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

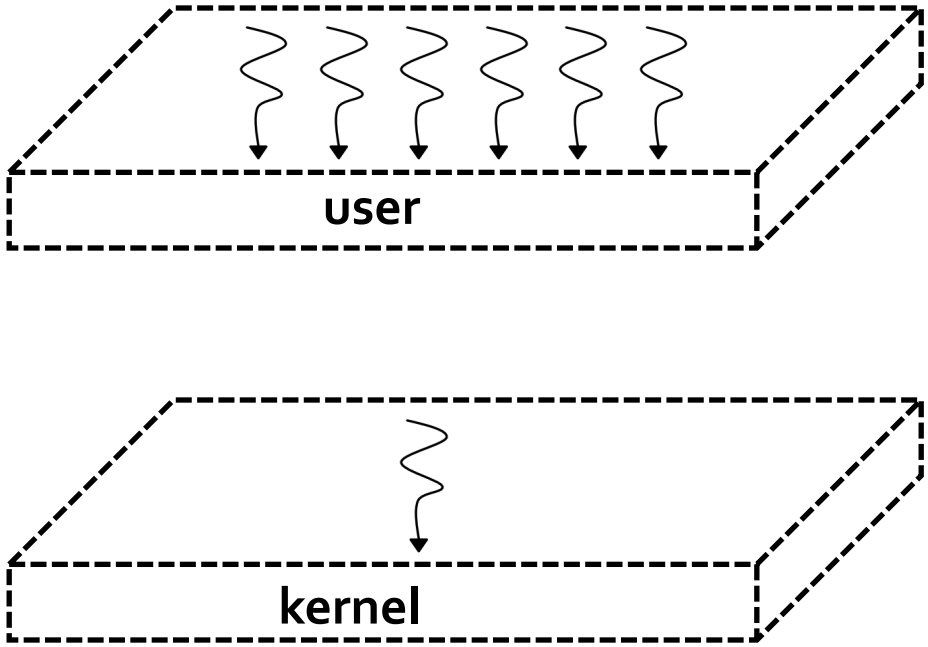
agenda

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

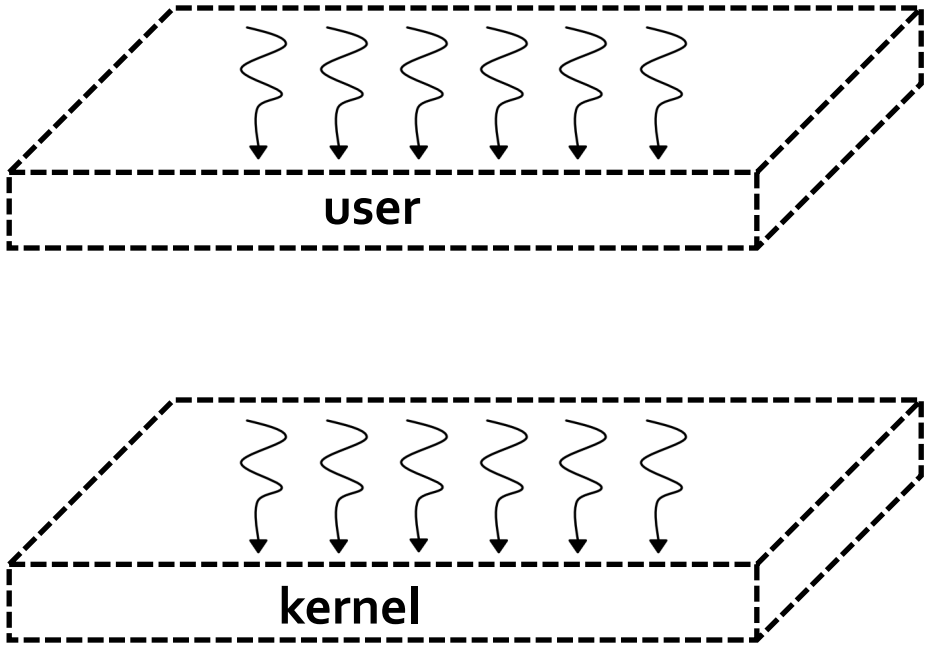
+ fast user space synchronization primitives

