Scalable Scheduling for Sub-Second Parallel Jobs

Patrick Wendell

Scalable Scheduling for Sub-Second Parallel Jobs

By

Patrick Mellen Wendell

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Michael J. Franklin
Professor Scott Shenker

Spring 2013

Abstract

Scalable Scheduling for Sub-Second Parallel Jobs

by

Patrick Mellen Wendell

Master of Science in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Large-scale data analytics frameworks are shifting towards shorter task durations and larger degrees of parallelism to provide low latency. However, scheduling highly parallel jobs that complete in hundreds of milliseconds poses a major challenge for cluster schedulers, which will need to place millions of tasks per second on appropriate nodes 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 real cluster and demonstrate that Sparrow performs within 14% of an ideal scheduler.

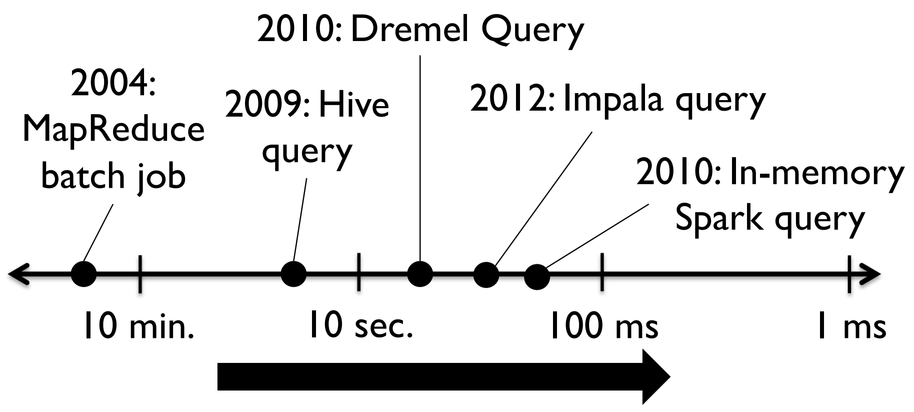


Figure 1: Data analytics frameworks can analyze large amounts of data with decreasing latency.

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 [16], Spark [30], Hadapt [2], Impala [15]) that stripe work across thousands of machines or store data in memory in order to complete jobs 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 user-facing services to run sophisticated parallel computations on a per-query basis, such as language translation and highly personalized search.

Providing low response times for parallel jobs that execute on thousands of machines poses a significant scheduling challenge. Parallel jobs are composed of many (often hundreds or thousands) of concurrent tasks that each run on a single machine. Response time is determined by the last task to complete, so *every* task needs to be scheduled carefully: even a single task placed on a contended machine can increase response time.

Sub-second parallel jobs amplify the scheduling challenge. 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 well over 1 million scheduling decisions per second. Scheduling must also be performed with low latency: for 100ms tasks, scheduling 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 substantially from those of batch workloads.

Designing a traditional, centralized scheduler that supports 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 [12], YARN [19], SLURM [14]); meeting this design requirement will 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 by asking how well a completely decentralized task scheduler can perform. We propose scheduling from a set of machines that operate autonomously and without shared state. Such a decentralized design offers attractive scaling and availability properties: the system can support more requests by adding additional schedulers and if a particular scheduler fails, users can direct requests to a different scheduler. For this reason, in other domains such as web request load balancing, decentralized architectures are commonplace. Many of these architectures [8, 10] build on the power of two choices technique [17], in which the scheduler simply probes two random servers and picks the less loaded one.

However, a decentralized scheduler based on the power of two choices must address three challenges to provide low response times for *parallel* jobs. First, as we show analytically, power of two sampling performs poorly as jobs become increasingly parallel. A parallel job finishes only when its last task finishes and thus its response time depends heavily on the tail distribution of its task duration, which remains high even with the power of two choices. Second, due to messaging delays, multiple schedulers sampling in parallel may experience race conditions. Third, the power of two choices requires workers to estimate the durations of tasks in their queues, which is notoriously difficult.

To address these challenges, we present Sparrow, a stateless distributed task scheduler that is scalable and highly resilient. Sparrow extends simpler sampling approaches using two core techniques: *batch sampling* and *virtual reservations*. Batch-sampling applies the recently developed multiple choices approach [20] to the domain of parallel job scheduling. With batch-sampling, a scheduler places the m tasks in a job on the least loaded of dm randomly selected worker machines (for $d > 1$). We show that, unlike the power of two choices, batch sampling's performance does not degrade as the job's parallelism increases. With virtual reservations, node monitors queue probes until they are ready to run the task. This eliminates the need to estimate task durations and eliminates race conditions due to multiple schedulers making concurrent decisions.

We have implemented Sparrow in a working cluster and evaluated its performance. When scheduling TPC-H queries on a 100-node cluster, Sparrow provides response times within 14% of an optimal scheduler and schedules with fewer than 8 milliseconds of queueing delay. Sparrow provides low response times for 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 (without resorting to preemption) such that a misbehaving low priority user increases response times for high priority jobs by at most 41%. Simulation results demonstrate that Sparrow continues to perform well as cluster size increases to tens of thousands of cores.

In summary, we make the following contributions:

- We propose Sparrow, a *decentralized scheduler* that is highly scalable and resilient.
- We introduce *batch sampling*, a scheduling technique that, unlike the power of two choices [17], does not lead to larger response times as the parallelism of jobs increases.
- We introduce *virtual reservations* that, together with batch-sampling, allow Sparrow to closely approach the performance of an optimal scheduler.
- We show that in spite of its decentralized design, Sparrow supports common global policies, such as proportional and priority scheduling.

2 Design Goals

This paper focuses on task scheduling for low-latency, data-intensive applications. Such applications typically decrease latency by fanning work out over large numbers of machines. As a result, their workload is composed of many small, parallel tasks. This stands in contrast to batch frameworks which acquire resources for long periods of time. The scheduler's job is to place these tasks expediently on worker machines. Short-task workloads result in a set of unique scheduling requirements:

Low latency: To ensure that scheduling delay is not a substantial fraction of job completion time, the scheduler must provide at most *millisecond-scale scheduling delay*.

