Practical Considerations for Multi-Level Schedulers

Benjamin Hindman – @benh
agenda

① multi-level scheduling (scheduler activations)
② intra-process multi-level scheduling (Lithe)
③ distributed multi-level scheduling (Mesos)
  ① optimistic multi-level scheduling
  ② Dominant Resource Fairness
  ③ reservations and revocable resources
  ④ persistent volumes
  ⑤ inverse offers
④ conclusion
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers

④ conclusion
multi-level scheduling

N:1 “user-level” threading

user-level thread scheduler maps $N$ user-level threads onto 1 kernel-level thread (i.e., virtual processor)
N:1 threading

+ fast context switch times
+ fast user space synchronization primitives
- unnecessary blocking (i.e., on I/O) even when other user-level threads are runnable
multi-level scheduling

1:1 threading

each user-level thread maps onto a kernel-level thread (i.e., virtual processor)
1:1 threading

+ blocking thread doesn’t hold up others
- expensive context switches
- poorly timed preemption within user space synchronization primitives
1:1 threading

+ blocking thread doesn’t hold up others

- expensive context switches

- poorly timed preemption within user space synchronization primitives

  kernel space synchronization primitives (e.g., wait queues)
1:1 threading

+ blocking thread doesn’t hold up others

- expensive context switches

- poorly timed preemption within user space
  synchronization primitives

  └→ kernel space synchronization
      primitives (e.g., wait queues)

  └→ expensive synchronization
1:1 threading

+ blocking thread doesn’t hold up others

- expensive context switches

- poorly timed preemption within user space

synchronization primitives

\[\leftarrow\text{kernel space synchronization primitives (e.g., wait queues)}\]

\[\leftarrow\text{expensive synchronization}\]

\[\leftarrow\text{hybrid (Linux’s futex)}\]
N:1 + 1:1

best of both worlds?

N:1 fast context switches

N:1 fast user space synchronization

1:1 no unnecessary blocking
scheduler activations

best of both worlds!

N:1 fast context switches

N:1 fast user space synchronization

1:1 no unnecessary blocking
multi-level scheduling

N:M threading

user-level thread scheduler maps $N$ user-level threads onto $M$ kernel-level threads (i.e., virtual processor)
scheduler activations

kernel-level scheduler *communicates* with user-level thread library via “upcalls” (i.e., events):

1) a virtual processor has been allocated or deallocated from you

2) this activation (i.e., context) has blocked or unblocked
the bigger picture

kernel-level scheduler has inadequate knowledge of application’s execution needs/semantics to make optimal decisions

user-level scheduler execution needs/semantics can’t easily or efficiently be expressed to kernel-level scheduler
the bigger picture

mechanisms that facilitate the exchange of information between *levels of abstractions* are a good thing

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes ...
the bigger picture

... another example: LRU page replacement algorithm

what if we could just ask the application which pages it is no longer using!?
N:M threading today

Linux provides 1:1 threading

N:M threading abandoned in NetBSD and FreeBSD for 1:1 threading
multi-level scheduling

user-level “tasks” (e.g., Intel’s Threading Building Blocks)

user-level task scheduler maps $N$ user-level tasks onto $M$ kernel-level threads (i.e., virtual processor)
multi-level scheduling

user-level "tasks" (e.g., Intel's Threading Building Blocks)

user-level task scheduler maps $N$ user-level tasks onto $M$ kernel-level threads (i.e., virtual processor)

N:M without the primitives!
N:M threading today

Windows 7 introduced User Mode Scheduling (UMS) which provides the “upcall” mechanism introduced with scheduler activations (but only for blocking/unblocking, not for allocation/deallocation of processors)
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers

④ conclusion
today’s reality:

