Sparrow: Distributed, Low Latency Scheduling

Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica
University of California, Berkeley

Abstract

Large-scale data analytics frameworks are shifting towards shorter task durations and larger degrees of parallelism to provide low latency. Scheduling highly parallel jobs that complete in hundreds of milliseconds poses a major challenge for task schedulers, which will need to schedule millions of tasks per second on appropriate machines while offering millisecond-level latency and high availability. We demonstrate that a decentralized, randomized sampling approach provides near-optimal performance while avoiding the throughput and availability limitations of a centralized design. We implement and deploy our scheduler, Sparrow, on a 110-machine cluster and demonstrate that Sparrow performs within 12% of an ideal scheduler.

1 Introduction

Today's data analytics clusters are running ever shorter and higher-fanout jobs. Spurred by demand for lower-latency interactive data processing, efforts in research and industry alike have produced frameworks (e.g., Dremel [12], Spark [26], Impala [11]) that stripe work across thousands of machines or store data in memory in order to analyze large volumes of data in seconds, as shown in Figure 1. We expect this trend to continue with a new generation of frameworks targeting sub-second response times. Bringing response times into the 100ms range will enable powerful new applications; for example, user-facing services will be able to run sophisticated parallel computations, such as language translation and highly personalized search, on a per-query basis.

SOSP '13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522716

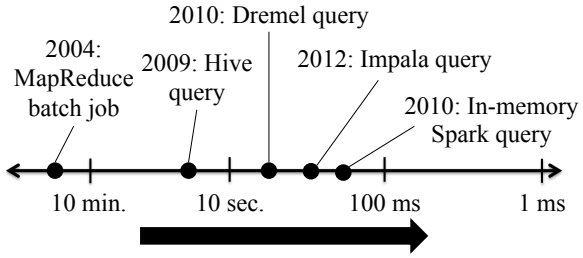


Figure 1: Data analytics frameworks can analyze large volumes of data with ever lower latency.

Jobs composed of short, sub-second tasks present a difficult scheduling challenge. These jobs arise not only due to frameworks targeting low latency, but also as a result of breaking long-running batch jobs into a large number of short tasks, a technique that improves fairness and mitigates stragglers [17]. When tasks run in hundreds of milliseconds, scheduling decisions must be made at very high throughput: a cluster containing ten thousand 16-core machines and running 100ms tasks may require over 1 million scheduling decisions per second. Scheduling must also be performed with low latency: for 100ms tasks, scheduling delays (including queueing delays) above tens of milliseconds represent intolerable overhead. Finally, as processing frameworks approach interactive time-scales and are used in customer-facing systems, high system availability becomes a requirement. These design requirements differ from those of traditional batch workloads.

Modifying today's centralized schedulers to support sub-second parallel tasks presents a difficult engineering challenge. Supporting sub-second tasks requires handling two orders of magnitude higher throughput than the fastest existing schedulers (e.g., Mesos [8], YARN [16], SLURM [10]); meeting this design requirement would be difficult with a design that schedules and launches all tasks through a single node. Additionally, achieving high availability would require the replication or recovery of large amounts of state in sub-second time.

This paper explores the opposite extreme in the design space: we propose scheduling from a set of machines that operate autonomously and without centralized or logically centralized state. A decentralized design offers

attractive scaling and availability properties. The system can support more requests by adding additional schedulers, and if a scheduler fails, users can direct requests to an alternate scheduler. The key challenge with a decentralized design is providing response times comparable to those provided by a centralized scheduler, given that concurrently operating schedulers may make conflicting scheduling decisions.

We present Sparrow, a stateless distributed scheduler that adapts the power of two choices load balancing technique [14] to the domain of parallel task scheduling. The power of two choices technique proposes scheduling each task by probing two random servers and placing the task on the server with fewer queued tasks. We introduce three techniques to make the power of two choices effective in a cluster running parallel jobs:

Batch Sampling: The power of two choices performs poorly for parallel jobs because job response time is sensitive to tail task wait time (because a job cannot complete until its last task finishes) and tail wait times remain high with the power of two choices. Batch sampling solves this problem by applying the recently developed multiple choices approach [18] to the domain of parallel job scheduling. Rather than sampling for each task individually, batch sampling places the m tasks in a job on the least loaded of $d \cdot m$ randomly selected worker machines (for $d > 1$). We demonstrate analytically that, unlike the power of two choices, batch sampling's performance does not degrade as a job's parallelism increases.

Late Binding: The power of two choices suffers from two remaining performance problems: first, server queue length is a poor indicator of wait time, and second, due to messaging delays, multiple schedulers sampling in parallel may experience race conditions. Late binding avoids these problems by delaying assignment of tasks to worker machines until workers are ready to run the task, and reduces median job response time by as much as 45% compared to batch sampling alone.

Policies and Constraints: Sparrow uses multiple queues on worker machines to enforce global policies, and supports the per-job and per-task placement constraints needed by analytics frameworks. Neither policy enforcement nor constraint handling are addressed in simpler theoretical models, but both play an important role in real clusters [21].

We have deployed Sparrow on a 110-machine cluster to evaluate its performance. When scheduling TPC-H queries, Sparrow provides response times within 12% of an ideal scheduler, schedules with median queueing delay of less than 9ms, and recovers from scheduler failures in less than 120ms. Sparrow provides low response times for jobs with short tasks, even in the presence of tasks that take up to 3 orders of magnitude longer. In spite of its decentralized design, Sparrow maintains aggregate fair shares, and isolates users with different priorities such that a misbehaving low priority user increases response times for high priority jobs by at most 40%. Simulation results suggest that Sparrow will continue to perform well as cluster size increases to tens of thousands of cores. Our results demonstrate that distributed scheduling using Sparrow presents a viable alternative to centralized scheduling for low latency, parallel workloads.

2 Design Goals

This paper focuses on fine-grained task scheduling for low-latency applications.

Low-latency workloads have more demanding scheduling requirements than batch workloads do, because batch workloads acquire resources for long periods of time and thus require infrequent task scheduling. To support a workload composed of sub-second tasks, a scheduler must provide millisecond-scale scheduling delay and support millions of task scheduling decisions per second. Additionally, because low-latency frameworks may be used to power user-facing services, a scheduler for low-latency workloads should be able to tolerate scheduler failure.

Sparrow provides fine-grained task scheduling, which is complementary to the functionality provided by cluster resource managers. Sparrow does not launch new processes for each task; instead, Sparrow assumes that a long-running executor process is already running on each worker machine for each framework, so that Sparrow need only send a short task description (rather than a large binary) when a task is launched. These executor processes may be launched within a static portion of a cluster, or via a cluster resource manager (e.g., YARN [16], Mesos [8], Omega [20]) that allocates resources to Sparrow along with other frameworks (e.g., traditional batch workloads).

Sparrow also makes approximations when scheduling and trades off many of the complex features supported by sophisticated, centralized schedulers in order to provide higher scheduling throughput and lower latency. In particular, Sparrow does not allow certain types of placement constraints (e.g., "my job should not be run on machines where User X's jobs are running"), does not perform bin packing, and does not support gang scheduling.

Sparrow supports a small set of features in a way that can be easily scaled, minimizes latency, and keeps the design of the system simple. Many applications run low-latency queries from multiple users, so Sparrow enforces strict priorities or weighted fair shares when aggregate demand exceeds capacity. Sparrow also supports basic

constraints over job placement, such as per-task constraints (e.g. each task needs to be co-resident with input data) and per-job constraints (e.g., all tasks must be placed on machines with GPUs). This feature set is similar to that of the Hadoop MapReduce scheduler [23] and the Spark [26] scheduler.

3 Sample-Based Scheduling for Parallel Jobs

A traditional task scheduler maintains a complete view of which tasks are running on which worker machines, and uses this view to assign incoming tasks to available workers. Sparrow takes a radically different approach: many schedulers operate in parallel, and schedulers do not maintain any state about cluster load. To schedule a job's tasks, schedulers rely on instantaneous load information acquired from worker machines. Sparrow's approach extends existing load balancing techniques [14, 18] to the domain of parallel job scheduling and introduces late binding to improve performance.