High throughput: To handle clusters with tens of thousands of nodes (and correspondingly hundreds of thousands of cores), the scheduler must support *millions of task scheduling decisions per second*.

High availability: Cluster operators already go to great lengths to increase the availability of centralized batch schedulers. We expect that low-latency frameworks will be used to power user-facing services, making *high availability an operating requirement*.

To meet these requirements, we are willing to forgo many features of sophisticated centralized resource managers. In particular, we do not design for arbitrarily long tasks that may run for days or weeks, we do not allow complex placement constraints (e.g., "my job should not be run on any machines where User X's jobs are running"), we do not perform bin packing, and we do not support gang scheduling. To co-exist with long-running, batch jobs, our scheduler runs tasks in a statically or dynamically allocated portion of the cluster that has been allocated by a more general resource manager such as YARN [19], Mesos [12], Omega [22], or vSphere [3].

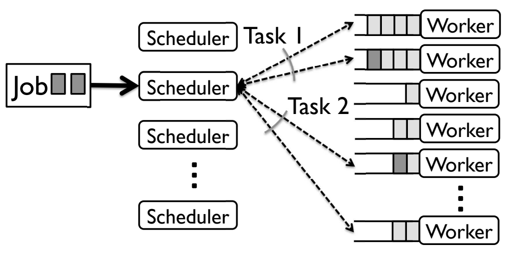
Our key focus is on supporting a small set of features in a way that can be easily scaled, minimizes latency, and keeps the design of the system simple. Many applications wish to run low-latency queries from multiple users, so a scheduler should enforce sensible resource allocation policies when aggregate demand exceeds capacity. We also aim to support basic constraints over job placement, such as task-level constraints (e.g. each task needs to be co-resident with input data) and job-level constraints (e.g., all tasks must be placed on machines with GPUs). This feature set is similar to that of the Hadoop MapReduce scheduler [25] and the Spark [30] scheduler.

3 Sample-Based Scheduling for Parallel Jobs

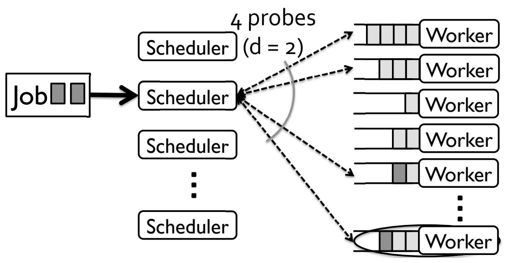
Traditional cluster schedulers maintain a complete view of which tasks are running on which worker machines, and use this view to assign incoming tasks to available workers. To support low-latency workloads, Sparrow takes a radically different approach: schedulers maintain no state about cluster load and instead place tasks based on instantaneous load information acquired from worker machines. Sparrow's approach extends existing load balancing techniques [17, 20] to the domain of parallel job scheduling and introduces virtual reservations to address practical problems.

3.1 Terminology

We consider a cluster composed of *worker machines* that execute tasks and *schedulers* that assign tasks to worker machines. A scheduling request consists of m tasks that are allocated to worker machines. Scheduling requests can be handled by any scheduler; a scheduler assigns each task



(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.

in the request to a worker machine. 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. *Response time* describes the time from when the request is submitted to the scheduler until the last task finishes executing.

3.2 Per-Task Sampling

Sparrow's design takes inspiration from the power of two choices load balancing technique [17], which provides near-optimal 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. Mitzenmacher demonstrated that assigning tasks in this manner improves expected wait time exponentially compared to using random placement [17].[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 of the two worker machines, 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 provides high tail wait times when jobs are parallel. Intuitively, for jobs that contain a large number of tasks, every job is expected to experience tail task wait time, making response times with per-task sampling 2x or more worse than optimal placement. We simulated per-task sampling and random placement in a 10,000 node cluster running 500-task jobs where the duration of each task

[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 $\sum_{i=1}^{\infty} \rho^{\frac{d^i-d}{d-1}} + o(1)$ [17].

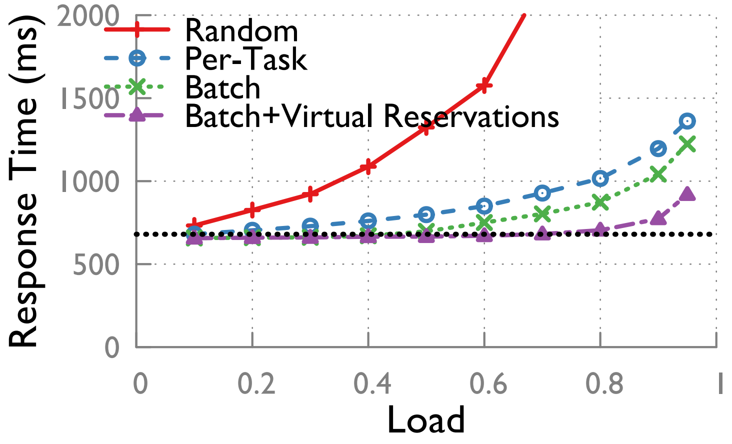


Figure 3: Comparison of random placement, per-task sampling, batch sampling, and batch sampling with virtual reservations in a simulated 10,000 node cluster running 500-task jobs. Task durations are exponentially distributed with mean 100ms; expected optimal job response time is 680ms (shown with the black dashed line).

is exponentially distributed with mean 100ms. Because job response time is determined by the last of 500 tasks to complete, the expected optimal job response time (if all tasks are scheduled with no wait time) is 680ms. We assume 1ms network RTTs, and we model jobs arrivals as a Poisson process. As shown in Figure 3, response time increases with increasing load, because schedulers have increasing difficulty finding free machines on which to place tasks. At 90% load, per-task sampling improves performance by a factor of 5 compared to random placement, but still adds 75% overhead compared to an optimal scheduler.