N:M × P:Q × R:S
intra-process
multi-level
scheduling
intra-process multi-level

example: sparse QR factorization (SPQR)
intra-process multi-level

example: sparse QR factorization (SPQR)
intra-process multi-level

example: sparse QR factorization (SPQR)
intra-process multi-level

example: sparse QR factorization (SPQR)
intra-process multi-level

decision: sparse QR factorization (SPQR)

functional composition of parallel libraries!
intra-process multi-level

eexample: sparse QR factorization (SPQR)
intra-process multi-level

eample: sparse QR factorization (SPQR)
intra-process multi-level

how many CPUs (threads) to allocate between libraries?
intra-process multi-level

be careful not to oversubscribe resources!
SPQR software architecture
SPQR software architecture

each TBB tasks creates M threads per MKL (OpenMP) invocation

oversubscription!
Using Intel MKL with Threaded Applications


- If more than one thread calls Intel MKL and the function being called is threaded, it is important that threading in Intel MKL be turned off. Set **OMP_NUM_THREADS=1** in the environment.
Recommended settings for calling Intel MKL routines from multi-threaded applications

MKL\_NUM\_THREADS=N

Enable MKL threading - use when you are sure that there are enough resources (physical cores) for MKL threading in addition to your own threads. Choose N carefully.

**Example 1:**

application has 2 threads, each thread calls MKL and the system has 8 cores: it's reasonable to set MKL\_NUM\_THREADS=4.

**Example 2:**

MKL function is called from a critical section of a parallel region - set MKL\_NUM\_THREADS=N, where N is the number of physical cores in the system (or use mkl\_set\_num\_thread( N) routine).

**NOTE:**
set additional options when the application is based on OpenMP* threads.
Recommended settings for calling Intel MKL routines from multi-threaded applications

MKL_NUM_THREADS=N

Enable MKL threading - use when you are sure that there are enough resources (physical cores) for MKL threading in addition to your own threads. **Choose N carefully.**

**Example 1:**

application has 2 threads, each thread calls MKL and the system has 8 cores: it's reasonable to set MKL_NUM_THREADS=4.

**Example 2:**

MKL function is called from a critical section of a parallel region - set MKL_NUM_THREADS=N, where N is the number of physical cores in the system (or use mkl_set_num_thread(N) routine).

**NOTE:**
set additional options when the application is based on OpenMP* threads.
Recommended settings for calling Intel MKL routines from multi-threaded applications

MKL_NUM_THREADS=N

Enable MKL threading - use when you are sure that there are enough resources (physical cores) for MKL threading in addition to your own threads. Choose N carefully.

**Example 1:**

application has 2 threads, each thread calls MKL and the system has 8 cores: it’s reasonable to set MKL_NUM_THREADS=4.

**Example 2:**

MKL function is called from a critical section of a parallel region - set MKL_NUM_THREADS=N, where N is the number of physical cores in the system (or use mkl_set_num_thread(N) routine).

**NOTE:**
set additional options when the application is based on OpenMP* threads.
recommendation: static partitioning
static partitioning

fix the number of CPUs (threads) allocated between libraries
optimal allocation?
recommendation:
manual tuning
static partitioning

Runtime (in seconds) for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)
static partitioning

- default: TBB=16 • OMP=16
- manually tuned: TBB=11 • OMP=8

Time (sec)

<table>
<thead>
<tr>
<th>Input Matrix</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>landmark</td>
<td>2.5</td>
</tr>
<tr>
<td>deltaX</td>
<td>3.1</td>
</tr>
<tr>
<td>ESOC</td>
<td>7.5</td>
</tr>
<tr>
<td>Rucci</td>
<td>27.5</td>
</tr>
</tbody>
</table>

Input matrix

TBB=11 • OMP=8
TBB=3 • OMP=5
TBB=16 • OMP=5
TBB=16 • OMP=8
can’t we do better?
dynamic needs

Give resources to OpenMP

Give resources to TBB

manual tuning means a fixed TBB/OMP thread allocation throughout execution
static partitioning is scheduling
static partitioning is scheduling; but sub-optimal (scheduler/human is missing information)
Lithe:
multi-level scheduling between libraries
Lithe
Lithe
library interfaces remain the same

zero lines of high-level codes changed (SPQR, MKL)

add a scheduler to each library (the Lithe version of the library)

just link in Lithe runtime + Lithe versions of libraries (TBB, OpenMP)
Lithe “upcall” interface