3.1 Terminology and job model

We consider a cluster composed of *worker machines* that execute tasks and *schedulers* that assign tasks to worker machines. A job consists of m tasks that are each allocated to a worker machine. Jobs can be handled by any scheduler. Workers run tasks in a fixed number of slots; we avoid more sophisticated bin packing because it adds complexity to the design. If a worker machine is assigned more tasks than it can run concurrently, it queues new tasks until existing tasks release enough resources for the new task to be run. We use *wait time* to describe the time from when a task is submitted to the scheduler until when the task begins executing and *service time* to describe the time the task spends executing on a worker machine. *Job response time* describes the time from when the job is submitted to the scheduler until the last task finishes executing. We use *delay* to describe the total delay within a job due to both scheduling and queueing. We compute delay by taking the difference between the job response time using a given scheduling technique, and job response time if all of the job's tasks had been scheduled with zero wait time (equivalent to the longest service time across all tasks in the job).

In evaluating different scheduling approaches, we assume that each job runs as a single wave of tasks. In real clusters, jobs may run as multiple waves of tasks when, for example, m is greater than the number of slots assigned to the user; for multiwave jobs, the scheduler can place some early tasks on machines with longer queueing delay without affecting job response time. We assume a single wave job model when we evaluate scheduling techniques because single wave jobs are most negatively affected by the approximations involved in our distributed scheduling approach: even a single delayed task affects the job's response time. However, Sparrow also handles multiwave jobs.

3.2 Per-task sampling

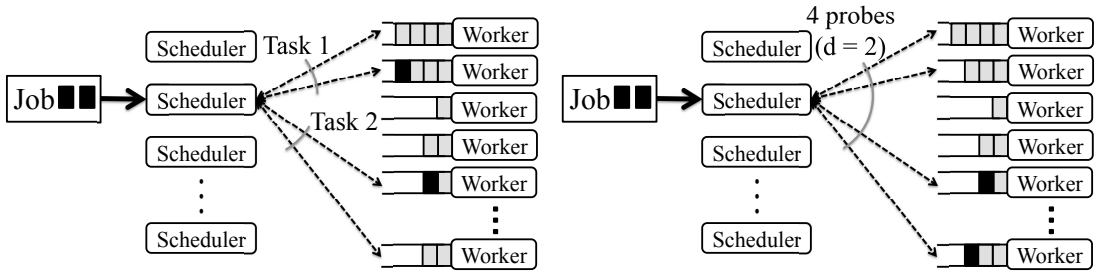
Sparrow's design takes inspiration from the power of two choices load balancing technique [14], which provides low expected task wait times using a stateless, randomized approach. The power of two choices technique proposes a simple improvement over purely random assignment of tasks to worker machines: place each task on the least loaded of two randomly selected worker machines. Assigning tasks in this manner improves expected wait time exponentially compared to using random placement [14].[1]

We first consider a direct application of the power of two choices technique to parallel job scheduling. The scheduler randomly selects two worker machines for each task and sends a *probe* to each, where a probe is a lightweight RPC. The worker machines each reply to the probe with the number of currently queued tasks, and the scheduler places the task on the worker machine with the shortest queue. The scheduler repeats this process for each task in the job, as illustrated in Figure 2(a). We refer to this application of the power of two choices technique as *per-task sampling*.

Per-task sampling improves performance compared to random placement but still performs $2\times$ or more worse than an omniscient scheduler.[2] Intuitively, the problem with per-task sampling is that a job's response time is dictated by the longest wait time of any of the job's tasks, making average *job* response time much higher (and also much more sensitive to tail performance) than average *task* response time. We simulated per-task sampling and random placement in cluster composed of 10,000 4-core machines with 1ms network round trip time. Jobs arrive as a Poisson process and are each composed of 100 tasks. The duration of a job's tasks is chosen from the exponential distribution such that across jobs, task durations are exponentially distributed with mean 100ms, but within a particular job, all tasks are the same du-

[1] More precisely, expected task wait time using random placement is $1/(1-\rho)$, where ρ represents load. Using the least loaded of d choices, wait time in an initially empty system over the first T units of time is upper bounded by $\sum_{i=1}^{\infty} \rho^{\frac{d^i-d}{d-1}} + o(1)$ [14].

[2] The omniscient scheduler uses a greedy scheduling algorithm based on complete information about which worker machines are busy. For each incoming job, the scheduler places the job's tasks on idle workers, if any exist, and otherwise uses FIFO queueing.



(a) Per-task sampling selects queues of length 1 and 3.

(b) Batch sampling selects queues of length 1 and 2.

Figure 2: Placing a parallel, two-task job. Batch sampling outperforms per-task sampling because tasks are placed in the least loaded of the entire *batch* of sampled queues.

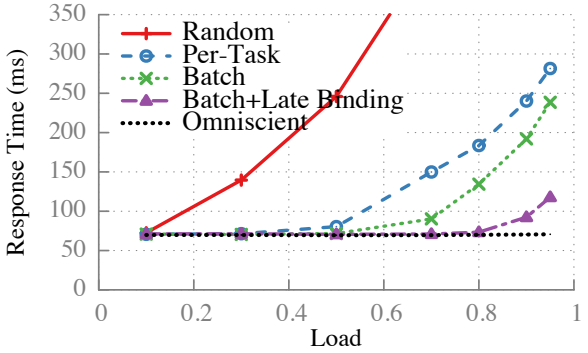


Figure 3: Comparison of scheduling techniques in a simulated cluster of 10,000 4-core machines running 100-task jobs.

ration.[3] As shown in Figure 3, response time increases with increasing load, because schedulers have less success finding free machines on which to place tasks. At 80% load, per-task sampling improves performance by over 3× compared to random placement, but still results in response times equal to over 2.6× those offered by a omniscient scheduler.

3.3 Batch sampling

Batch sampling improves on per-task sampling by sharing information across all of the probes for a particular job. Batch sampling is similar to a technique recently proposed in the context of storage systems [18]. With per-task sampling, one pair of probes may have gotten unlucky and sampled two heavily loaded machines (e.g., Task 1 in Figure 2(a)), while another pair may have gotten lucky and sampled two lightly loaded machines (e.g, Task 2 in Figure 2(a)); one of the two lightly loaded machines will go unused. Batch sampling aggregates load information from the probes sent for all of a job's tasks, and places the job's m tasks on the least loaded of all the worker machines probed. In the example shown in Figure 2, per-task sampling places tasks in queues of length 1 and 3; batch sampling reduces the maximum queue length to 2 by using both workers that were probed by Task 2 with per-task sampling.

To schedule using batch sampling, a scheduler randomly selects dm worker machines (for $d \geq 1$). The scheduler sends a probe to each of the dm workers; as with per-task sampling, each worker replies with the number of queued tasks. The scheduler places one of the job's m tasks on each of the m least loaded workers. Unless otherwise specified, we use $d = 2$; we justify this choice of d in §7.9.

As shown in Figure 3, batch sampling improves performance compared to per-task sampling. At 80% load, batch sampling provides response times 0.73× those with per-task sampling. Nonetheless, response times with batch sampling remain a factor of 1.92× worse than those provided by an omniscient scheduler.

3.4 Problems with sample-based scheduling

Sample-based techniques perform poorly at high load due to two problems. First, schedulers place tasks based on the queue length at worker nodes. However, queue length provides only a coarse prediction of wait time. Consider a case where the scheduler probes two workers to place one task, one of which has two 50ms tasks queued and the other of which has one 300ms task queued. The scheduler will place the task in the queue with only one task, even though that queue will result in a 200ms longer wait time. While workers could reply with an estimate of task duration rather than queue length, accurately predicting task durations is notoriously difficult. Furthermore, almost all task duration estimates would need to be accurate for such a technique

[3] We use this distribution because it puts the most stress on our approximate, distributed scheduling technique. When tasks within a job are of different duration, the shorter tasks can have longer wait times without affecting job response time.

to be effective, because each job includes many parallel tasks, *all* of which must be placed on machines with low wait time to ensure good performance.