3.3 Batch Sampling

Batch sampling improves on per-task sampling by sharing information across all of the probes for a particular scheduling request. Batch sampling is similar to a technique recently proposed in the context of storage systems [20]. 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 specificed, we use $d = 2$; we explore the impact of d in §7.7.

As shown in Figure 3, batch sampling improves performance compared to per-task sampling. With exponentially distributed task durations, batch sampling reduces job response time by over 10% compared to per-task sampling. For other task duration distributions including constant and Pareto, batch sampling improves performance by a factor of two (results omitted for brevity). Nonetheless, batch sampling adds over 50% of overhead at 90% load compared to an optimal 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.

Sampling also suffers from a race condition where multiple schedulers may concurrently place tasks on a worker that appears lightly loaded. Consider a case where two different schedulers probe the same idle worker machine, w, at the same time. Since the worker machine 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 Virtual Reservations

Sparrow introduces *virtual reservations* to solve the aforementioned problems. With virtual reservations, 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 requesting a specific task. 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 signalling 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 90% load, virtual reservations improve response time by 35% compared to batch sampling, bringing response time to within 14% of optimal (as shown in Figure 3).

The downside of virtual reservations 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. 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 other environments where network latencies and task runtimes are the same order of magnitude, virtual reservations will not present a worthwhile tradeoff.

4 Scheduling Policies and Constraints

Sparrow aims to support a small but useful set of policies within its decentralized framework. This section outlines 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, job-level and task-level constraints. Such constraints are commonly required in data-parallel frameworks, for instance, to run tasks co-resident with a machine that holds data on disk or in memory. As mentioned in §2, Sparrow does not support complex constraints (e.g., inter-job constraints) supported by some general-purpose resource managers.

Job-level 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.

For jobs with task-level constraints, Sparrow uses per-task rather than batch sampling. 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.

Sparrow implements a small optimization over per-task sampling for jobs with task-level 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 task-level constraints, our distributed approach still provides near-optimal response times for these jobs. Jobs with task level constraints can still use virtual reservations, so the scheduler is guaranteed to place each task on whichever of two probed machines that the task will run soonest. Because task-level constraints typically constrain a task to run on three machines, 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 [29].

Many cluster sharing policies reduce to using strict priorities; Sparrow supports all such poli-

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

Table 1: Summary of notation.

cies 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 §7.6, 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 [9] 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 [23]) add considerable complexity; as we demonstrate in §7.5, Sparrow's simple mechanism provides near-perfect fair shares.

5 Analysis

Before delving into our experimental evaluation, we use mathematical analysis to prove that batch sampling, the key technique underlying Sparrow, achieves near-optimal performance, *regardless of the task duration distribution*. Section 3 demonstrated that Sparrow performs well, but only under one particular workload; this section generalizes those results to all workloads. Batch sampling's good performance comes in contrast to the performance of per-task sampling: the performance of per-task sampling decreases exponentially with the number of tasks in a job, making it perform poorly for parallel workloads.

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. Because an ideal, omniscient scheduler could place all jobs on idle machines when the the cluster is under 100% utilized, 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 purposes of this analysis. We assume zero

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 diffelacement approaches.

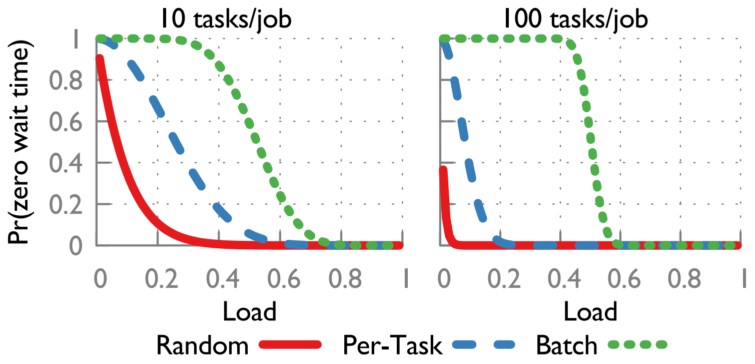


Figure 4: Probability that a job will experience zero wait time in a single-core environment, using random placement, sampling 2 servers/task, and sampling $2m$ machines to place an m-task job.

network delay, an infinitely large number of servers, and that each server runs one task at a time. Our experimental evaluation evaluates 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 (equivalent to the probability that all of a job's tasks are placed in idle queues) for both 10 and 100-task jobs, and demonstrates that the probability of experiencing zero wait time for a 100-task job drops to less than 2% at just 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 $d \geq \frac{1}{1-\rho}$. 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.[2]

Our analysis thus far has considered machines that can run only one task at a time; however, today's clusters typicaly feature multi-core machines. Multicore machines significantly improve

[2]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.

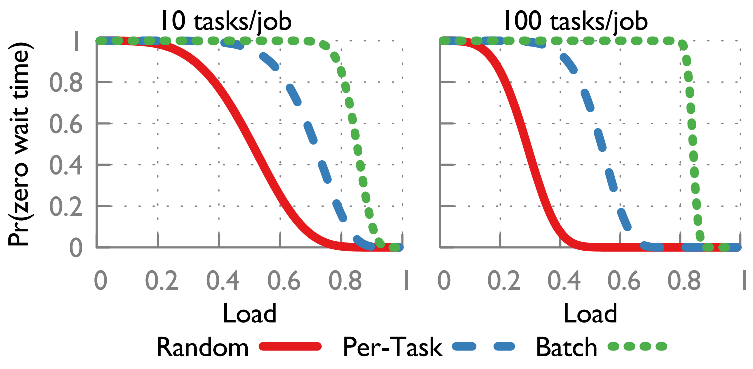


Figure 5: Probability that a job will experience zero wait time in a system of 4-core servers.

the performance of batch sampling. Consider a model where each server can run up to c tasks concurrently without degraded performance compared to running the task on an idle machine. Because each server can run multiple concurrent tasks, 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 envirionment, we make two simplifying assumptions: first, we assume that the distribution of idle cores is independent of whether the cores reside on the same machine; and second, we assume that the scheduler places at most 1 task on each machine, even if multiple cores are idle. 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 batch sampling performs well *regardless of the distribution of task durations*.