- 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
  - ↳ kernel space synchronization primitives (e.g., wait queues)
    - ↳ expensive synchronization
      - ↳ 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 ✓

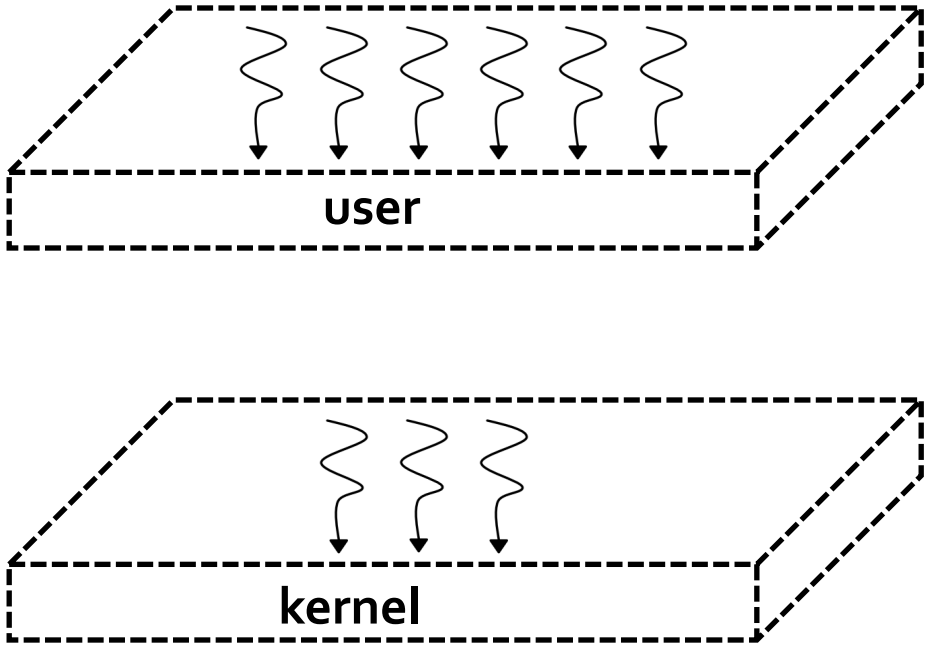
N:1 fast user space synchronization ✓

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!?

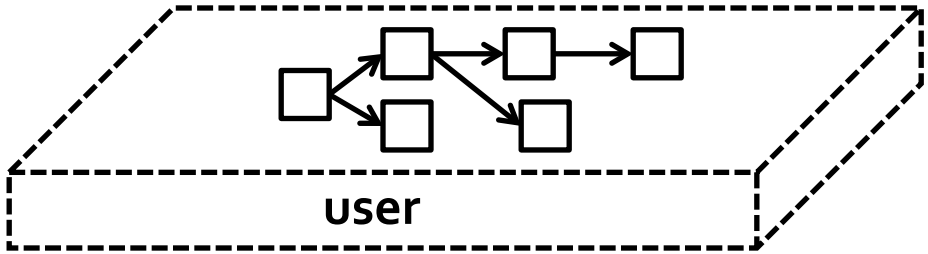
N:M threading today

Linux provides 1:1 threading

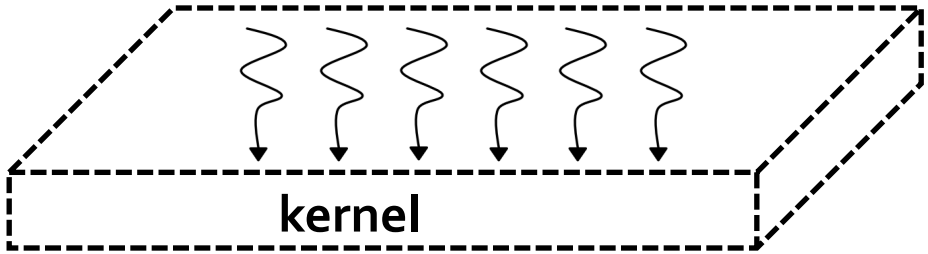
N:M threading abandoned in NetBSD and FreeBSD for 1:1 threading