Sampling also suffers from a race condition where multiple schedulers concurrently place tasks on a worker that appears lightly loaded [13]. Consider a case where two different schedulers probe the same idle worker machine, w, at the same time. Since w is idle, both schedulers are likely to place a task on w; however, only one of the two tasks placed on the worker will arrive in an empty queue. The queued task might have been placed in a different queue had the corresponding scheduler known that w was not going to be idle when the task arrived.

3.5 Late binding

Sparrow introduces *late binding* to solve the aforementioned problems. With late binding, workers do not reply immediately to probes and instead place a reservation for the task at the end of an internal work queue. When this reservation reaches the front of the queue, the worker sends an RPC to the scheduler that initiated the probe requesting a task for the corresponding job. The scheduler assigns the job's tasks to the first m workers to reply, and replies to the remaining $(d-1)m$ workers with a no-op signaling that all of the job's tasks have been launched. In this manner, the scheduler guarantees that the tasks will be placed on the m probed workers where they will be launched soonest. For exponentially-distributed task durations at 80% load, late binding provides response times $0.55\times$ those with batch sampling, bringing response time to within 5% (4ms) of an omniscient scheduler (as shown in Figure 3).

The downside of late binding is that workers are idle while they are sending an RPC to request a new task from a scheduler. All current cluster schedulers we are aware of make this tradeoff: schedulers wait to assign tasks until a worker signals that it has enough free resources to launch the task. In our target setting, this tradeoff leads to a 2% efficiency loss compared to queueing tasks at worker machines. The fraction of time a worker spends idle while requesting tasks is $(d \cdot \text{RTT})/(t + d \cdot \text{RTT})$ (where d denotes the number of probes per task, RTT denotes the mean network round trip time, and t denotes mean task service time). In our deployment on EC2 with an un-optimized network stack, mean network round trip time was 1 millisecond. We expect that the shortest tasks will complete in 100ms and that scheduler will use a probe ratio of no more than 2, leading to at most a 2% efficiency loss. For our target workload, this tradeoff is worthwhile, as illustrated by the results shown in Figure 3, which incorporate network delays. In environments where network latencies and task runtimes are comparable, late binding will not present a worthwhile tradeoff.

3.6 Proactive Cancellation

When a scheduler has launched all of the tasks for a particular job, it can handle remaining outstanding probes in one of two ways: it can proactively send a cancellation RPC to all workers with outstanding probes, or it can wait for the workers to request a task and reply to those requests with a message indicating that no unlaunched tasks remain. We use our simulation to model the benefit of using proactive cancellation and find that proactive cancellation reduces median response time by 6% at 95% cluster load. At a given load ρ, workers are busy more than ρ of the time: they spend ρ proportion of time executing tasks, but they spend additional time requesting tasks from schedulers. Using cancellation with 1ms network RTT, a probe ratio of 2, and with tasks that are an average of 100ms long reduces the time workers spend busy by approximately 1%; because response times approach infinity as load approaches 100%, the 1% reduction in time workers spend busy leads to a noticeable reduction in response times. Cancellation leads to additional RPCs if a worker receives a cancellation for a reservation after it has already requested a task for that reservation: at 95% load, cancellation leads to 2% additional RPCs. We argue that the additional RPCs are a worthwhile tradeoff for the improved performance, and the full Sparrow implementation includes cancellation. Cancellation helps more when the ratio of network delay to task duration increases, so will become more important as task durations decrease, and less important as network delay decreases.

4 Scheduling Policies and Constraints

Sparrow aims to support a small but useful set of policies within its decentralized framework. This section describes support for two types of popular scheduler policies: constraints over where individual tasks are launched and inter-user isolation policies to govern the relative performance of users when resources are contended.

4.1 Handling placement constraints

Sparrow handles two types of constraints, per-job and per-task constraints. Such constraints are commonly required in data-parallel frameworks, for instance, to run tasks on a machine that holds the task's input data on disk or in memory. As mentioned in §2, Sparrow

does not support many types of constraints (e.g., inter-job constraints) supported by some general-purpose resource managers.

Per-job constraints (e.g., all tasks should be run on a worker with a GPU) are trivially handled at a Sparrow scheduler. Sparrow randomly selects the dm candidate workers from the subset of workers that satisfy the constraint. Once the dm workers to probe are selected, scheduling proceeds as described previously.

Sparrow also handles jobs with per-task constraints, such as constraints that limit tasks to run on machines where input data is located. Co-locating tasks with input data typically reduces response time, because input data does not need to be transferred over the network. For jobs with per-task constraints, each task may have a different set of machines on which it can run, so Sparrow cannot aggregate information over all of the probes in the job using batch sampling. Instead, Sparrow uses per-task sampling, where the scheduler selects the two machines to probe for each task from the set of machines that the task is constrained to run on, along with late binding.

Sparrow implements a small optimization over per-task sampling for jobs with per-task constraints. Rather than probing individually for each task, Sparrow shares information across tasks when possible. For example, consider a case where task 0 is constrained to run in machines A, B, and C, and task 1 is constrained to run on machines C, D, and E. Suppose the scheduler probed machines A and B for task 0, which were heavily loaded, and probed machines C and D for task 1, which were both idle. In this case, Sparrow will place task 0 on machine C and task 1 on machine D, even though both machines were selected to be probed for task 1.

Although Sparrow cannot use batch sampling for jobs with per-task constraints, our distributed approach still provides near-optimal response times for these jobs, because even a centralized scheduler has only a small number of choices for where to place each task. Jobs with per-task constraints can still use late binding, so the scheduler is guaranteed to place each task on whichever of the two probed machines where the task will run sooner. Storage layers like HDFS typically replicate data on three different machines, so tasks that read input data will be constrained to run on one of three machines where the input data is located. As a result, even an ideal, omniscient scheduler would only have one additional choice for where to place each task.

4.2 Resource allocation policies

Cluster schedulers seek to allocate resources according to a specific policy when aggregate demand for resources exceeds capacity. Sparrow supports two types of policies: strict priorities and weighted fair sharing. These policies mirror those offered by other schedulers, including the Hadoop Map Reduce scheduler [25].

Many cluster sharing policies reduce to using strict priorities; Sparrow supports all such policies by maintaining multiple queues on worker nodes. FIFO, earliest deadline first, and shortest job first all reduce to assigning a priority to each job, and running the highest priority jobs first. For example, with earliest deadline first, jobs with earlier deadlines are assigned higher priority. Cluster operators may also wish to directly assign priorities; for example, to give production jobs high priority and ad-hoc jobs low priority. To support these policies, Sparrow maintains one queue for each priority at each worker node. When resources become free, Sparrow responds to the reservation from the highest priority non-empty queue. This mechanism trades simplicity for accuracy: nodes need not use complex gossip protocols to exchange information about jobs that are waiting to be scheduled, but low priority jobs may run before high priority jobs if a probe for a low priority job arrives at a node where no high priority jobs happen to be queued. We believe this is a worthwhile tradeoff: as shown in §7.8, this distributed mechanism provides good performance for high priority users. Sparrow does not currently support preemption when a high priority task arrives at a machine running a lower priority task; we leave exploration of preemption to future work.

Sparrow can also enforce weighted fair shares. Each worker maintains a separate queue for each user, and performs weighted fair queuing [6] over those queues. This mechanism provides cluster-wide fair shares in expectation: two users using the same worker will get shares proportional to their weight, so by extension, two users using the same set of machines will also be assigned shares proportional to their weight. We choose this simple mechanism because more accurate mechanisms (e.g., Pisces [22]) add considerable complexity; as we demonstrate in §7.7, Sparrow's simple mechanism provides near-perfect fair shares.

5 Analysis

Before delving into our experimental evaluation, we analytically show that batch sampling achieves near-optimal performance, *regardless of the task duration distribution*, given some simplifying assumptions. Section 3 demonstrated that Sparrow performs well, but only under one particular workload; this section generalizes those results to all workloads. We also demonstrate that with per-task sampling, performance decreases exponentially with the number of tasks in a job, making it poorly suited for parallel workloads.

n	Number of servers in the cluster
ρ	Load (fraction non-idle workers)
m	Tasks per job
d	Probes per task
t	Mean task service time
$\rho n/(mt)$	Mean request arrival rate

Table 1: Summary of notation.

