Firmament

Fast, centralized cluster scheduling at scale

Ionel Gog Malte Schwarzkopf Adam Gleave
Robert N. M. Watson Steven Hand

UNIVERSITY OF CAMBRIDGE  MIT Google
Meet Sesame, Inc.

- Sesame’s employees run:

  1. Interactive data analytics that must complete in seconds

  2. Long-running services that must provide high performance

  3. Batch processing jobs that increase resource utilization
The cluster scheduler must achieve:

1. Good task placements
   - high utilization without interference

2. Low task scheduling latency
   - support interactive tasks
   - no idle resources
State of the art

**Good task placements**

**Centralized**
Sophisticated algorithms
[Borg, Quincy, Quasar]

**Distributed**
Simple heuristics
[Sparrow, Tarcil, Yaq-d]

**Low scheduling latency**

**Hybrid**
Split workload, provide either
[Mercury, Hawk, Eagle]

Can’t get both good placements and low latency for the entire workload!
Firmament provides a solution!

- Centralized architecture
- Good task placements
- Low task scheduling latency
- Scales to 10,000+ machines
Finds **optimal** task placements

Min-cost **flow-based** centralized scheduler
Min-cost flow scheduler

Preference for first rack

Me too!

Rack 1

Rack 2
Min-cost flow scheduler

Schedules all tasks at the same time
Considers tasks for migration or preemption
Globally optimal placement!

Min-cost flow scheduler

Rack 1

Rack 2

Interactive
Batch
Service
Introduction to min-cost flow scheduling

Tasks $T_0, T_1, T_2, T_3, T_4, T_5$

Machines $M_0, M_1, M_2, M_3, M_4, M_5$
Introduction to min-cost flow scheduling

Flow scheduling: tasks to machines

$T_0$ $T_1$ $T_2$ $T_3$ $T_4$ $T_5$ $M_0$ $M_1$ $M_2$ $M_3$ $M_4$ $M_5$
Introduction to min-cost flow scheduling

Flow scheduling: zoom in

Cost: 3

Cost: 5
Introduction to min-cost flow scheduling

Min-cost flow places tasks with minimum overall cost
Introduction to min-cost flow scheduling

Flow supply

Flow demand: 0
How well does the Quincy approach scale?
Simulated Quincy using Google trace, 50% utilization
Simulated Quincy using Google trace, 50% utilization 66 sec on average.

Too slow! 30% of tasks wait to be scheduled for over 33% of their runtime and waste resources.
Goal: sub-second scheduling latency in common case

better

Avg. scheduling latency [log10]

Goal

Cost scaling

Cluster size [machines]
Contributions

- **Low task scheduling latency**
  - Uses best suited min-cost flow algorithm
  - Incrementally recomputes the solution

- **Good task placement**
  - Same optimal placements as Quincy
  - Customizable scheduling policies
Scheduling policy
class QuincyPolicy {
   
   Cost_t TaskToResourceNodeCost(
      TaskID_t task_id) {
      return task_unscheduled_time * 
      quincy_wait_time_factor;
   }
   ...
}

Specifying scheduling policies

N.B: More details in the paper.
Scheduling policy

*Defines graph*

Flow graph
Scheduling policy

Defines graph

Flow graph

Submits graph

Min-cost, max-flow solver
Scheduling policy

Defines graph

Flow graph

Submits graph

Min-cost max-flow solver

Most time spent here
### Algorithm

<table>
<thead>
<tr>
<th>Cost scaling</th>
<th>Worst-case complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(V^2E \log(VC))$</td>
<td></td>
</tr>
</tbody>
</table>

**Used by Quincy**

- $E$: number of arcs
- $V$: number of nodes
- $U$: largest arc capacity
- $C$: largest cost value

$E > V > C \approx U$
Subsampled Google trace, 50% slot utilization [Quincy policy]

Cost scaling is too slow beyond 1,000 machines
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost scaling</td>
<td>$O(V^2 E \log(VC))$</td>
</tr>
<tr>
<td>Successive shortest path</td>
<td>$O(V^2 U \log(V))$</td>
</tr>
</tbody>
</table>

Lower worst-case complexity

$E$: number of arcs  
$V$: number of nodes  
$U$: largest arc capacity  
$C$: largest cost value  