6 Implementation

We implemented Sparrow to evalute its performance in a running cluster. 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 slaves. Because batch sampling 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 cross-platform Thrift service [1] (illustrated in Figure 7) to allow frameworks to submit scheduling requests. 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 slave, and federates resource usage on the slave by

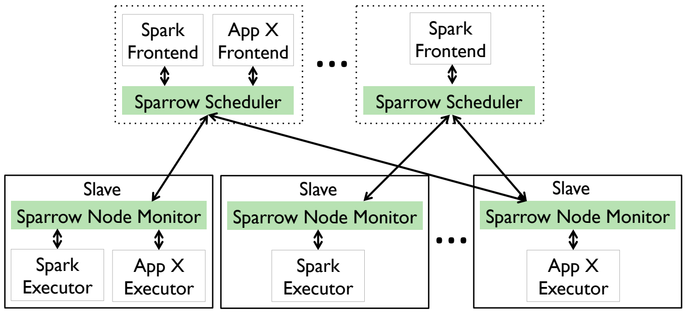


Figure 6: Frameworks that use Sparrow are decomposed into frontends, which generate tasks, and executors, which run tasks. Frameworks schedule jobs by communicating with any one of a set of distributed Sparrow schedulers. Sparrow node monitors run on each slave machine and federate resource usage.

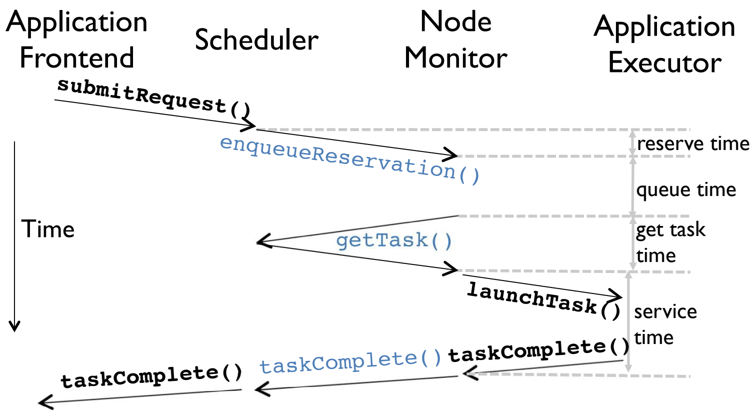


Figure 7: RPCs (parameters not shown) and timings associated with launching a job. Sparrow's external interface is shown in bold text.

enqueuing virtual 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 [12]). 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. Executor processes are responsible for executing tasks, and are long-lived to avoid startup overhead such as shipping binaries or bringing large datasets into cache. 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; the `launchTask()` RPC includes a task description (opaque to Sparrow) provided by the application frontend.

6.2 Spark on Sparrow

In order to test batch sampling using a realistic workload, we have ported Spark [30] to Sparrow by writing a Spark scheduling plugin that uses the Java Sparrow client. This plugin is 132 lines of Scala code.

The execution of a Spark job begins at a Spark frontend, which compiles a functional query definition into a multi-phase parallel job. The first phase of the job is typically constrained to execute on machines that contain partitions of the cached input data set, while the remaining phases (which read data shuffled over the network) can execute anywhere. The Spark frontend passes a list of task descriptions (and any associated placement constraints) for the first phase of the job to a Sparrow scheduler, which assigns the tasks to slaves. Because Sparrow schedulers are lightweight, we run a scheduler alongside each Spark frontend to ensure minimum scheduling latency. When the Sparrow scheduler assigns a task to a slave, it passes the task description provided by the Spark frontend to the Spark executor running on the slave, which uses the task description to launch the task. When one phase completes, Spark requests scheduling of the tasks in the subsequent phase.

6.3 Fault Tolerance

Because Sparrow schedules from multiple schedulers, the design is inherently fault tolerant: if a scheduler becomes unavailable, an application can direct its requests to one of many alternate schedulers. When initializing a Sparrow client, applications pass a list of schedulers to connect to. When a scheduler fails, the Sparrow clients using it re-connect to the next available scheduler and trigger a callback at the application. This approach lets frameworks decide how to handle tasks which were in-flight during the scheduler failure. In some cases, they may want to simply ignore failed tasks. Other frameworks might want to re-launch them or determine whether they have run once already. Spark's approach to failure tolerance is discussed further in §7.3.

7 Experimental Evaluation

We evaluate Sparrow by performing a variety of experiments on 110 node 16-core EC2 clusters. 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 exact overhead that Sparrow incurs and quantify its performance in comparison with an optimal scheduler. Second, 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

To evaluate Sparrow against a realistic workload, we measure Sparrow's performance scheduling TPC-H queries. 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.