multi-level scheduling

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

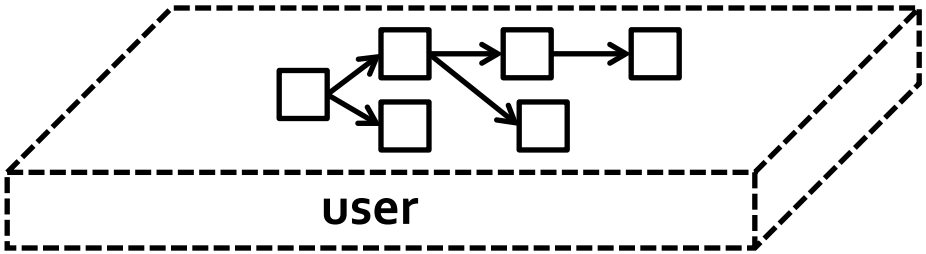


user-level task scheduler maps *N* user-level tasks onto M kernel-level threads (i.e., virtual processor)

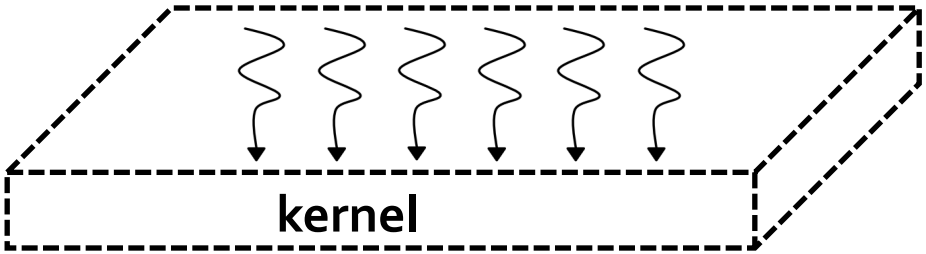


multi-level scheduling

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:

$N{:}M \times P{:}Q \times R{:}S$

intra-process multi-level scheduling

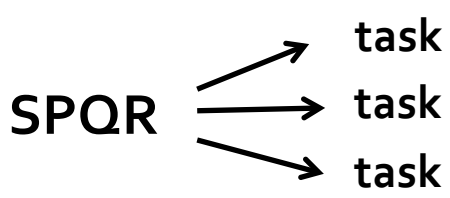
intra-process multi-level

example: sparse QR factorization (SPQR)

SPQR

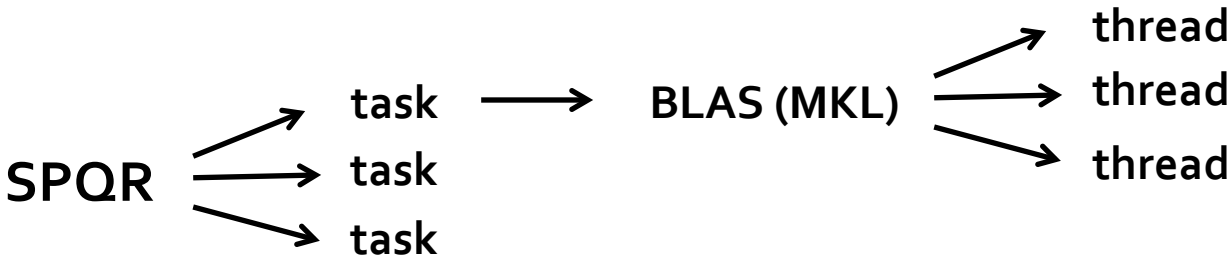
intra-process multi-level

example: sparse QR factorization (SPQR)