Random Placement	$(1-\rho)^m$
Per-Task Sampling	$(1-\rho^d)^m$
Batch Sampling	$\sum_{i=m}^{d\cdot m}(1-\rho)^i\rho^{d\cdot m-i}\binom{d\cdot m}{i}$

Table 2: Probability that a job will experience zero wait time under three different scheduling techniques.

To analyze the performance of batch and per-task sampling, we examine the probability of placing all tasks in a job on idle machines, or equivalently, providing zero wait time. Quantifying how often our approach places jobs on idle workers provides a bound on how Sparrow performs compared to an optimal scheduler.

We make a few simplifying assumptions for the purpose of this analysis. We assume zero network delay, an infinitely large number of servers, and that each server runs one task at a time. Our experimental evaluation shows results in the absence of these assumptions.

Mathematical analysis corroborates the results in §3 demonstrating that per-task sampling performs poorly for parallel jobs. The probability that a particular task is placed on an idle machine is one minus the probability that all probes hit busy machines: $1-\rho^d$ (where ρ represents cluster load and d represents the probe ratio; Table 1 summarizes notation). The probability that *all* tasks in a job are assigned to idle machines is $(1-\rho^d)^m$ (as shown in Table 2) because all m sets of probes must hit at least one idle machine. This probability decreases exponentially with the number of tasks in a job, rendering per-task sampling inappropriate for scheduling parallel jobs. Figure 4 illustrates the probability that a job experiences zero wait time for both 10 and 100-task jobs, and demonstrates that the probability of experiencing zero wait time for a 100-task job drops to $< 2\%$ at 20% load.

Batch sampling can place all of a job's tasks on idle machines at much higher loads than per-task sampling. In expectation, batch sampling will be able to place all m tasks in empty queues as long as $d \geq 1/(1-\rho)$. Crucially, this expression does not depend on the number of tasks in a job (m). Figure 4 illustrates this effect: for both 10 and 100-task jobs, the probability of experiencing zero wait time drops from 1 to 0 at 50% load.[4]

[4]With the larger, 100-task job, the drop happens more rapidly because the job uses more total probes, which decreases the variance in the proportion of probes that hit idle machines.

Figure 4: Probability that a job will experience zero wait time in a single-core environment using random placement, sampling 2 servers/task, and sampling $2m$ machines to place an m-task job.

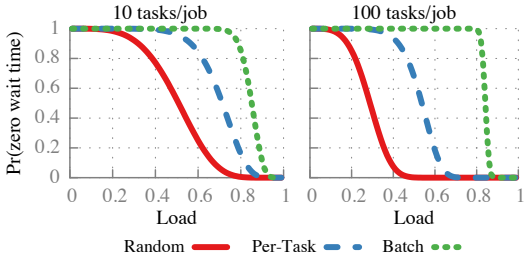


Figure 5: Probability that a job will experience zero wait time in a system of 4-core servers.

Our analysis thus far has considered machines that can run only one task at a time; however, today's clusters typically feature multi-core machines. Multicore machines significantly improve the performance of batch sampling. Consider a model where each server can run up to c tasks concurrently. Each probe implicitly describes load on c processing units rather than just one, which increases the likelihood of finding an idle processing unit on which to run each task. To analyze performance in a multicore environment, we make two simplifying assumptions: first, we assume that the probability that a core is idle is independent of whether other cores on the same machine are idle; and second, we assume that the scheduler places at most 1 task on each machine, even if multiple cores are idle (placing multiple tasks on an idle machine exacerbates the "gold rush effect" where many schedulers concurrently place tasks on an idle machine). Based on these assumptions, we can replace ρ in Table 2 with ρ^c to obtain the results shown in Figure 5. These results improve dramatically on the single-core results: for batch sampling with 4 cores per machine and 100 tasks per job, batch sampling achieves near perfect performance (99.9% of jobs experience zero wait time) at up to 79% load. This result demonstrates that, under some simplifying assumptions, batch sampling performs well regardless of the distribution of task durations.

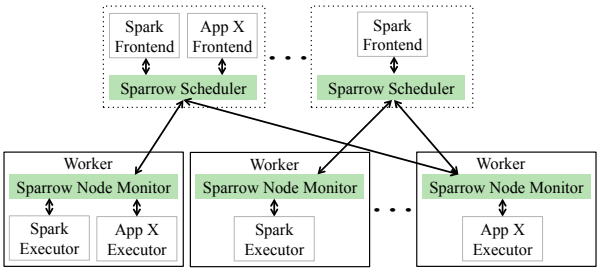


Figure 6: Frameworks that use Sparrow are decomposed into frontends, which generate tasks, and executors, which run tasks. Frameworks schedule jobs by communicating with any one of a set of distributed Sparrow schedulers. Sparrow node monitors run on each worker machine and federate resource usage.

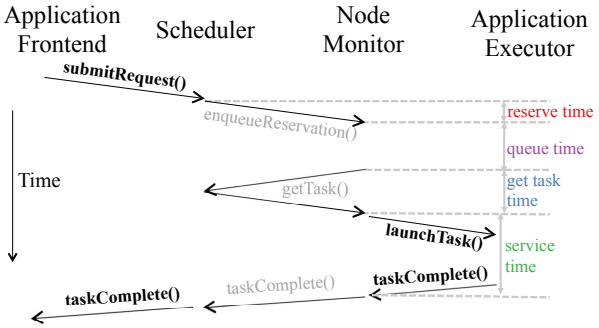


Figure 7: RPCs (parameters not shown) and timings associated with launching a job. Sparrow's external interface is shown in bold text and internal RPCs are shown in grey text.

6 Implementation

We implemented Sparrow to evaluate its performance on a cluster of 110 Amazon EC2 virtual machines. The Sparrow code, including scripts to replicate our experimental evaluation, is publicly available at `http://github.com/radlab/sparrow`.

6.1 System components

As shown in Figure 6, Sparrow schedules from a distributed set of schedulers that are each responsible for assigning tasks to workers. Because Sparrow does not require any communication between schedulers, arbitrarily many schedulers may operate concurrently, and users or applications may use any available scheduler to place jobs. Schedulers expose a service (illustrated in Figure 7) that allows frameworks to submit job scheduling requests using Thrift remote procedure calls [1]. Thrift can generate client bindings in many languages, so applications that use Sparrow for scheduling are not tied to a particular language. Each scheduling request includes a list of task specifications; the specification for a task includes a task description and a list of constraints governing where the task can be placed.

A Sparrow node monitor runs on each worker, and federates resource usage on the worker by enqueuing reservations and requesting task specifications from schedulers when resources become available. Node monitors run tasks in a fixed number of *slots*; slots can be configured based on the resources of the underlying machine, such as CPU cores and memory.

Sparrow performs task scheduling for one or more concurrently operating frameworks. As shown in Figure 6, frameworks are composed of long-lived *frontend* and *executor* processes, a model employed by many systems (e.g., Mesos [8]). Frontends accept high level queries or job specifications (e.g., a SQL query) from exogenous sources (e.g., a data analyst, web service, business application, etc.) and compile them into parallel tasks for execution on workers. Frontends are typically distributed over multiple machines to provide high performance and availability. Because Sparrow schedulers are lightweight, in our deployment, we run a scheduler on each machine where an application frontend is running to ensure minimum scheduling latency.

Executor processes are responsible for executing tasks, and are long-lived to avoid startup overhead such as shipping binaries or caching large datasets in memory. Executor processes for multiple frameworks may run co-resident on a single machine; the node monitor federates resource usage between co-located frameworks. Sparrow requires executors to accept a `launchTask()` RPC from a local node monitor, as shown in Figure 7; Sparrow uses the `launchTask()` RPC to pass on the task description (opaque to Sparrow) originally supplied by the application frontend.