- **TBB**
  - enter
  - yield
  - request
  - register
  - unregister

- **MKL**
  - enter
  - yield
  - request
  - register
  - unregister

- **OpenMP**
  - enter
  - yield
  - request
  - register
  - unregister

- **Lithe**
  - task
Lithe “upcall” interface
Lithe
dynamically allocate CPUs (threads) between TBB and OpenMP

... matching demand as closely as possible
Runtime (in seconds) for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)
Lithe

- Default
  - TBB=16 • OMP=16
- Manually tuned
  - TBB=11 • OMP=8
  - TBB=3 • OMP=5
  - TBB=16 • OMP=5
- Lithe
  - TBB=16 • OMP=8

input matrix
L2 cache misses for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)
Number of context switches for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)
the bigger picture

*explicit* transfer of control is *implicit* transfer of resources!

you “schedule” some computation when you call into another library, so you should account for its resource usage!

*Lithe: make transfer of resources explicit via library schedulers*
agenda

① multi-level scheduling (scheduler activations)
② intra-process multi-level scheduling (Lithe)
③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers
④ conclusion
Lithe $\rightarrow$ Mesos

Lithe: sharing a machines resources between libraries via multi-level scheduling

Mesos: sharing a datacenters resources between distributed systems via multi-level scheduling
static partitioning
status quo in
datacenters
static partitioning considered harmful
static partitioning considered harmful
static partitioning considered harmful
static partitioning considered harmful
static partitioning considered harmful
static partitioning considered harmful

higher utilization!
static partitioning considered harmful
static partitioning considered harmful
what if ...
we built something to \textit{manage} resources in the datacenter like Lithe manages resources on a machine
we built something to manage resources in the datacenter like scheduler activations manages resources on a machine
Apache Mesos: cluster manager based on multi-level scheduling
Mesos is a cluster manager with a master/slave architecture
Mesos challenged the status quo of cluster managers
cluster manager status quo

The specification includes as much information as possible to assist the cluster manager in scheduling and execution.
cluster manager status quo

application

wait for task to be executed

cluster manager
cluster manager status quo
problems with specifications

① hard to specify certain desires or constraints

② hard to update specifications dynamically as tasks execute and finish/fail
MapReduce specification?

Input:
- the quick brown fox
- the fox ate the mouse
- how now brown cow

Map:
- the, 1
- brown, 1
- fox, 1

Shuffle & Sort:
- the, 1
- fox, 1
- the, 1
- how, 1
- now, 1
- brown, 1

Reduce:
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3
- quick, 1

Output:
- ate, 1
- cow, 1
- mouse, 1
- quick, 1
- cow, 1
MapReduce specification?

course-grained

fine-grained
MapReduce specification?

course-grained

fine-grained

best resource utilization, but hard if impossible to specify
MapReduce specification?

course-grained  fine-grained

worst resource utilization, but easy to express (how most cluster managers run something like Hadoop)
multi-level scheduling

motivation:
scheduler has *inadequate knowledge* of application/framework’s execution needs/semantics to make optimal decisions
multi-level scheduling

motivation:

scheduler has *inadequate knowledge* of application/framework’s execution needs/semantics to make optimal decisions

application/framework specific knowledge of execution needs/semantics *can’t easily be expressed* to scheduler
frameworks register with the Mesos master(s) in order to run jobs/tasks
Mesos model

A request is purposely simplified subset of a specification including just the required resources at that point in time.
question: what should you do if you can’t satisfy a request?
question: what should you do if you can’t satisfy a request?

① wait until you can …
question: what should you do if you can’t satisfy a request?

① wait until you can ...

② offer best you can immediately
question: what should you do if you can’t satisfy a request?

① wait until you can ...