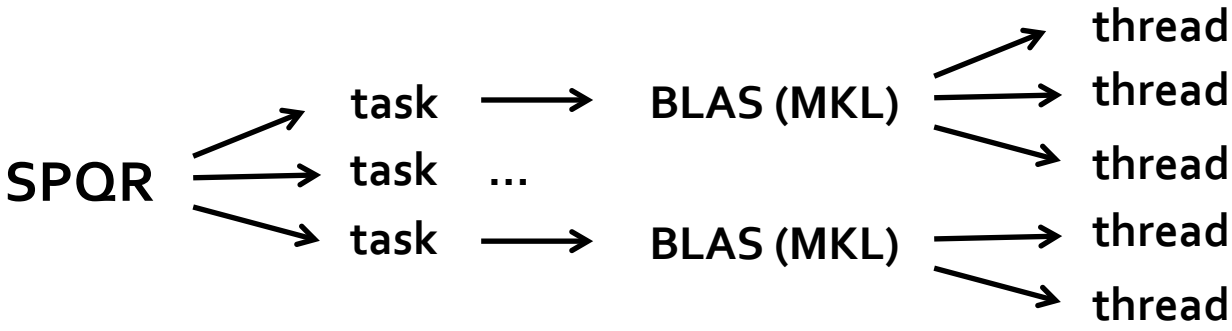
intra-process multi-level

example: sparse QR factorization (SPQR)



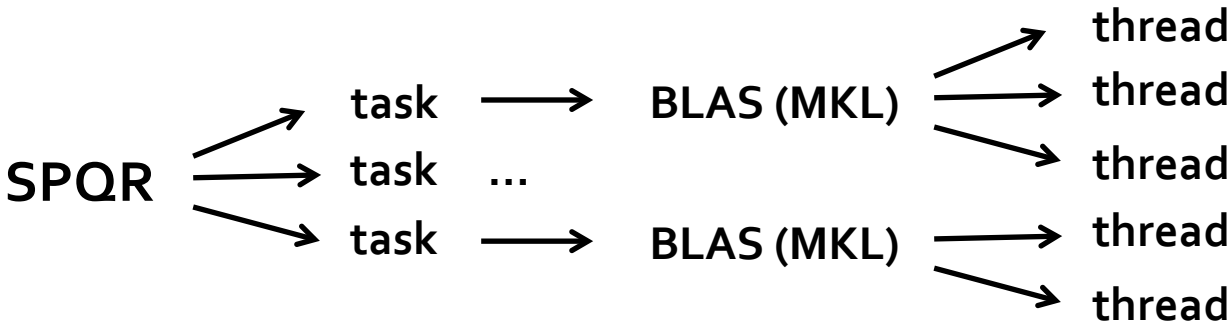
intra-process multi-level

example: sparse QR factorization (SPQR)



intra-process multi-level

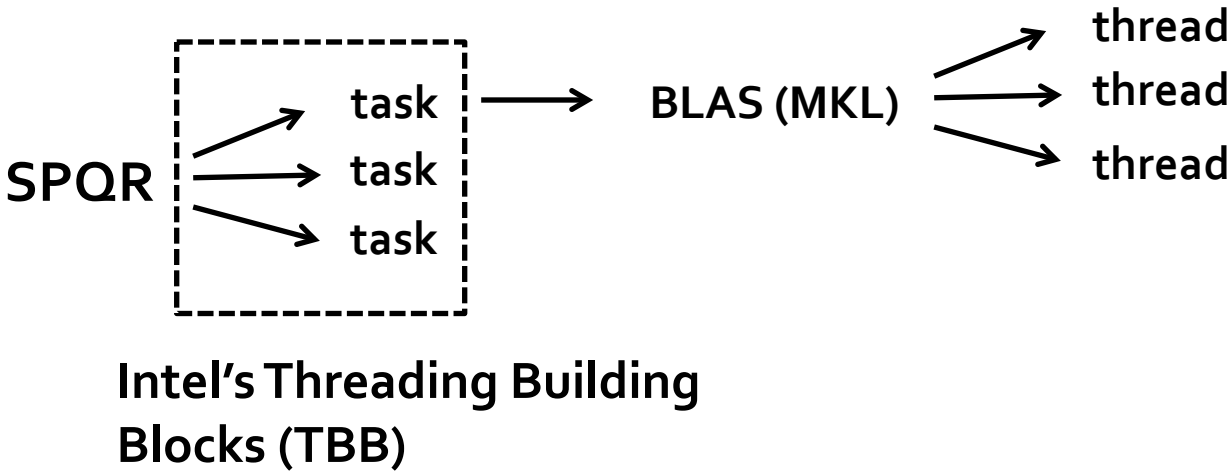
example: sparse QR factorization (SPQR)



functional composition of parallel libraries!