6.2 Spark on Sparrow

In order to test Sparrow using a realistic workload, we ported Spark [26] to Sparrow by writing a Spark scheduling plugin. This plugin is 280 lines of Scala code, and can be found at `https://github.com/kayousterhout/spark/tree/sparrow`.

The execution of a Spark query begins at a Spark frontend, which compiles a functional query definition into multiple parallel stages. Each stage is submitted as a Sparrow job, including a list of task descriptions and any associated placement constraints. The first stage is typically constrained to execute on machines that contain input data, while the remaining stages (which read data shuffled or broadcasted over the network) are unconstrained. When one stage completes, Spark requests scheduling of the tasks in the subsequent stage.

6.3 Fault tolerance

Because Sparrow schedulers do not have any logically centralized state, the failure of one scheduler does not affect the operation of other schedulers. Frameworks that were using the failed scheduler need to detect the failure and connect to a backup scheduler. Sparrow includes a Java client that handles failover between Sparrow schedulers. The client accepts a list of schedulers from the application and connects to the first scheduler in the list. The client sends a heartbeat message to the scheduler it is using every 100ms to ensure that the scheduler is still alive; if the scheduler has failed, the client connects to the next scheduler in the list and triggers a callback at the application. This approach allows frameworks to decide how to handle tasks that were in-flight during the scheduler failure. Some frameworks may choose to ignore failed tasks and proceed with a partial result; for Spark, the handler instantly relaunches any phases that were in-flight when the scheduler failed. Frameworks that elect to re-launch tasks must ensure that tasks are idempotent, because the task may have been partway through execution when the scheduler died. Sparrow does not attempt to learn about in-progress jobs that were launched by the failed scheduler, and instead relies on applications to relaunch such jobs. Because Sparrow is designed for short jobs, the simplicity benefit of this approach outweighs the efficiency loss from needing to restart jobs that were in the process of being scheduled by the failed scheduler.

While Sparrow's design allows for scheduler failures, Sparrow does not provide any safeguards against rogue schedulers. A misbehaving scheduler could use a larger probe ratio to improve performance, at the expensive of other jobs. In trusted environments where schedulers are run by a trusted entity (e.g., within a company), this should not be a problem; in more adversarial environments, schedulers may need to be authenticated and rate-limited to prevent misbehaving schedulers from wasting resources.

Sparrow does not handle worker failures, as discussed in §8, nor does it handle the case where the entire cluster fails. Because Sparrow does not persist scheduling state to disk, in the event that all machines in the cluster fail (for example, due to a power loss event), all jobs that were in progress will need to be restarted. As in the case when a scheduler fails, the efficiency loss from this approach is minimal because jobs are short.

7 Experimental Evaluation

We evaluate Sparrow using a cluster composed of 100 worker machines and 10 schedulers running on Amazon EC2. Unless otherwise specified, we use a probe ratio of 2. First, we use Sparrow to schedule tasks for a TPC-H workload, which features heterogeneous analytics queries. We provide fine-grained tracing of the overhead that Sparrow incurs and quantify its performance in comparison with an ideal scheduler. Second, we demonstrate Sparrow's ability to handle scheduler failures. Third, we evaluate Sparrow's ability to isolate users from one another in accordance with cluster-wide scheduling policies. Finally, we perform a sensitivity analysis of key parameters in Sparrow's design.

7.1 Performance on TPC-H workload

We measure Sparrow's performance scheduling queries from the TPC-H decision support benchmark. The TPC-H benchmark is representative of ad-hoc queries on business data, which are a common use case for low-latency data parallel frameworks.

Each TPC-H query is executed using Shark [24], a large scale data analytics platform built on top of Spark [26]. Shark queries are compiled into multiple Spark stages that each trigger a scheduling request using Sparrow's `submitRequest()` RPC. Tasks in the first stage are constrained to run on one of three machines holding the task's input data, while tasks in remaining stages are unconstrained. The response time of a query is the sum of the response times of each stage. Because Shark is resource-intensive, we use EC2 high-memory quadruple extra large instances, which each have 8 cores and 68.4GB of memory, and use 4 slots on each worker. Ten different users launch random permutations of the TPC-H queries to sustain an average cluster load of 80% for a period of approximately 15 minutes. We report response times from a 200 second period in the middle of the experiment; during the 200 second period, Sparrow schedules over 20k jobs that make up 6.2k TPC-H queries. Each user runs queries on a distinct denormalized copy of the TPC-H dataset; each copy of the data set is approximately 2GB (scale factor 2) and is broken into 33 partitions that are each triply replicated in memory.

The TPC-H query workload has four qualities representative of a real cluster workload. First, cluster utilization fluctuates around the mean value of 80% depending on whether the users are collectively in more resource-intensive or less resource-intensive stages. Second, the stages have different numbers of tasks: the first stage has 33 tasks, and subsequent stages have either 8 tasks (for reduce-like stages that read shuffled data) or 1 task (for aggregation stages). Third, the duration of each stage is non-uniform, varying from a few tens of milliseconds to several hundred. Finally, the queries have a mix of constrained and unconstrained scheduling requests: 6.2k requests are constrained (the first stage in each query) and the remaining 14k requests are unconstrained.

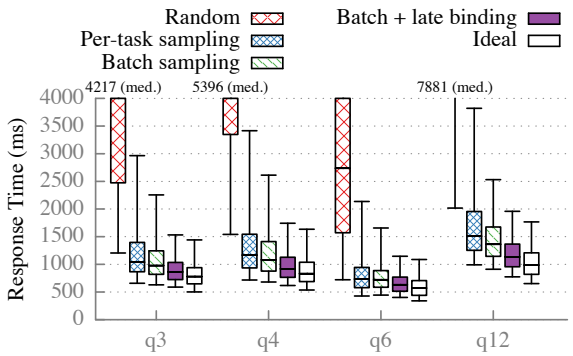


Figure 8: Response times for TPC-H queries using different placement stategies. Whiskers depict 5th and 95th percentiles; boxes depict median, 25th, and 75th percentiles.

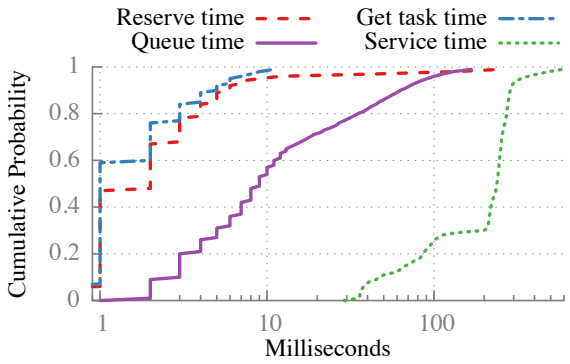


Figure 9: Latency distribution for each phase in the Sparrow scheduling algorithm.

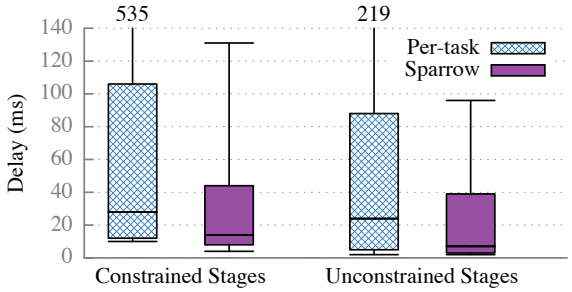


Figure 10: Delay using both Sparrow and per-task sampling, for both constrained and unconstrained Spark stages. Whiskers depict 5th and 95 percentiles; boxes depict median, 25th, and 75th percentiles.

To evaluate Sparrow's performance, we compare Sparrow to an ideal scheduler that always places all tasks with zero wait time, as described in §3.1. To compute the ideal response time for a query, we compute the response time for each stage if all of the tasks in the stage had been placed with zero wait time, and then sum the ideal response times for all stages in the query. Sparrow always satisfies data locality constraints; because the ideal response times are computed using the service times when Sparrow executed the job, the ideal response time assumes data locality for all tasks. The ideal response time does not include the time needed to send tasks to worker machines, nor does it include queueing that is inevitable during utilization bursts, making it a conservative lower bound on the response time attainable with a centralized scheduler.

