [5] and Mercury [14]. They combine a centralized scheduler and a number of distributed schedulers. Mercury uses the centralized scheduler for jobs that need guaranteed resources, while using distributed schedulers for best-effort jobs. To compensate for possible load imbalances introduced by the distributed schedulers, Mercury employs load shedding. Unlike Eagle, however, Mercury is not job aware and load shedding is incapable of eliminating head-of-line blocking.

Hawk introduces the idea of separate scheduling for long and short jobs, and the reservation of a small portion of the cluster for short jobs. Short jobs can be scheduled in the long jobs' partition, to prevent resource under-utilization. This, however, introduces head-of-line blocking. Randomized work stealing is employed to alleviate this issue. As discussed in Section 3, work stealing is not sufficient to eliminate head-of-line blocking. Further, tasks are scheduled in a job unaware fashion. Eagle embraces the hybrid design of Hawk, but fundamentally augments it with SSS, avoiding head-of-line blocking altogether, without imposing much overhead on the centralized scheduler or the rest of the system. Moreover, it enables job awareness by means of SBP.

Queue reordering has been extensively studied in queueing theory [8], and in recent systems [12, 21, 24]. Existing systems implementing queue reordering are typically either centralized [12, 24] or can enforce reordering only within the boundaries of a single worker node [21]. In contrast, by means of SBP, Eagle can enforce cross-nodes queue reordering policies in a scalable fashion, and addresses the problem of straggler tasks.

Finally, a class of schedulers casts the optimal task-to-node allocation as an optimization problem, e.g., Quincy [13], Rayon [4], Tetrisched [24] and Firmament [7]. Computing the solution to such problems yields very good scheduling decisions, but is computationally expensive and needs to be performed in a centralized fashion. These solutions are complementary with respect to Eagle. Eagle can in fact integrate such advanced techniques in its centralized scheduler. This would improve Eagle's scheduling of long jobs, while preserving the scalability of its hybrid design.

9. Conclusion

In this paper we present two new techniques to improve scheduling of data-parallel jobs in data centers. First, we dynamically divide the nodes into different partitions for executing long and short jobs, thereby avoiding head-of-line blocking, the queueing of a short job behind a long one. This division is implemented by a technique called succinct state sharing (SSS), by which distributed schedulers are informed about where long jobs are queued or executing. Second, sticky batch probing (SBP) allows a probe for a short job to request further tasks of that job, thereby avoiding straggler tasks. In combination with SRPT, it favors the expedient completion of short jobs.

We have incorporated these techniques in the Eagle hybrid scheduler, in which the centralized component schedules the long jobs, and the distributed schedulers take care of the short jobs. With such a hybrid scheduler, it becomes particular easy to implement SSS, by simply piggybacking the location of long jobs on scheduling messages. We have implemented Eagle as a Spark plug-in, and measured the performance of this implementation. We have also extensively evaluated Eagle by means of simulation. Our results indicate that Eagle avoids head-of-line blocking altogether, and outperforms various state-of-the-art schedulers.

Acknowledgments

We thank the anonymous reviewers for their feedback. We also thank Ricardo Bianchini, Christos Gkantsidis, Sergey Legtchenko, Baptiste Lepers and Nicolas Schiper for the discussions, feedback and help. This research has been supported in part by a grant from Microsoft Research Cambridge.

References

[1] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. of OSDI*, 2014.

[2] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. of MASCOTS*, 2011.

[3] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. of VLDB Endow.*, 5(12), 2012.

[4] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proc. of SoCC*, 2014.

[5] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of Usenix ATC*, 2015.

[6] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proc. of EuroSys*, 2012.

[7] I. Gog, M. Schwarzkopf, A. Gleave, R. M. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. of OSDI*, 2016, to appear.

[8] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 1st edition, 2013.

[9] M. Harchol-Balter, C. Li, T. Osogami, A. Scheller-Wolf, and M. S. Squillante. Analysis of task assignment with cycle stealing under central queue. In *Proc. of ICDCS*, 2003.

[10] M. Harchol-Balter, A. Scheller-Wolf, and A. R. Young. Surprising results on task assignment in server farms with high-variability workloads. In *Proc. of SIGMETRICS*, 2009.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of NSDI*, 2011.

[12] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proc. of SoCC*, 2015.

[13] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. of SOSP*, 2009.

[14] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of Usenix ATC*, 2015.

[15] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proc. of CCGrid*, 2010.

[16] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[17] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. of STOC*, 1997.

[18] J. D. C. Little. A proof for the queuing formula: L=λw. *Operations Research*, 9(3), 1961.

[19] M. Mitzenmacher. The power of two choices in randomized load balancing. *TPDS*, 12(10), 2001.

[20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. of SOSP*, 2013.

[21] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proc. of EuroSys*, 2016.

[22] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. of SoCC*, 2012.

[23] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proc. of EuroSys*, 2013.

[24] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. of EuroSys*, 2016.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proc. of SoCC*, 2013.

[26] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of EuroSys*, 2010.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI*, 2012.