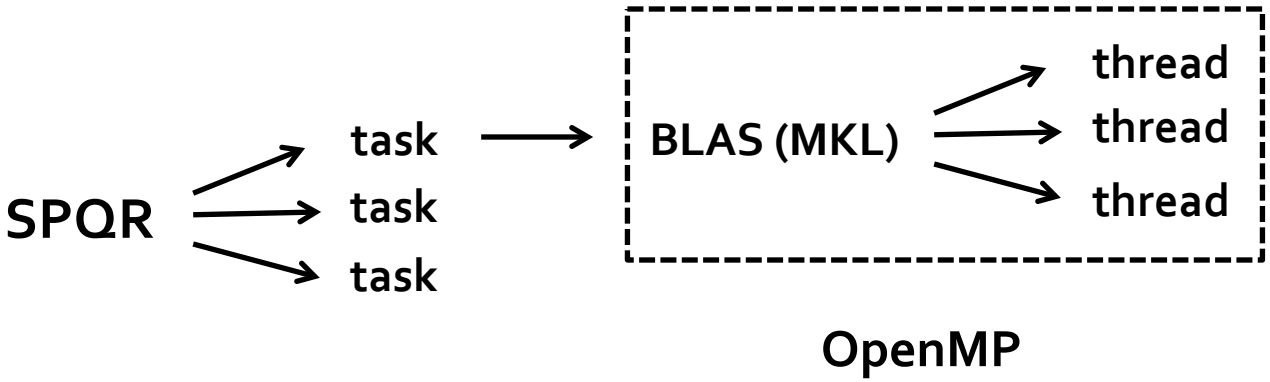
intra-process multi-level

example: sparse QR factorization (SPQR)