Figure 8 demonstrates that Sparrow outperforms alternate techniques and provides response times within 12% of an ideal scheduler. Compared to randomly assigning tasks to workers, Sparrow (batch sampling with late binding) reduces median query response time by 4–8× and reduces 95th percentile response time by over 10×. Sparrow also reduces response time compared to per-task sampling (a naïve implementation based on the power of two choices): batch sampling with late binding provides query response times an average of 0.8× those provided by per-task sampling. Ninety-fifth percentile response times drop by almost a factor of two with Sparrow, compared to per-task sampling. Late binding reduces median query response time by an average of 14% compared to batch sampling alone. Sparrow also provides good absolute performance: Sparrow provides median response times just 12% higher than those provided by an ideal scheduler.

7.2 Deconstructing performance

To understand the components of the delay that Sparrow adds relative to an ideal scheduler, we deconstruct Sparrow scheduling latency in Figure 9. Each line corresponds to one of the phases of the Sparrow scheduling algorithm depicted in Figure 7. The reserve time and queue times are unique to Sparrow—a centralized scheduler might be able to reduce these times to zero. However, the get task time is unavoidable: no matter the scheduling algorithm, the scheduler will need to ship the task to the worker machine.

7.3 How do task constraints affect performance?

Sparrow provides good absolute performance and improves over per-task sampling for both constrained and unconstrained tasks. Figure 10 depicts the delay for constrained and unconstrained stages in the TPC-H workload using both Sparrow and per-task sampling. Sparrow schedules with a median of 7ms of delay for jobs with unconstrained tasks and a median of 14ms of delay for jobs with constrained tasks; because Sparrow cannot aggregate information across the tasks in a job when tasks are constrained, delay is longer. Nonetheless, even for

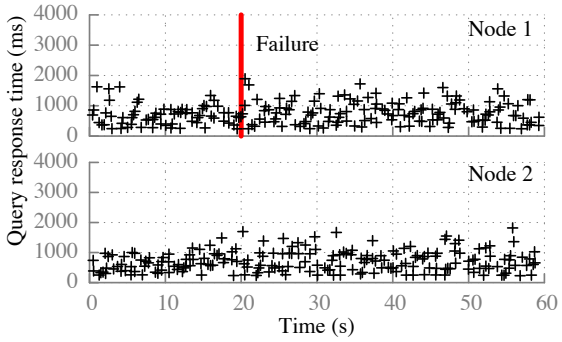


Figure 11: TPC-H response times for two frontends submitting queries to a 100-node cluster. Node 1 suffers from a scheduler failure at 20 seconds.

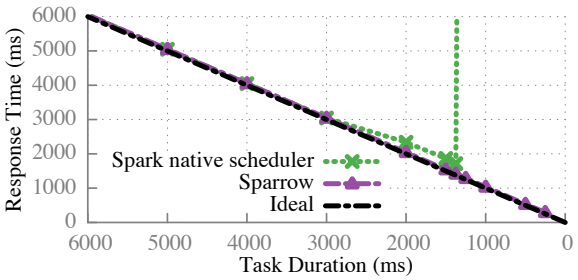


Figure 12: Response time when scheduling 10-task jobs in a 100 node cluster using both Sparrow and Spark's native scheduler. Utilization is fixed at 80%, while task duration decreases.

constrained tasks, Sparrow provides a performance improvement over per-task sampling due to its use of late binding.

7.4 How do scheduler failures impact job response time?

Sparrow provides automatic failover between schedulers and can failover to a new scheduler in less than 120ms. Figure 11 plots the response time for ongoing TPC-H queries in an experiment parameterized as in §7.1, with 10 Shark frontends that submit queries. Each frontend connects to a co-resident Sparrow scheduler but is initialized with a list of alternate schedulers to connect to in case of failure. At time `t=20`, we terminate the Sparrow scheduler on node 1. The plot depicts response times for jobs launched from the Spark frontend on node 1, which fails over to the scheduler on node 2. The plot also shows response times for jobs launched from the Spark frontend on node 2, which uses the scheduler on node 2 for the entire duration of the experiment. When the Sparrow scheduler on node 1 fails, it takes 100ms for the Sparrow client to detect the failure, less than 5ms to for the Sparrow client to connect to the scheduler on node 2, and less than 15ms for Spark to relaunch all outstanding tasks. Because of the speed at which failure recovery occurs, only 2 queries have tasks in flight during the failure; these queries suffer some overhead.

7.5 Synthetic workload

The remaining sections evaluate Sparrow using a synthetic workload composed of jobs with constant duration tasks. In this workload, ideal job completion time is always equal to task duration, which helps to isolate the performance of Sparrow from application-layer variations in service time. As in previous experiments, these experiments run on a cluster of 110 EC2 servers, with 10 schedulers and 100 workers.

7.6 How does Sparrow compare to Spark's native, centralized scheduler?

Even in the relatively small, 100-node cluster in which we conducted our evaluation, Spark's existing centralized scheduler cannot provide high enough throughput to support sub-second tasks.[5] We use a synthetic workload where each job is composed of 10 tasks that each sleep for a specified period of time, and measure job response time. Since all tasks in the job are the same duration, ideal job response time (if all tasks are launched immediately) is the duration of a single task. To stress the schedulers, we use 8 slots on each machine (one per core). Figure 12 depicts job response time as a function of task duration. We fix cluster load at 80%, and vary task submission rate to keep load constant as task duration decreases. For tasks longer than 2 seconds, Sparrow and Spark's native scheduler both provide near-ideal response times. However, when tasks are shorter than 1355ms, Spark's native scheduler cannot keep up with the rate at which tasks are completing so jobs experience infinite queueing.

To ensure that Sparrow's distributed scheduling is necessary, we performed extensive profiling of the Spark scheduler to understand how much we could increase scheduling throughput with improved engineering. We did not find any one bottleneck in the Spark scheduler; instead, messaging overhead, virtual function call overhead, and context switching lead to a best-case throughput (achievable when Spark is scheduling only a single job) of approximately 1500 tasks per second. Some of these factors could be mitigated, but at the expense of code readability and understandability. A clus-

[5] For these experiments, we use Spark's standalone mode, which relies on a simple, centralized scheduler. Spark also allows for scheduling using Mesos; Mesos is more heavyweight and provides worse performance than standalone mode for short tasks.

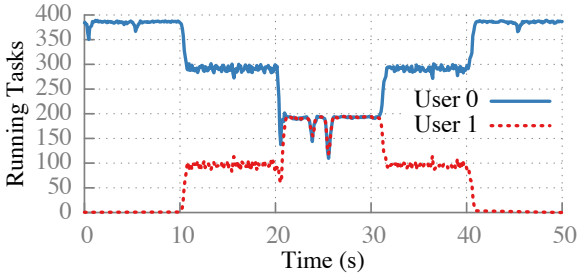


Figure 13: Cluster share used by two users that are each assigned equal shares of the cluster. User 0 submits at a rate to utilize the entire cluster for the entire experiment while user 1 adjusts its submission rate each 10 seconds. Sparrow assigns both users their max-min fair share.

HP load	LP load	HP response time in ms	LP response time in ms
0.25	0	106 (111)	N/A
0.25	0.25	108 (114)	108 (115)
0.25	0.5	110 (148)	110 (449)
0.25	0.75	136 (170)	40.2k (46.2k)
0.25	1.75	141 (226)	255k (270k)

Table 3: Median and 95th percentile (shown in parentheses) response times for a high priority (HP) and low priority (LP) user running jobs composed of 10 100ms tasks in a 100-node cluster. Sparrow successfully shields the high priority user from a low priority user. When aggregate load is 1 or more, response time will grow to be unbounded for at least one user.

ter with tens of thousands of machines running subsecond tasks may require millions of scheduling decisions per second; supporting such an environment would require 1000× higher scheduling throughput, which is difficult to imagine even with a significant rearchitecting of the scheduler. Clusters running low latency workloads will need to shift from using centralized task schedulers like Spark's native scheduler to using more scalable distributed schedulers like Sparrow.