We use Shark [11], a large scale data analytics platform built on top of Spark, to execute the TPC-H benchmark queries. Shark lets users load working sets into memory on several parallel

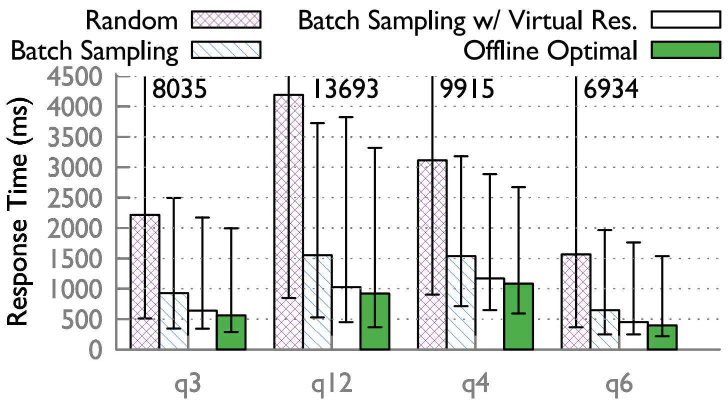


Figure 8: Median, 5th, and 95th percentile response times for TPC-H queries with several placement strategies.

machines and analyze them using a SQL-based query language. Our experiment features several users concurrently using Shark to run ad-hoc analytical queries in a 100-node cluster. The data set and queries are drawn from the TPC-H OLAP database benchmark.

The query corpus features four TPC-H queries, with 2, 3, 3, and 5 phases respectively. The queries operate on a denormalized in-memory copy of the TPC-H input dataset (scale factor 2.5). Each frontend queries a distinct data-set that is striped over 90 of the 100 machines: this consists of 30 three-way replicated partitions. Each Shark query breaks down into multiple phases. For example, for query 3, the first phase scans and filters records, the second aggregates data into groups (via a shuffle) and the third sorts by a group-related value. These phases are each scheduled through Sparrow's `submitRequest` RPC.

In the experiment, each of 10 users launches random permutations of the TPC-H queries continuously for 30 minutes. The queries are launched at an aggregate rate of approximately 30 per second in order to keep the cluster 65% utilized, on average. Queries are launched through 10 Sparrow schedulers, which use a sample ratio of 2 to assign tasks to worker machines.

Shark queries are compiled into multiple Spark [30] stages that each trigger a scheduling request to Sparrow. The response time of a Shark query is dictated by the accumulated response time of each sub-phase. The duration of each phase is not uniform, instead varying from a few tens of milliseconds to several hundred. The phases in this workload have heterogeneous numbers of tasks, corresponding to the amount of data processed in that phase. The queries require a mix of constrained and unconstrained scheduling requests.

Figure 8 plots the median, 5th, and 95th percentile response times for each TPC-H query across all frontends during a 2 minute sampled period in the middle of the experiment. Approximately 4000 queries are completed during this window, resulting in around 12TB of in-memory table scans (the first phase of each query scans the entire table). The figure compares response time using Sparrow to response time with three alternatives. The first alternative, random placement, places each task on a randomly selected machine. The second strategy implements batch sampling

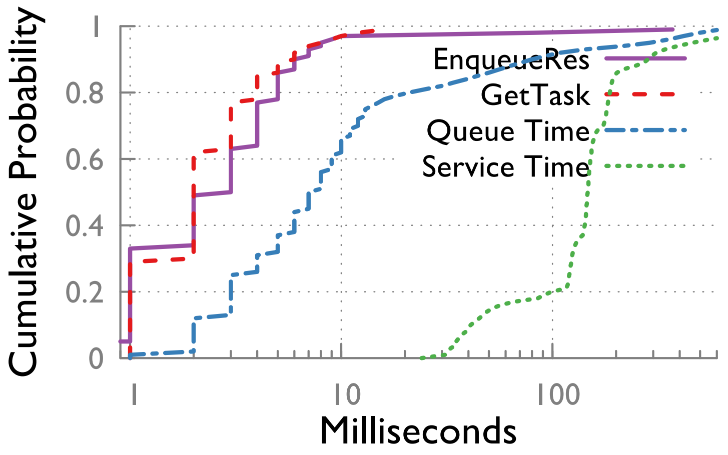


Figure 9: Latency distributions for each stage in the Sparrow scheduling algorithm. Note that control plane messages are much shorter than task durations.

and uses queue-length estimations. The third group represents the full Sparrow algorithm, which includes virtual reservations.

As shown in Figure 8, Sparrow improves the median query response time by 2-3×, and the 95th by up to 130× compared to naïve randomized techniques. In addition to improving on existing techniques, Sparrow also provides good absolute performance: response times using Sparrow are within 14% of an offline optimal. To determine the offline optimal, we take each query and calculates the response time of the query if all tasks were launched immediately; that is, with zero wait time. This offline optimal calculation is conservative—it does not take into account task launch delay or queuing that is unavoidable due to small utilization bursts—both of which are inevitable even with an omniscient scheduler. Even so, Sparrow performs within 14% of the offline optimal.

7.2 Scheduling Latency

Figure 9 deconstructs Sparrow scheduling latency from the proceeding TPC-H experiment into four components. These describe a complete trace of 21,651 requests. Four lines are plotted, each characterizing the latency of a particular phase of the Sparrow scheduling algorithm. These phases are depicted in Figure 7. `EnqueueRes` plots the time it takes for the scheduler to enqueue a given task reservation on a slave node. `GetTask` describes the time it takes for the slave to fetch and run a new task. This includes time spent asking for "null" tasks which were already launched elsewhere. `ServiceTime` and `QueueTime` reflect the amount of time individual tasks spent running or queued on the slave nodes. The messaging in Sparrow is fast, requiring only a handful of milliseconds to enqueue and dequeue individual tasks. The majority of this time is spent shipping the task information itself, a component of `GetTask`, which is an unavoidable overhead. Sparrow achieves good performance because it keeps queue time low relative to service time. The long enqueue reservation 99th percentile latency does not affect performance: workers that take a long time to receive or response to a reservation are simply not assigned a task.