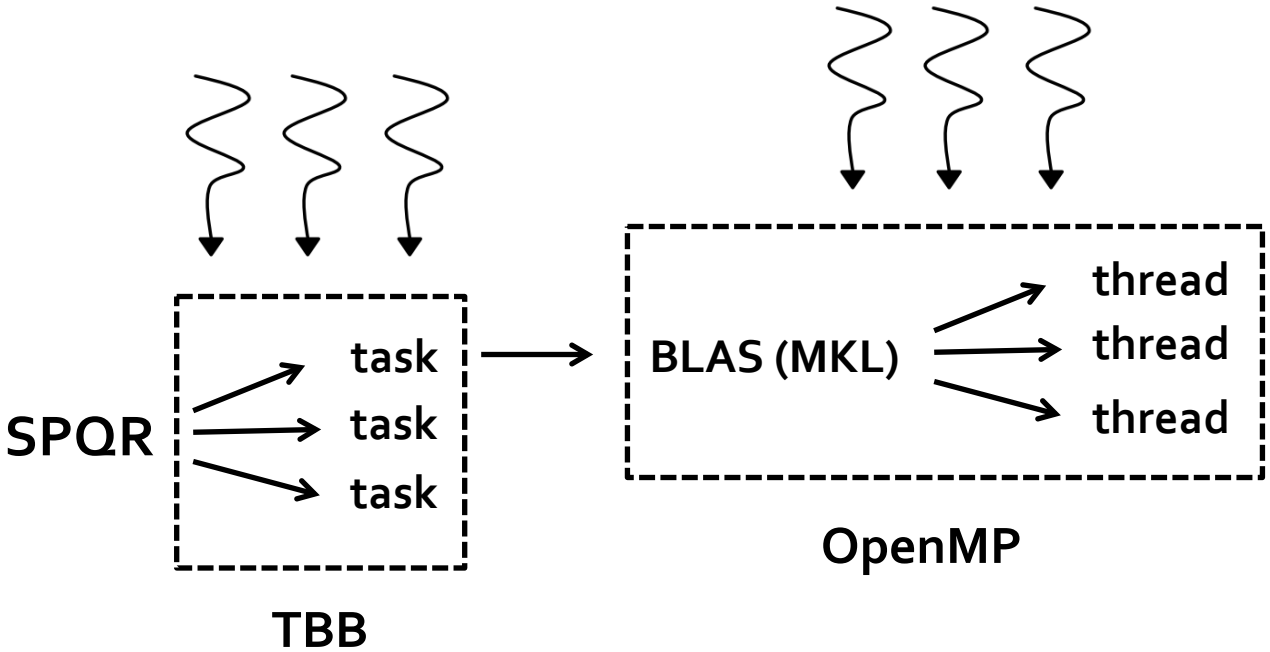
intra-process multi-level

example: sparse QR factorization (SPQR)



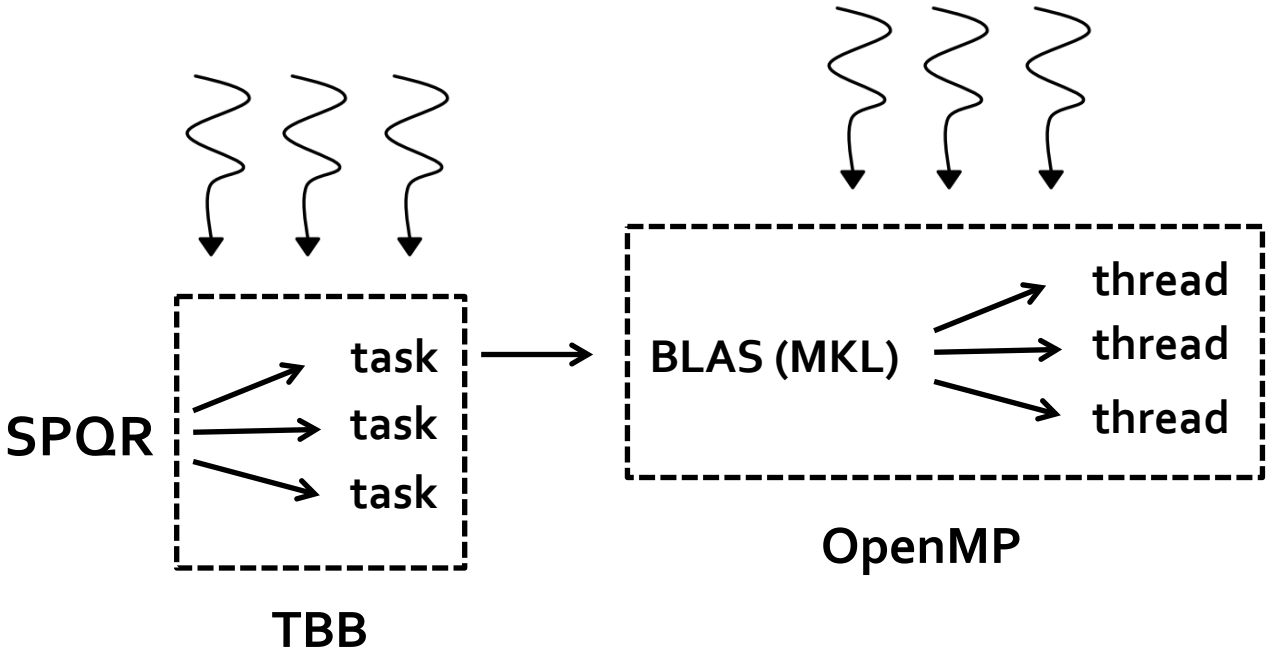
intra-process multi-level

how many CPUs (threads) to allocate between libraries?