$E > V > C \approx U$
Successive shortest path only scales to ~100 machines.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost scaling</td>
<td>$O(V^2E\log(VC))$</td>
</tr>
<tr>
<td>Successive shortest path</td>
<td>$O(V^2U\log(V))$</td>
</tr>
<tr>
<td>Relaxation</td>
<td>$O(E^3CU^2)$</td>
</tr>
</tbody>
</table>

$E$: number of arcs  
$V$: number of nodes  
$U$: largest arc capacity  
$C$: largest cost value  

$E > V > C \approx U$  

Highest complexity
Subsampled Google trace, 50% slot utilization [Quincy policy]

Relaxation meets our sub-second latency goal
Why is Relaxation fast?

Single-ish pass flow push

Relaxation is well-suited to the graph structure
Relaxation suffers in pathological edge cases

Machine utilization:
- High
- Medium

Capacity: 1 task
Relaxation suffers in pathological edge cases

- $T_0$, $T_1$, $T_2$, $T_3$, $T_4$ are tasks.
- $M_0$, $M_1$, $M_2$, $M_3$ are machines.
- $S$ is the sink.

- Machine utilization: high, medium.
- Capacity: 1 task.
Relaxation suffers in pathological edge cases

Relaxation cannot push flow in a single pass any more
How bad does Relaxation’s edge case get?

Experimental setup:
- Simulated 12,500 machine cluster
- Used the Quincy scheduling policy
- Utilization >90% to oversubscribed cluster
Quincy, 12,500 machines cluster, job of increasing size
Relaxation’s runtime increases with utilization.
Cost scaling is unaffected by high utilization.

Quincy, 12,500 machines cluster, job of increasing size.

Cost scaling is faster.

Cost scaling is unaffected by high utilization.
Scheduling policy

Flow graph

Min-cost, max-flow solver

Input graph

Relaxation

Cost scaling
Quincy, 12,500 machines cluster, job of increasing size

Algorithm runtime is still high at utilization > 94%
Min-cost max-flow solver

Input graph

Flow graph

Scheduling policy

Min-cost, max-flow solver

Graph scaling

Cost scaling

Graph state

State discarded

Output flow
Graph changes

Min-cost max-flow solver

Previous graph state

Cost scaling

Graph state

Output flow

Scheduling policy

Flow graph

Min-cost, max-flow solver
Quincy, 12,500 machines cluster, job of increasing size

Incremental cost scaling is ~2x faster
Evaluation

Does Firmament choose good placements with low latency?

Note: many additional experiments in the paper.
How do Firmament’s placements compare to other schedulers?

Experimental setup:
- Homogeneous 40-machine cluster, 10G network
- Mixed batch/service/interactive workload
Network utilization: low  medium  high
Firmament chooses good placements

5 seconds task response time on idle cluster
Firmament chooses good placements

20% of tasks experience poor placement

Sparrow is unaware of resource utilization
20% of tasks experience poor placement

Centralized Kubernetes and Docker still suffer

better
Firmament chooses good placements

Avoided co-location interference

Firmament outperforms centralized and distributed schedulers
How well does Firmament handle challenging workloads?

Experimental setup:
• Simulated 12,500 machine Google cluster
• Used the centralized Quincy scheduling policy
• Utilization varies between 75% and 95%
Firmament handles challenging workloads at low latency

Simulate interactive workloads by scaling down task runtimes

Median task runtime: 420s
Median task runtime: 1.7s
Firmament handles challenging workloads at low latency

![Graph showing task placement latency](image)

- 45 seconds average latency (tuned over Quincy setup’s 66s)

- Average latency is too high even without many short tasks
Firmament handles challenging workloads at low latency.

Latency with a 250x acceleration:
- 75th percentile: 2 sec
- Maximum: 57 sec
Firmament handles challenging workloads at low latency.

Firmament’s common-case latency is sub-second even at 250x acceleration.
Conclusions

- Low task scheduling latency
  - Uses best algorithm at all times
  - Incrementally recomputes solution
- Good task placement
  - Same optimal placements as Quincy
  - Customizable scheduling policies

Open-source and available at: firmament.io