Research in Operating Systems Sparrow

Sparrow: Distributed, Low Latency Scheduling

*Slides partially based on Ousternout’s presentation*
(Some) useful concepts picked up from OS

- Scheduler, preemption, shortest-job first, RPC, gang-scheduling, fair-share scheduling, …

  *(In no particular order)*
Jobs and scheduling for data analytics

- Large data analytics clusters
- Running ever shorter and higher-fanout jobs
- What for? Finance, language translation, highly personalize search, …
Jobs and scheduling for data analytics

- Jobs composed of short tasks
- Produced from frameworks that stripe work across $1^3$ machines (e.g., Dremel, Spark, …)
- Targeting task running in $\sim100$ms
Job latencies decreasing rapidly

2004: MapReduce batch job
2009: Hive query
2010: Dremel query
2010: In-memory Spark query
2012: Impala query

10 min. 10 sec. 100 msec. 1 msec.
Scheduling challenges

- Millisecond latency
- Quality placement
- Fault tolerant
- High throughput
- *Fixing centralized schedulers is not an option*
Sparrow schedules tasks in clusters

- using a decentralized, randomized approach
- support constraints and fair sharing and
- provides response times within 12% of ideal

Adapts power-of-two-choices to parallel task scheduling, introducing three techniques
  - Batch sampling
  - Late binding
  - Policies and constrains
Per-task sampling

- A direct application of power of two choices
- For each task in a job, the scheduler
  - Randomly selects two workers
  - Probes each worker (a lightweight RPC) for queue length
  - Places task in shortest queue
Per-task Sampling
Per-task sampling
Per-task sampling

- A direct application of power of two choices
- For each task in a job, the scheduler
  - Randomly selects two workers
  - Probes each worker (a lightweight RPC) for queue length
  - Places task in shortest queue
- For comparison
  - Random
  - Omniscient – greedy, based on complete information
Better than random, >2x worst than opt

Simulated Results
100-task jobs in 10,000-node cluster, exp. task durations

Higher load, harder to find a free worker where to place a task

100-task jobs in 10,000-node cluster, exp. task duration
Response time grows with tasks/jobs

![Graph showing response time growth with tasks/job. The x-axis represents tasks/job (1, 10, 100, 1000) and the y-axis represents response time (ms). The graph indicates that response time increases significantly with increasing tasks/job. The text also mentions a 70% cluster load.]
Per-task sampling

An unlucky draw

And a lucky one

Job is done when all tasks are done ...
Batch sampling

Place $m$ tasks on the least loaded $d*m$ workers
Batch sampling

4 probes (d=2)
Per-task and Batch Sampling

- **Per-task**
- **Batch**

![Bar chart showing response time (ms) for different numbers of tasks/job (1, 10, 100, 1000) under 70% cluster load.]

- **70% cluster load**
Batch sampling – better, still 1.92x opt
Late binding

- Sample-based scheduling performs poorly under load
  - Select based on queue length, coarse predictor of wait time
    - Better predictors using estimates of task durations is hard
  - Race condition – multiple schedulers picking the same worker
Late binding

- Late binding
  - Workers put task internal work queue, hold back response
  - When task gets to the front, worker replies RPC
  - Scheduler assigns job’s tasks to first $m$ workers
  - ... and sends no-ops to the rest (proactive)

- Cost
  - Idle while sending RPC – a 2% efficiency loss
  - Fraction of time idle while requesting tasks
    $\frac{(d \times \text{RTT})}{(t + d \times \text{RTT})}$
Late binding benefits

Simulated Results

100-task jobs in 10,000-node cluster, exp. task durations

Within 5% of Opt

0.55x of batch sampling

Response Time (ms)

Load

Random
Per-Task
Batch
Batch+Late Binding
Omniscient
Handling constrains and policies

- **Job constraints**, e.g., *run in workers with GPU*
  - Trivially handled, pick $dm$ candidate from the subset

- **Per-task constraints**, e.g., *run where input is*
  - Use per-task sampling, improved with
  - ... sharing information across tasks when possible
  - ... use late-binding
Handling constrains and policies

- Resource allocation policies, under load
  - Strict priorities through multiple queues on workers
    - FIFO, earliest deadline first, shortest job first, class, …
    - Trade accuracy for simplicity (no global information but low priority jobs can run first)
    - No preemption
  - Weighted fair sharing
    - One queue per user
Sparrow implementation

- Sparrow works for 1+ concurrent frameworks
  - Front end + executor (long-lived processes responsible for executing task w/o startup overhead)
  - Node monitor federates resources usage bet/ co-located frameworks

Scheduler service:
submitRequest() / taskComplete()
Sparrow and Spark’s native scheduler

How does Sparrow compare to Spark’s native scheduler?

All tasks in a job are of same duration; ideal job response time is duration of a task.

Spark’s centralized scheduler cannot keep up.

Spark native scheduler

Sparrow

Ideal

Small 100 16-core EC2, 10 tasks/job, 10 schedulers, 80% load (synthetic workload)
TPC-H queries: background

- A common benchmark for analytics workloads; representative of ad-hoc queries on business data
- Shark queries compiled into multiple spark stages
- Each stage triggers a scheduling request using submitRequest()
- Task in first stage constrained to machines holding input data
- Stages have different number of tasks, durations and un/constrained queries

**Shark**: SQL execution engine

**Spark**: Distributed in-memory analytics framework

**Sparrow**
TPC-H Queries

100 16-core EC2, 10 tasks/job, 10 schedulers, 80% load

Within 12% of ideal

Random
Per-task binding
Batch sampling
Batch + late binding

Response Time (ms)

q3
q4
q6
q12

4217 (med.)
5396 (med.)
7881 (med.)
Effect of probe ratio

Too low, can’t find lightly loaded machines
Too high, pay the cost of increased messaging

Response Time (ms)

9279 (95th)
574 (med.), 4169 (95th)
678 (med.), 4212 (95th)

80% load

90% load

Probe Ratio

Random

Improvement with small amount

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.

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 or-}

can’t find lightly loaded machines
Too high, pay the cost of increased messaging

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.

ters of magnitude longer than the short tasks. We ran as eri so fe xperiment sw i t hei fo bj e s: s h o r t 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 degradation 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 over-

subscribed, 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 is an area of future work.

13
Failure impact

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.

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. 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 as in 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 clustering

Some of these factors could be mitigated, but at the expense of code readability and understandability. A clustering
Limitations/future work

- Scheduling policies – can they do better than approximate?
- Inter-job constraints (e.g., *tasks of job A cannot run with those of B*) – hard to do w/o drastic changes
- Gang scheduling – no central point where to do it
- Query-level policies – easy to extend, FIFO
- ...
- *Want to try?* [http://github.com/radlab/sparrow](http://github.com/radlab/sparrow)