7.7 How well can Sparrow's distributed fairness enforcement maintain fair shares?

Figure 13 demonstrates that Sparrow's distributed fairness mechanism enforces cluster-wide fair shares and quickly adapts to changing user demand. Users 0 and 1 are both given equal shares in a cluster with 400 slots. Unlike other experiments, we use 100 4-core EC2 machines; Sparrow's distributed enforcement works better as the number of cores increases, so to avoid over stating performance, we evaluate it under the smallest number of cores we would expect in a cluster today. User 0 submits at a rate to fully utilize the cluster for the entire duration of the experiment. User 1 changes her demand every 10 seconds: she submits at a rate to consume 0%, 25%, 50%, 25%, and finally 0% of the cluster's available slots. Under max-min fairness, each user is allocated her fair share of the cluster unless the user's demand is less than her share, in which case the unused share is distributed evenly amongst the remaining users. Thus, user 1's max-min share for each 10-second interval is 0 concurrently running tasks, 100 tasks, 200 tasks, 100 tasks, and finally 0 tasks; user 0's max-min fair share is the remaining resources. Sparrow's fairness mechanism lacks any central authority with a complete view of how many tasks each user is running, leading to imperfect fairness over short time intervals. Nonetheless, as shown in Figure 13, Sparrow quickly allocates enough resources to User 1 when she begins submitting scheduling requests (10 seconds into the experiment), and the cluster share allocated by Sparrow exhibits only small fluctuations from the correct fair share.

7.8 How much can low priority users hurt response times for high priority users?

Table 3 demonstrates that Sparrow provides response times within 40% of an ideal scheduler for a high priority user in the presence of a misbehaving low priority user. This experiment uses workers that each have 16 slots. The high priority user submits jobs at a rate to fill 25% of the cluster, while the low priority user increases her submission rate to well beyond the capacity of the cluster. Without any isolation mechanisms, when the aggregate submission rate exceeds the cluster capacity, both users would experience infinite queueing. As described in §4.2, Sparrow node monitors run all queued high priority tasks before launching any low priority tasks, allowing Sparrow to shield high priority users from misbehaving low priority users. While Sparrow prevents the high priority user from experiencing infinite queueing delay, the high priority user still experiences 40% worse response times when sharing with a demanding low priority user than when running alone on the cluster. This is because Sparrow does not use preemption: high priority tasks may need to wait to be launched until low priority tasks complete. In the worst case, this wait time may be as long as the longest running low-priority task. Exploring the impact of preemption is a subject of future work.

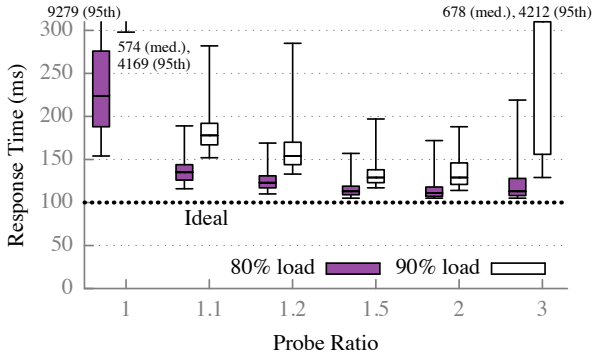


Figure 14: Effect of probe ratio on job response time at two different cluster loads. Whiskers depict 5th and 95th percentiles; boxes depict median, 25th, and 75th percentiles.

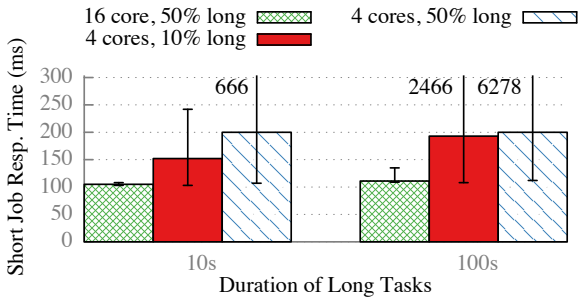


Figure 15: Sparrow provides low median response time for jobs composed of 10 100ms tasks, even when those tasks are run alongside much longer jobs. Error bars depict 5th and 95th percentiles.

7.9 How sensitive is Sparrow to the probe ratio?

Changing the probe ratio affects Sparrow's performance most at high cluster load. Figure 14 depicts response time as a function of probe ratio in a 110-machine cluster of 8-core machines running the synthetic workload (each job has 10 100ms tasks). The figure demonstrates that using a small amount of oversampling significantly improves performance compared to placing tasks randomly: oversampling by just 10% (probe ratio of 1.1) reduces median response time by more than 2.5× compared to random sampling (probe ratio of 1) at 90% load. The figure also demonstrates a sweet spot in the probe ratio: a low probe ratio negatively impacts performance because schedulers do not oversample enough to find lightly loaded machines, but additional oversampling eventually hurts performance due to increased messaging. This effect is most apparent at 90% load; at 80% load, median response time with a probe ratio of 1.1 is just 1.4× higher than median response time with a larger probe ratio of 2. We use a probe ratio of 2 throughout our evaluation to facilitate comparison with the power of two choices and because non-integral probe ratios are not possible with constrained tasks.

7.10 Handling task heterogeneity

Sparrow does not perform as well under extreme task heterogeneity: if some workers are running long tasks, Sparrow schedulers are less likely to find idle machines on which to run tasks. Sparrow works well unless a large fraction of tasks are long *and* the long tasks are many orders of magnitude longer than the short tasks. We ran a series of experiments with two types of jobs: short jobs, composed of 10 100ms tasks, and long jobs, composed of 10 tasks of longer duration. Jobs are submitted to sustain 80% cluster load. Figure 15 illustrates the response time of short jobs when sharing the cluster with long jobs. We vary the percentage of jobs that are long, the duration of the long jobs, and the number of cores on the machine, to illustrate where performance breaks down. Sparrow provides response times for short tasks within 11% of ideal (100ms) when running on 16-core machines, even when 50% of tasks are 3 orders of magnitude longer. When 50% of tasks are 3 orders of magnitude longer, over 99% of the execution time across all jobs is spent executing long tasks; given this, Sparrow's performance is impressive. Short tasks see more significant performance degredation in a 4-core environment.

7.11 Scaling to large clusters

We used simulation to evaluate Sparrow's performance in larger clusters. Figure 3 suggests that Sparrow will continue to provide good performance in a 10,000 node cluster; of course, the only way to conclusively evaluate Sparrow's performance at scale will be to deploy it on a large cluster.

8 Limitations and Future Work

To handle the latency and throughput demands of low-latency frameworks, our approach sacrifices features available in general purpose resource managers. Some of these limitations of our approach are fundamental, while others are the focus of future work.

Scheduling policies When a cluster becomes oversubscribed, Sparrow supports aggregate fair-sharing or priority-based scheduling. Sparrow's distributed setting lends itself to *approximated* policy enforcement in order to minimize system complexity; exploring whether Sparrow can provide more exact policy enforcement

without adding significant complexity is a focus of future work. Adding pre-emption, for example, would be a simple way to mitigate the effects of low-priority users' jobs on higher priority users.

Constraints Our current design does not handle *inter-job constraints* (e.g. "the tasks for job A must not run on racks with tasks for job B"). Supporting inter-job constraints across frontends is difficult to do without significantly altering Sparrow's design.

Gang scheduling Some applications require gang scheduling, a feature not implemented by Sparrow. Gang scheduling is typically implemented using bin-packing algorithms that search for and reserve time slots in which an entire job can run. Because Sparrow queues tasks on several machines, it lacks a central point from which to perform bin-packing. While Sparrow often places all jobs on entirely idle machines, this is not guaranteed, and deadlocks between multiple jobs that require gang scheduling may occur. Sparrow is not alone: many cluster schedulers do not support gang scheduling [8, 9, 16].

Query-level policies Sparrow's performance could be improved by adding query-level scheduling policies. A user query (e.g., a SQL query executed using Shark) may be composed of many stages that are each executed using a separate Sparrow scheduling request; to optimize *query* response time, Sparrow should schedule queries in FIFO order. Currently, Sparrow's algorithm attempts to schedule jobs in FIFO order; adding query-level scheduling policies should improve end-to-end query performance.