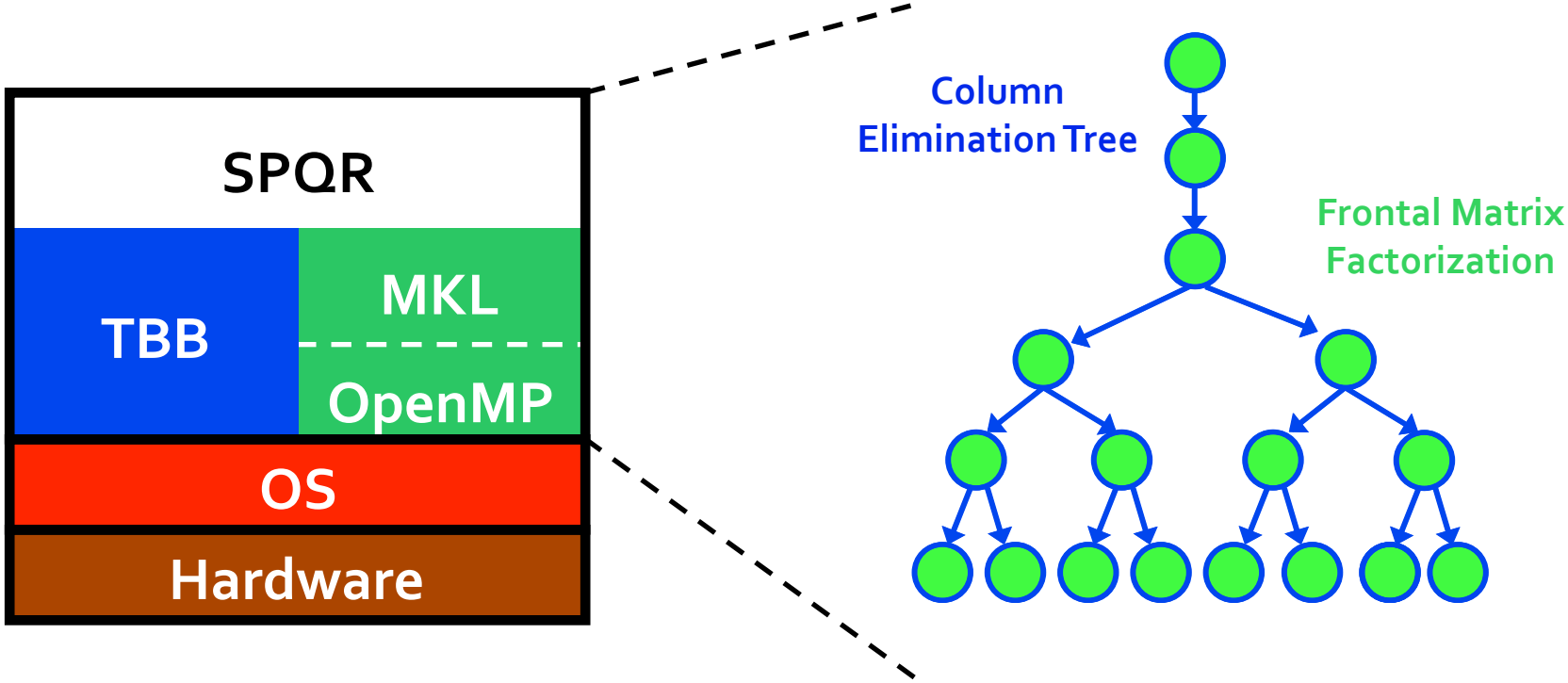


intra-process multi-level

be careful not to oversubscribe resources!