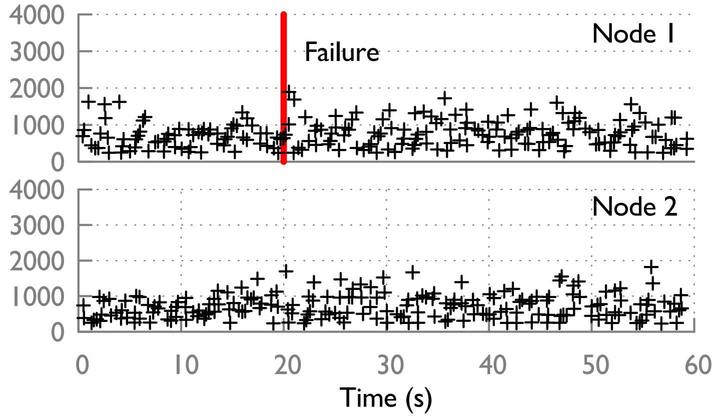


Figure 10: TPC-H response times for two frontends submitting queries to a 100-node cluster. Node 1 suffers from a scheduler failure at 20 seconds.

7.3 How do scheduler failures impact job response time?

Sparrow is resilient to scheduler failure, providing automatic client failover between schedulers. Figure 10 plots the response time for ongoing TPC-H queries in an experiment parameterized exactly as in §7.1. This experiment features 10 Shark frontends which submit continuous queries. The frontends are each initialized with a list of several Sparrow schedulers and initially connect to the scheduler resident on the same node. At time `t=20`, we terminate the Sparrow scheduler on Node 1. The frontend is programmed to fail over to a scheduler on Node 2, so we plot that node as well.

The speed of failover recovery makes the interruption almost unnoticeable. Because Sparrow schedulers maintain no session state, the process of failing-over reduces to timing out and connecting to another node. The Sparrow client library detects failures with a fixed 100ms timeout. On failure, it triggers an application-layer failure handler which has the option of resubmitting in-flight tasks. In the case of Spark, that handler instantly re-launches any phases which were in-flight. In this experiment, detecting the failure took 100ms, connecting to a new scheduler took less than 5ms, and re-launching outstanding tasks took less than 15ms. Only a small number of queries (2) have tasks in flight during the failure; these queries suffer some overhead.

7.4 Synthetic Workload

The remaining sections evaluate Sparrow using a synthetic workload composed of 10 100ms tasks, unless otherwise specified. Each task runs floating-point multiplications to fully utilize a single CPU for 100ms. In this workload, ideal job completion time is always 100ms, 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.

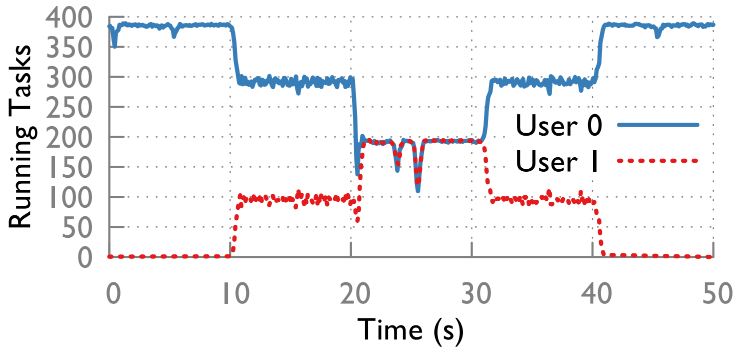


Figure 11: Cluster share used by two users that are each assigned equal shares of the cluster. User 0 submits at a rate to utilize the entire cluster for the entire experiment while user 1 adjusts its submission rate each 10 seconds. Sparrow assigns both users their max-min fair share.

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

Figure 11 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 1'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. Nonethess, as shown in Figure 11, 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 fluctionations from the correct fair share.

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

Table 3 demonstrates that Sparrow provides response times with 40% of optimal for a high priority user in the presence of a misbehaving low priority user. 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,

HP Load	LP Load	HP Response Time	LP Response Time
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)	40202 (46191)
0.25	1.75	141 (226)	254896 (269661)

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.

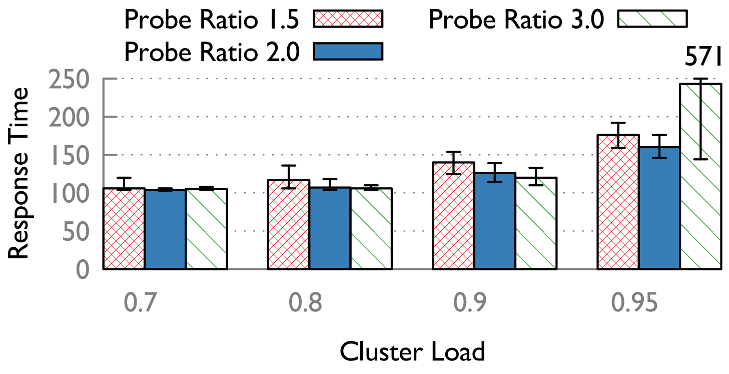


Figure 12: Effect of probe ratio on median response time. Error bars depict 5th and 95th percentile performance.

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, the high priority experiences 40% worse response times when sharing with a demanding low priority user than when running alone on the cluster. Because Sparrow does not use preemption, high priority tasks may need to wait to be launched until low priority tasks (that are running when the higher priority task arrives) 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.

7.7 Probe Ratio Sensitivity Analysis

This section seeks to answer two questions: first, if Sparrow reduces messaging by using a probe ratio less than 2, how does it perform; and second, how much can a larger probe ratio improves Sparrow's performance? Figure 12 illustrates response time at increasing load in a system running our synthetic workload. We use 4-core nodes for this experiment to emphasize the difference between probe ratios; with larger numbers of cores, fewer probes are necessary. A higher probe ratio decreases tail behavior at loads of at most 90%, but at 95% load, using a probe ratio of 3 leads

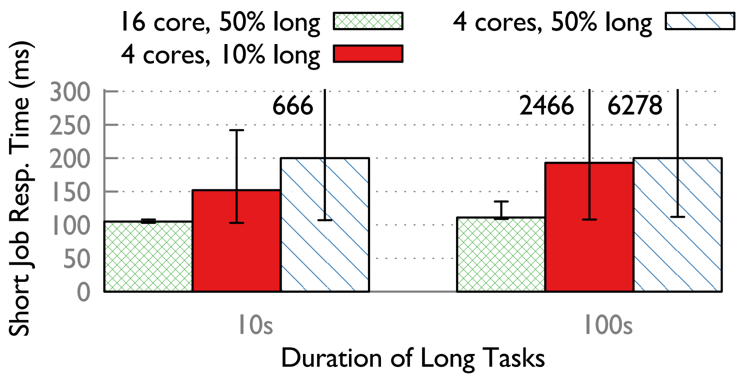


Figure 13: Sparrow provides low response time for jobs composed of 10 100ms tasks, even when those tasks are run alongside much longer jobs.

to increased response times due to the increased messaging. A lower probe ratio of 1.5 provides nearly as good performance as a probe ratio of 2; systems where messaging is expensive might choose to use a lower probe ratio.