② offer best you can immediately
Mesos model

framework

offer
hostname
4 CPUs
4 GB RAM

masters
Mesos model

framework

offer
hostname
4 CPUs
4 GB RAM

masters
Mesos model

framework uses the offers to perform its own *scheduling*
Mesos model

framework

\[
\text{task} \\
3 \text{ CPUs} \\
2 \text{ GB RAM}
\]

framework uses the offers to perform it’s own *scheduling*
Mesos model

The framework uses the offers to perform its own *scheduling*

*multi-level scheduling*
1\textsuperscript{st} level: master makes allocations via offers

2\textsuperscript{nd} level: frameworks schedule tasks using offers
an analogue: non-blocking sockets

```
write(s, buffer, size);
```
an analogue: non-blocking sockets

application

kernel

42 of 100 bytes written!
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers

④ conclusion
offers represent the current *snapshot* of available resources a framework can use
multiple offers outstanding

framework1

offer
foo.bar.com
4 CPUs
4 GB RAM

framework2

offer
baz.bar.com
2 CPUs
1 GB RAM

masters
question: should resources within offers be disjoint?
question: should multiple frameworks be able to compete for available resources?
concurrency control

pessimistic

optimistic

offers made to different frameworks are *disjoint*
concurrency control

pessimistic  optimistic

all offers overlap with one another, thus causing frameworks to "compete" first-come-first-served
optimistic offers

- framework1
  - offer
  - foo.bar.com
  - 4 CPUs
  - 4 GB RAM

- framework2
  - offer
  - foo.bar.com
  - 4 CPUs
  - 4 GB RAM

- masters
isomorphism:
Google Omega’s snapshots and Mesos’ optimistic offers
Omega and Mesos

- framework
- snapshot
- database

- framework
- masters
- offer
  - hostname
  - 4 CPUs
  - 4 GB RAM
Omega and Mesos

- framework
- database
- transaction

- framework
- masters
- task
  - 3 CPUs
  - 2 GB RAM
Google’s Omega: multi-level scheduler optimistic by default
Mesos: multi-level scheduler pessimistic by default; optimistic via optimistic offers
concurrency control

pessimistic → optimistic

some offers overlap with one another, thus causing only some of the frameworks to “compete” first-come-first-served
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
  ① optimistic multi-level scheduling
  ② Dominant Resource Fairness
  ③ reservations and revocable resources
  ④ persistent volumes
  ⑤ inverse offers

④ conclusion
Mesos 1\textsuperscript{st}-level scheduler allocates resources to frameworks using a fair-sharing algorithm we created called Dominant Resource Fairness (DRF)
DRF, born of static partitioning
static partitioning across teams

team  \{  promotions  \quad \text{trends} \quad \text{recommendations}  \}
static partitioning across teams

team

promotions

- - - - - -

- - - - - -

trends

- - - - - -

- - - - - -

recommendations

- - - - - -

- - - - - -

fairly shared!
goal: fairly share the resources
without static partitioning
partition utilizations

team

promotions

45% CPU
100% RAM

trends

75% CPU
100% RAM

recommendations

100% CPU
50% RAM
observation: a dominant resource bottlenecks each team from running any more jobs/tasks
dominant resource bottlenecks

team
promotionstrendsrecommendations
utilization

bottleneck

RAM

45% CPU
100% RAM

75% CPU
100% RAM

100% CPU
50% RAM

CPU
insight: allocating a *fair share* of each team’s dominant resource guarantees they can run *at least* as many jobs/tasks as with static partitioning!
... if my team gets at least $1/N$ of my dominant resource I will do no worse than if I had my own cluster, but I might do better when resources are available!
DRF in Mesos

framework

① Frameworks specify a *role* when they register (i.e., the team to charge for used resources)

masters
DRF in Mesos

① frameworks specify a role when they register (i.e., the team to charge for used resources)

② master calculates each role’s dominant resource (dynamically) and allocates appropriately
step 4: Profit (statistical multiplexing)
in practice, *fair* sharing is insufficient
weighted fair sharing

team

promotions
trends
recommendations
weighted fair sharing

team

promotions

weight

0.17

trends

0.5

recommendations

0.33
Mesos implements weighted DRF

Masters can be configured with weights *per role*

Resource allocation decisions incorporate the weights to determine dominant fair shares
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers

④ conclusion
in practice, *weighted fair sharing* is still insufficient
a non-cooperative framework (i.e., has long tasks or is buggy) can get hoard too many resources
resource reservations

resources on individual slaves can be reserved for particular roles

resource offers include the reservation role (if any)
reservations

slave

reservations

mesos-slave

role-foo

role-bar
static reservations

reservations available with Mesos using the --resources flag on each mesos-slave

```
$ mesos-slave --resources='cpus(role-foo):2;mem(role-foo):1024;cpus(role-bar):2;mem(role-bar):1024;cpus(*):4;mem(*):4096'
```
static reservations

+ strong guarantees
- set up by an operator when starting slave
- immutable (must drain/restart the slave)
dynamic reservations

framework scheduler reserves resources at runtime when it accepts an offer (allocation)
**dynamic reservations**

framework scheduler reserves resources at runtime when it accepts an offer (allocation)

```
framework
(role-baz)
```

```
Accept
Launch/
Reserve(4
CPUs, 4
GB RAM)
```

```
masters
```

```
slave
(2)
```
dynamic reservations

framework scheduler reserves resources at runtime when it accepts an offer (allocation)

framework (role-baz)

masters

slave

Launch/Reserve(4 CPUs, 4 GB RAM)
reservations provide guarantees, but at the cost of utilization
revocable resources

framework *(promotions)*

offer
hostname
4 CPUs
4 GB RAM
role: *trends*

resources that are reserved for another role and thus may be *revoked* at any time
preemption via revocation

... my tasks will not be killed unless I’m using revocable resources!
optimistic offers of revocable resources

framework1 (trends)

offer
hostname
4 CPUs
4 GB RAM
role: trends

framework2 (promotions)

masters
optimistic offers of revocable resources

framework1 (trends)
- offer hostname
- 4 CPUs
- 4 GB RAM
- role: trends

framework2 (promotions)
- offer hostname
- 4 CPUs
- 4 GB RAM
- role: trends

masters
oversubscription via revocable resources

oversubscribe resources by allocating unused resources as revocable!

framework (promotions)

offer hostname
4 CPUs
4 GB RAM
role: trends

masters
agenda

① multi-level scheduling (scheduler activations)
② intra-process multi-level scheduling (Lithe)
③ distributed multi-level scheduling (Mesos)
  ① optimistic multi-level scheduling
  ② Dominant Resource Fairness
  ③ reservations and revocable resources
  ④ persistent volumes
  ⑤ inverse offers
④ conclusion
persistent volumes

framework scheduler *creates volumes* for disk resources at runtime when it *accepts* an offer (allocation)

scheduler (role-baz)

Offer hostname
4 CPUs
4 GB RAM

master

slave

(1)
persistent volumes

framework scheduler *creates volumes* for disk resources at runtime when it *accepts* an offer (allocation)

scheduler (role-baz)

Accept Launch/Create(1 TB DISK)

masters

(2)

slave
persistent volumes

framework scheduler reserves resources at runtime when it accepts an offer (allocation)
persistent volumes

volumes are created before launching any tasks or executors

slave

mesos-slave

role-baz

role-bar

role-foo
persistent volumes

volumes are *mounted* into the container when a task or executor gets launched
persistent volumes

volumes persist even after task or executor terminate!
persistent volumes

volumes persist even after task or executor terminate!
agenda

① multi-level scheduling (scheduler activations)
② intra-process multi-level scheduling (Lithe)
③ distributed multi-level scheduling (Mesos)
  ① optimistic multi-level scheduling
  ② Dominant Resource Fairness
  ③ reservations and revocable resources
  ④ persistent volumes
  ⑤ inverse offers
④ conclusion
revocation “guarantee”

... my tasks will not be killed unless I’m using revocable resources!
revocation “guarantee”

what about when:

① want/need to defrag (think page replacement algorithm!)