Worker failures Handling worker failures is complicated by Sparrow's distributed design, because when a worker fails, all schedulers with outstanding requests at that worker must be informed. We envision handling worker failures with a centralized state store that relies on occasional heartbeats to maintain a list of currently alive workers. The state store would periodically disseminate the list of live workers to all schedulers. Since the information stored in the state store would be soft state, it could easily be recreated in the event of a state store failure.

Dynamically adapting the probe ratio Sparrow could potentially improve performance by dynamically adapting the probe ratio based on cluster load; however, such an approach sacrifices some of the simplicity of Sparrow's current design. Exploring whether dynamically changing the probe ratio would significantly increase performance is the subject of ongoing work.

9 Related Work

Scheduling in distributed systems has been extensively studied in earlier work. Most existing cluster schedulers rely on centralized architectures. Among logically decentralized schedulers, Sparrow is the first to schedule all of a job's tasks together, rather than scheduling each task independently, which improves performance for parallel jobs.

Dean's work on reducing the latency tail in serving systems [5] is most similar to ours. He proposes using hedged requests where the client sends each request to two workers and cancels remaining outstanding requests when the first result is received. He also describes tied requests, where clients send each request to two servers, but the servers communicate directly about the status of the request: when one server begins executing the request, it cancels the counterpart. Both mechanisms are similar to Sparrow's late binding, but target an environment where each task needs to be scheduled independently (for data locality), so information cannot be shared across the tasks in a job.

Work on load sharing in distributed systems (e.g., [7]) also uses randomized techniques similar to Sparrow's. In load sharing systems, each processor both generates and processes work; by default, work is processed where it is generated. Processors re-distribute queued tasks if the number of tasks queued at a processor exceeds some threshold, using either receiver-initiated policies, where lightly loaded processors request work from randomly selected other processors, or sender-initiated policies, where heavily loaded processors offload work to randomly selected recipients. Sparrow represents a combination of sender-initiated and receiver-initiated policies: schedulers ("senders") initiate the assignment of tasks to workers ("receivers") by sending probes, but workers finalize the assignment by responding to probes and requesting tasks as resources become available.

Projects that explore load balancing tasks in multiprocessor shared-memory architectures (e.g., [19]) echo many of the design tradeoffs underlying our approach, such as the need to avoid centralized scheduling points. They differ from our approach because they focus on a single machine where the majority of the effort is spent determining when to *re*schedule processes amongst cores to balance load.

Quincy [9] targets task-level scheduling in compute clusters, similar to Sparrow. Quincy maps the scheduling problem onto a graph in order to compute an optimal schedule that balances data locality, fairness, and starvation freedom. Quincy's graph solver supports more sophisticated scheduling policies than Sparrow but takes over a second to compute a scheduling assignment in a 2500 node cluster, making it too slow for our target workload.

In the realm of data analytics frameworks, Dremel [12] achieves response times of seconds with extremely high fanout. Dremel uses a hierarchical

scheduler design whereby each query is decomposed into a serving tree; this approach exploits the internal structure of Dremel queries so is not generally applicable.

Many schedulers aim to allocate resources at coarse granularity, either because tasks tend to be long-running or because the cluster supports many applications that each acquire some amount of resources and perform their own task-level scheduling (e.g., Mesos [8], YARN [16], Omega [20]). These schedulers sacrifice request granularity in order to enforce complex scheduling policies; as a result, they provide insufficient latency and/or throughput for scheduling sub-second tasks. High performance computing schedulers fall into this category: they optimize for large jobs with complex constraints, and target maximum throughput in the tens to hundreds of scheduling decisions per second (e.g., SLURM [10]). Similarly, Condor supports complex features including a rich constraint language, job checkpointing, and gang scheduling using a heavy-weight matchmaking process that results in maximum scheduling throughput of 10 to 100 jobs per second [4].

In the theory literature, a substantial body of work analyzes the performance of the power of two choices load balancing technique, as summarized by Mitzenmacher [15]. To the best of our knowledge, no existing work explores performance for parallel jobs. Many existing analyses consider placing balls into bins, and recent work [18] has generalized this to placing multiple balls concurrently into multiple bins. This analysis is not appropriate for a scheduling setting, because unlike bins, worker machines process tasks to empty their queue. Other work analyzes scheduling for single tasks; parallel jobs are fundamentally different because a parallel job cannot complete until the *last* of a large number of tasks completes.

Straggler mitigation techniques (e.g., Dolly [2], LATE [27], Mantri [3]) focus on variation in task execution time (rather than task wait time) and are complementary to Sparrow. For example, Mantri launches a task on a second machine if the first version of the task is progressing too slowly, a technique that could easily be used by Sparrow's distributed schedulers.

10 Conclusion

This paper presents Sparrow, a stateless decentralized scheduler that provides near optimal performance using two key techniques: batch sampling and late binding. We use a TPC-H workload to demonstrate that Sparrow can provide median response times within 12% of an ideal scheduler and survives scheduler failures. Sparrow enforces popular scheduler policies, including fair sharing and strict priorities. Experiments using a synthetic workload demonstrate that Sparrow is resilient to different probe ratios and distributions of task durations. In light of these results, we believe that distributed scheduling using Sparrow presents a viable alternative to centralized schedulers for low latency parallel workloads.

11 Acknowledgments

We are indebted to Aurojit Panda for help with debugging EC2 performance anomalies, Shivaram Venkataraman for insightful comments on several drafts of this paper and for help with Spark integration, Sameer Agarwal for help with running simulations, Satish Rao for help with theoretical models of the system, and Peter Bailis, Ali Ghodsi, Adam Oliner, Sylvia Ratnasamy, and Colin Scott for helpful comments on earlier drafts of this paper. We also thank our shepherd, John Wilkes, for helping to shape the final version of the paper. Finally, we thank the reviewers from HotCloud 2012, OSDI 2012, NSDI 2013, and SOSP 2013 for their helpful feedback.

This research is supported in part by a Hertz Foundation Fellowship, the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program, NSF CISE Expeditions award CCF-1139158, DARPA XData Award FA8750-12-2-0331, Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

References

[1] Apache Thrift. `http://thrift.apache.org`.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why Let Resources Idle? Aggressive Cloning of Jobs with Dolly. In *HotCloud*, 2012.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. OSDI*, 2010.

[4] D. Bradley, T. S. Clair, M. Farrellee, Z. Guo, M. Livny, I. Sfiligoi, and T. Tannenbaum. An Update on the Scalability Limits of the Condor Batch System. *Journal of Physics: Conference Series*, 331(6), 2011.

[5] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), February 2013.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. SIGCOMM*, 1989.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 1986.

[8] 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. NSDI*, 2011.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. SOSP*, 2009.

[10] M. A. Jette, A. B. Yoo, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proc. Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 44–60. Springer, 2003.

[11] M. Kornacker and J. Erickson. Cloudera Impala: Real Time Queries in Apache Hadoop, For Real. `http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/`, October 2012.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.*, 2010.

[13] M. Mitzenmacher. How Useful is Old Information? volume 11, pages 6–20, 2000.

[14] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Computing*, 12(10):1094–1104, 2001.

[15] M. Mitzenmacher. The Power of Two Random Choices: A Survey of Techniques and Results. In S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, editors, *Handbook of Randomized Computing*, volume 1, pages 255–312. Springer, 2001.

[16] A. C. Murthy. The Next Generation of Apache MapReduce. `http://developer.yahoo.com/blogs/hadoop/next-generation-apache-hadoop-mapreduce-3061.html`, February 2012.

[17] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *Proc. HotOS*, 2013.

[18] G. Park. A Generalization of Multiple Choice Balls-into-Bins. In *Proc. PODC*, pages 297–298, 2011.

[19] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proc. SPAA*, 1991.

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

[21] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In *Proc. SOCC*, 2011.

[22] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. OSDI*, 2012.

[23] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[24] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. SIGMOD*, 2013.

[25] 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. EuroSys*, 2010.

[26] 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. NSDI*, 2012.

[27] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI*, 2008.