7.8 Breaking Sparrow with Heterogeneity

We hypothesized that heterogeneous task distributions were the weak point in Sparrow's design: if some workers were running long tasks, we expected Sparrow's approach to have increasing difficulty finding idle machines on which to run tasks. We found that 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 13 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's provides response times for short tasks within 11% of optimal (100ms) when running on 16-core machines, even when 50% of tasks are 3 orders of magnitude longer. Short tasks see more significant performance degredation in a 4-core environment.

7.9 Scaling to Large Clusters

We use simulation to evaluate Sparrow's performance in larger clusters. Figure 3 demonstrated that Sparrow continues to provide good performance in a 10,000 node cluster.

8 Practical Challenges for Low-Latency Workloads

This paper focuses on optimizing scheduling for low-latency parallel workloads. In removing the scheduling bottleneck, we discovered numerous other barriers to supporting low latency; eliminating such bottlenecks will be important for future frameworks that seek to provide low latency. This section outlines three such issues and discusses how we addressed them in our implementation and evaluation.

Overhead of control-plane messages Messaging can impose significant overhead when tasks become very short. In particular, launching a task on a worker node requires shipping the relevant task information to that node (which occurs in response to the `getTask()` RPC). Long-lived executor processes running on the node help reduce the size of this information, because the application need not ship binaries or other information shared across all jobs. Nonetheless, the application must send information about the task, including information about input data, a partial query plan, function closures, or other meta data. If a node is launching a job composed of hundreds of tasks, task descriptions must be small enough to ensure that even the last task to ship doesn't experience appreciable delay. We found that 100-200Kb task descriptions impacted performance of Shark jobs on a 100-node cluster, and as a result, invested effort to compress tasks down to approximately 5Kb. Even with 5Kb task descriptions, shipping all of the tasks for a large job requires significant messaging at the scheduler. The overhead of this messaging presents another compelling reason to distribute scheduling (and associated messaging) over many schedulers.

Skew in data partitioning If partitions of a dataset are unevenly distributed, hot-spots develop that can significantly slow response time. Such data skew can hurt performance, even in the face of very efficient scheduling. For the experiments in this paper, we leveraged two features of Shark to alleviate hot-spots. First, we replicated each partition in-memory on multiple machines, which allows the underlying scheduler more choice in placing tasks. Second, we tightly controlled data-layout of each table to ensure that partitions were balanced.

Virtualized environments Many modern clusters use CPU virtualization to multiplex workloads between users and improve efficiency. Unfortunately, virtualization can inflate network round trip times to as much as 30ms, as discussed in Bobtail [28]. Because Sparrow node monitors are idle while requesting new tasks from the scheduler, 30ms network RTTs significantly degrade performance, particularly at high system load. To mitigate this problem, all experiments in this paper were run on EC2 high memory cluster compute instances with 16 cores (cr1.xlarge) to minimize the presence of swapping. We found that even medium to large size EC2 nodes (e.g., quad-core m2.2xlarge instances) introduced unacceptable virtualization-induced network delay.

9 Limitations and Future Work

To handle the latency and throughput demands of low-latency frameworks, our approach sacrifices some of the features sometimes 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 over-subscribed, batch sampling supports aggregate fair-sharing or priority-based scheduling. Sparrow's distributed setting lends itself to *approximated* policy enforcement in order to minimize system complexity. Bounding this inaccuracy under arbitrary workloads is a focus of future work.

With regards to job-level policies, 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 Applications that need to run m inter-communicating tasks require gang scheduling. Gang scheduling is typically implemented using bin-packing algorithms which search for and reserve time slots in which an entire job can run. Because Sparrow queues tasks on several machines, there is no central point from which to perform bin-packing. While it is often the case that Sparrow places all jobs on entirely idle machines, this is not guaranteed. Applications that use Sparrow could implement inter-task communication by using a map-reduce model, where data is shuffled and stored (in memory or on disk) before being joined. This trade-off is also made by several other cluster schedulers [12, 19, 13]).

Cancellation Schedulers could reduce the amount of time slaves spend idle due to virtual reservations by canceling outstanding reservations once all tasks in a job have been scheduled. Cancellation would decrease idle time on slaves with little cost.

10 Related Work

Scheduling in large distributed systems has been extensively studied in earlier work.

HPC Schedulers The high performance computing (HPC) community has produced a broad variety of schedulers for cluster management. HPC jobs tend to be monolithic, rather than composed of fine-grained tasks, obviating the need for high throughput schedulers. HPC schedulers thus optimize for large jobs with complex constraints, using constrained bin-packing algorithms. High throughput HPC schedulers, such as SLURM [14], target maximum throughput in the tens to hundreds of scheduling decisions per second.

Condor The Condor scheduler [24] targets high throughput computing environments using a combination of centralization and distribution. Condor's scheduling throughput is again in the regime of 10 to 100 jobs per second [6]. This is the result of several complex features, including a rich constraint language, a job checkpointing feature, and support for gang scheduling, all of which result in a heavy-weight matchmaking process. Unlike Sparrow, Condor is a general purpose cluster resource manager.