② maintenance
mechanism for deallocation

possible solution: introduce failures to deallocate resources

but why not communicate explicitly!?
inverse offer

framework scheduler gets deallocation requests in the form of inverse offers
inverse offer

framework scheduler can *kill tasks* and *acknowledge deallocation*
maintenance

① drain machine by sending out inverse offers

② while draining can still send out offers with revocable resources

③ remove machine from allocation once drained (or at some specific time)
the bigger picture

reservations +
persistent volumes +
inverse offers =
long-lived stateful frameworks!
Mesos: level of abstraction
Mesos: level of abstraction

Mesos (master)

- Framework

Mesos (slaves)

- Framework

Mesos (master) is responsible for allocation (and reallocation) of resources.
Mesos: level of abstraction

Mesos (master)

<table>
<thead>
<tr>
<th>framework</th>
<th>framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesos (slaves)</td>
<td></td>
</tr>
</tbody>
</table>

responsible for allocation (and reallocation) of resources
Mesos: level of abstraction

Mesos (master)

framework

responsible for allocation (and reallocation) of resources

framework

Mesos (slaves)
Mesos: level of abstraction

Mesos (master)

framework

framework

responsible for allocation (and reallocation) of resources

Mesos (slaves)
Mesos: level of abstraction

Mesos (master)

framework

responsible for allocation (and reallocation) of resources

Mesos (slaves)
Mesos: level of abstraction

+ enable running multiple distributed systems on the same cluster of machines and dynamically share the resources more efficiently!
Mesos: level of abstraction

+ provide common functionality every new distributed system re-implements
Mesos: datacenter kernel

provides common functionality every new distributed system re-implements:

• failure detection
• package distribution
• task starting
• resource isolation
• resource monitoring
• task killing, cleanup
• ...

today	

tomorrow
Mesos: datacenter kernel provides common functionality every new distributed system *re-implements*:

- failure detection
- package distribution
- task starting
- resource isolation
- resource monitoring
- task killing, cleanup
- ...

*don’t reinvent the wheel!*
scheduler activations

kernel-level scheduler *communicates* with user-level thread library via “upcalls” (i.e., events):

1) a virtual processor has been allocated or deallocated from you

2) this activation (i.e., context) has blocked or unblocked
Mesos

1st-level scheduler *communicates* with framework-level scheduler via “upcalls” (i.e., events):

1) resources have been allocated or deallocated from you

2) this task has completed, been preempted, etc

3) …
built on Mesos:

- Spark
- Apache Aurora
- Marathon/
- Chronos
ported to Mesos:
agenda

① multi-level scheduling (scheduler activations)

② intra-process multi-level scheduling (Lithe)

③ distributed multi-level scheduling (Mesos)
   ① optimistic multi-level scheduling
   ② Dominant Resource Fairness
   ③ reservations and revocable resources
   ④ persistent volumes
   ⑤ inverse offers

④ conclusion
more to Mesos

- isolation via control groups and namespaces
- fault-tolerant architecture
- scalability via state pushed to leaves (scheduler and slaves)
Mesos in production
the datacenter computer needs an operating system
build a distributed system in one organization and easily deploy it in another!
the bigger picture

mechanisms that facilitate the exchange of information between *levels of abstractions* are a good thing

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes ...
the bigger picture

mechanisms that facilitate the exchange of information between levels of abstractions managers are a good thing

abstractions managers exist for good reasons, but without sufficient communication they force sub-optimal outcomes ...
surge in cluster management interest!
Docker!
• Google’s Kubernetes
  • Not a multi-level scheduler.

• Docker’s Swarm
  • Not a multi-level scheduler.

• Amazon’s ECS
  • Multi-level scheduler, completely optimistic.

• YARN
  • Sort of multi-level, based on requests, no notion of optimistic allocation, reservations, deallocation, etc.
Thank You!

Composing parallel software efficiently with Lithe

In Proc. of the SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)

Mesos: a platform for fine-grained resource sharing in the data center

In Proc. of the USENIX 2011 conference on Networked Systems Design and Implementation (NSDI)