Sparrow schedules tasks in clusters using a decentralized, randomized approach.
Sparrow schedules tasks in clusters using a decentralized, randomized approach to support constraints and fair sharing.
Job Latencies Rapidly Decreasing

2004: MapReduce batch job
2009: Hive query
2010: Dremel Query
2010: Impala query
2010: In-memory Spark query
2012: Impala query
2013: Spark streaming

10 min.
10 sec.
100 ms
1 ms

Job Latencies Rapidly Decreasing
Scheduling challenges:
Millisecond Latency
Quality Placement
Fault Tolerant
High Throughput
2004: MapReduce batch job

2009: Hive query

2010: Dremel Query

2010: Impala query

2010: In-memory Spark query

2012: Impala query

2012: Spark streaming

2013: Spark streaming

Scheduling throughput

26 decisions/second

1.6 K decisions/second

160 K decisions/second

16 M decisions/second

1000 16-core machines

Scheduler throughput

10 min.

10 sec.

100 ms

1 ms
Today:
Completely Centralized

Sparrow:
Completely Decentralized

Millisecond Latency ✓

Quality Placement ✓

Fault Tolerant ✓

High Throughput ✓
Sparrow

Decentralized approach
Existing randomized approaches
Batch Sampling
Late Binding
Analytical performance evaluation

Handling constraints

Fairness and policy enforcement

Within 12% of ideal on 100 machines
Scheduling with Sparrow
Simulated Results

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

Omniscient: infinitely fast centralized scheduler

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

Power of Two Choices
Per-task sampling

Power of Two Choices
Simulated Results

100-task jobs in 10,000-node cluster, exp. task durations
Response Time Grows with Tasks/Job!

70% cluster load
Per-Task Sampling

Job → Scheduler → Task 1

Task 2

Worke

r

✓

✓
Place $m$ tasks on the least loaded of $d \cdot m$ slaves.
Per-task versus Batch Sampling

70% cluster load
Simulated Results

100-task jobs in 10,000-node cluster, exp. task durations
Queue length poor predictor of wait time

155 ms
530 ms

Poor performance on heterogeneous workloads
Late Binding

Place $m$ tasks on the least loaded of $d \cdot m$ slaves
Late Binding

Place $m$ tasks on the least loaded of $d \cdot m$ slaves
Late Binding

Job

Scheduler

Scheduler

Scheduler

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Place $m$ tasks on the least loaded of $d \cdot m$ slaves
Simulated Results

100-task jobs in 10,000-node cluster, exp. task durations
What about constraints?
Restrict probed machines to those that satisfy the constraint.
Probe separately for each task
Technique Recap

Batch sampling + Late binding + Constraint handling
How does Sparrow perform on a real cluster?
Spark on Sparrow

Query: DAG of Stages

Scheduler

Job

Workers
Spark on Sparrow

Query: DAG of Stages

Sparrow Scheduler

Jobs

Workers
Spark on Sparrow

Query: DAG of Stages

Sparrow Scheduler

Job

Workers
How does Sparrow compare to Spark’s native scheduler?

100 16-core EC2 nodes, 10 tasks/job, 10 schedulers, 80% load
TPC-H Queries: Background

TPC-H: Common benchmark for analytics workloads

**Shark**: SQL execution engine

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

**Sparrow**
TPC-H Queries

100 16-core EC2 nodes, 10 schedulers, 80% load

100 16-core EC2 nodes, 10 schedulers, 80% load
TPC-H Queries

100 16-core EC2 nodes, 10 schedulers, 80% load

Within 12% of ideal
Median queuing delay of 9ms
Fault Tolerance

Timeout: 100ms
Failover: 5ms
Re-launch queries: 15ms
When does Sparrow not work as well?

High cluster load
Related Work

Centralized task schedulers: e.g., Quincy

Two level schedulers: e.g., YARN, Mesos

Coarse-grained cluster schedulers: e.g., Omega

Load balancing: single task
Sparrows provides near-ideal job response times without global visibility
Backup Slides
Can we do better without losing simplicity?

Policy Enforcement

Priorities
Serve queues based on strict priorities

High Priority
Low Priority

Fair Shares
Serve queues using weighted fair queuing

User A (75%)
User B (25%)

Can we do better without losing simplicity?