Fine-Grained Schedulers Quincy [13] provides fairness and locality by mapping the scheduling problem onto a graph and using a solver to compute the optimal online schedule. Because the size of the graph is proportional to the number of slaves, scheduling latency grows with cluster size. In a 2500 node cluster, the graph solver takes over a second to compute a scheduling assignment [13]; while multiple jobs can be batched in a single scheduling assignment, waiting seconds to schedule a job that can complete in hundreds of milliseconds is unacceptable overhead.

Dremel [16] 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 query's and its storage layout – it is closely tied to the underlying architecture of the system.

Cluster Resource Managers Cluster resource managers federate resource usage between multiple users and applications running in a cluster, typically using a centralized design. These resource managers target coarse grained resource allocation and are not designed to handle fine-grained task assignment.

YARN [19] extends the original Hadoop Map Reduce scheduler [25] to support multiple frameworks by introducing distributed per-job application masters. YARN relies on periodic (1s, by default) heartbeats from slaves to determine when resources have become available. To avoid wasted resources, heartbeats need to occur more frequently than the expected task turnover rate on a machine; to handle a multi-core machines running sub-second tasks, each slave would need to send hundreds of heartbeats per second, which would easily overwhelm the resource manager. Furthermore, high availability has not yet been implemented for YARN.

Mesos [12] imposes minimal scheduling overhead by delegating all aspects of scheduling other than fairness to framework schedulers and employs batching to handle high throughput. Still, Mesos was not designed to handle short tasks, so batches scheduling requests to provide high throughput. This batching introduces scheduling delay on the order of seconds.

A variety of other schedulers, e.g., Omega [22], target course-grained scheduling, scheduling dedicated resources for services that handle their own request-level scheduling. Batch sampling instead targets fine-grained scheduling, which allows high utilization by sharing resources across frameworks; we envision that batch sampling may be used to schedule a static subset of cluster resources allocated by a general scheduler like Omega.

Straggler Mitigation Straggler mitigation techniques such as task speculation [4, 31, 5, 8] deal with the nondeterministic variation in task execution time (rather than task wait time) due to unpredictable causes (e.g., resource contention and failures). These techniques are orthogonal (and complementary) to Sparrow's distributed scheduling technique, and could be implemented in Sparrow or by applications running on top of Sparrow.

Load Balancing A variety of projects explore load balancing tasks in multi-processor shared-memory architectures [21, 7, 27, 10]. In such systems, processes are dynamically scheduled amongst an array of distributed processors. Scheduling is necessary each time a process is swapped out, leading to a high aggregate volume of scheduling decisions. These projects 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, parallel machine with memory access uniformity. As a result, the majority of effort is spend determining when to *re*schedule processes to balance load.

Theoretical Work While a huge body of existing work analyzes the performance of the power of two choices load balancing technique, as summarized by Mitzenmacher [18], to the best of our knowledge, no existing work explores performance for parallel jobs. Many existing analyses, including the work that inspired batch sampling [20], examine a setting where balls are assigned to bins, and analyze how many balls fall into the most loaded bin. 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.

Parallel Disk Request Scheduling Scheduling parallel tasks has been explored in the context of parallel disk IO in RAID disk arrays [26].

11 Conclusion

This paper presented Sparrow, a stateless decentralized scheduler that provides near optimal performance using two key techniques: batch sampling and pull-based scheduling. Using mathematical analysis, we demonstrated Sparrow's provable performance. We used a realistic TPC-H workload to demonstrate that Sparrow provides response times within 14% of an optimal scheduler, and used a synthetic workload running on a deployed cluster to demonstrate that Sparrow is fault tolerant, provides aggregate fair shares, can enforce priorities, and is resilient to different probe ratios and distributions of task durations. In light of these results, we assert that distributed scheduling using Sparrow presents a viable alternative to centralized schedulers for low latency parallel workloads.

References

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

[2] The Hadapt Adaptive Analytic Platform. `http://hadapt.com`.

[3] VMware vSphere. `http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html`.

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

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

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

[7] Casavant, T. L., and Kuhl, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Software Eng. 14* (February 1988), 141–154.

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

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

[10] Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. Softw. Eng.* (1986).

[11] Engle, C., Lupher, A., Xin, R., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: Fast Data Analysis Using Coarse-Grained Distributed Memory. In *Proc. SIGMOD* (2012).

[12] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform For Fine-Grained Resource Sharing in the Data Center. In *NSDI* (2011).

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

[14] Jette, M. A., Yoo, A. B., and Grondona, M. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[15] KORNACKER, M., AND ERICKSON, J. 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.

[16] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* (2010).

[17] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* (2001).

[18] MITZENMACHER, M. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*, S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, Eds., vol. 1. Springer, 2001, pp. 255–312.

[19] MURTHY, A. C. The Next Generation of Apache MapReduce. `http://tinyurl.com/7deh64l`, February 2012.

[20] PARK, G. A Generalization of Multiple Choice Balls-into-Bins: Tight Bounds. *CoRR abs/1201.3310* (2012).

[21] RUDOLPH, L., SLIVKIN-ALLALOUF, M., AND UPFAL, E. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proc. ACM SPAA* (1991).

[22] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys* (2013).

[23] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. OSDI* (2012).

[24] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation : Practice and Experience 17*, 2-4 (Feb. 2005), 323–356.

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

[26] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.* (1996).

[27] WILLEBEEK-LEMAIR, M., AND REEVES, A. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems 4* (1993), 979–993.

[28] XU, Y., MUSGRAVE, Z., NOBLE, B., AND BAILEY, M. Bobtail: Avoiding Long Tails in the Cloud. In *Proc. NSDI* (2013).

[29] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay Scheduling: A Simple Technique For Achieving Locality and Fairness in Cluster Scheduling. In *Proc. EuroSys* (2010).

[30] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI* (2012).

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