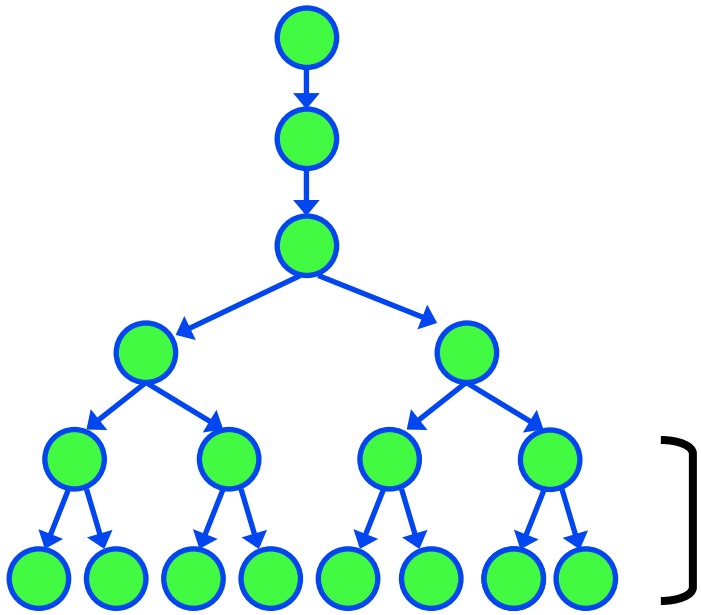
SPQR software architecture



system stack

software architecture

SPQR software architecture



oversubscription!

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

Using Intel MKL with Threaded Applications

http://www.intel.com/support/performancetools/libraries/mkl/sb/CS-017177.htm

Software Products

Intel® Math Kernel Library (Intel® MKL)
Using Intel® MKL with Threaded Applications

Page Contents:

- Memory Allocation MKL: Memory appears to be allocated and not released when calling some Intel MKL routines (e.g. sgetrf).
- Using Threading with BLAS and LAPACK
- Setting the Numbe
- Changing the Num
- Can I use Intel M

Memory Allocation MKL: M
some Intel® MKL routines
One of the advantages of us
OpenMP*. OpenMP* require
even for single-processor sy
occurs once the first time the
allocation persists until the a
will allocate a stack equal to
of memory that is automatica
allocations and the number o

Using Threading with BLA
Intel MKL is threaded in a nu
Level 3 BLAS, DFTs, and FF
situations in which conflicts c
We list them here with recom
the problem exists is appropr

If the user threads the program using OpenMP directives and uses the Intel® Compilers to compile the program, Intel MKL and the user program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads. But Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If the user program is threaded by some other means, Intel MKL may operate in multithreaded mode and the computations may be corrupted. Here are several cases and our recommendations:

- User threads the program using OS threads (pthreads on Linux*, Win32* threads on Windows*). 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.
- User threads the program using OpenMP directives and/or pragmas and compiles the program using a compiler other than a compiler from Intel. This is more problematic because setting OMP_NUM_THREADS in the environment affects both the compiler's threading library and the threading library with Intel MKL. In this case, the safe approach is to set OMP_NUM_THREADS=1.
- Multiple programs are running on a multiple-CPU system. In cluster applications, the parallel program can run separate instances of the program on each processor. However, the threading software will see multiple processors on the system even though each processor has a separate process running on it. In this case OMP_NUM_THREADS should be set to 1.
- If the variable OMP_NUM_THREADS environment variable is not set, then the default number of threads will be assumed 1.

Setting the Number of Threads for OpenMP* (OMP)

```
void main(int args, char *argv[]){

  double *a, *b, *c;
  a = new double [SIZE*SIZE];
  b = new double [SIZE*SIZE];
  c = new double [SIZE*SIZE];

  double alpha=1, beta=1;
  int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
  char transa='n', transb='n';

  for( i=0; i<SIZE; i++){
     for( j=0; j<SIZE; j++){
        a[i*SIZE+j]= (double)(i+j);
        b[i*SIZE+j]= (double)(i*j);
        c[i*SIZE+j]= (double)0;
     }
  }
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
              m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
  printf("row\ta\tc\n");
  for ( i=0;i<10;i++){
     printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
  }

  omp_set_num_threads(1);

  for( i=0; i<SIZE; i++){
     for( j=0; j<SIZE; j++){
        a[i*SIZE+j]= (double)(i+j);
        b[i*SIZE+j]= (double)(i*j);
        c[i*SIZE+j]= (double)0;
     }
  }
  printf("row\ta\tc\n");
  for ( i=0;i<10;i++){
     printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
  }

  delete [] a;
  delete [] b;
  delete [] c;
}
```

Can I use Intel MKL if I thread my application?
The Intel Math Kernel Library is designed and compiled for thread safety so it can be called from programs that are threaded. Calling Intel MKL routines that are threaded from multiple application threads can lead to conflict (including incorrect answers or program failures), if the calling library differs from the Intel MKL threading library.

- **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:

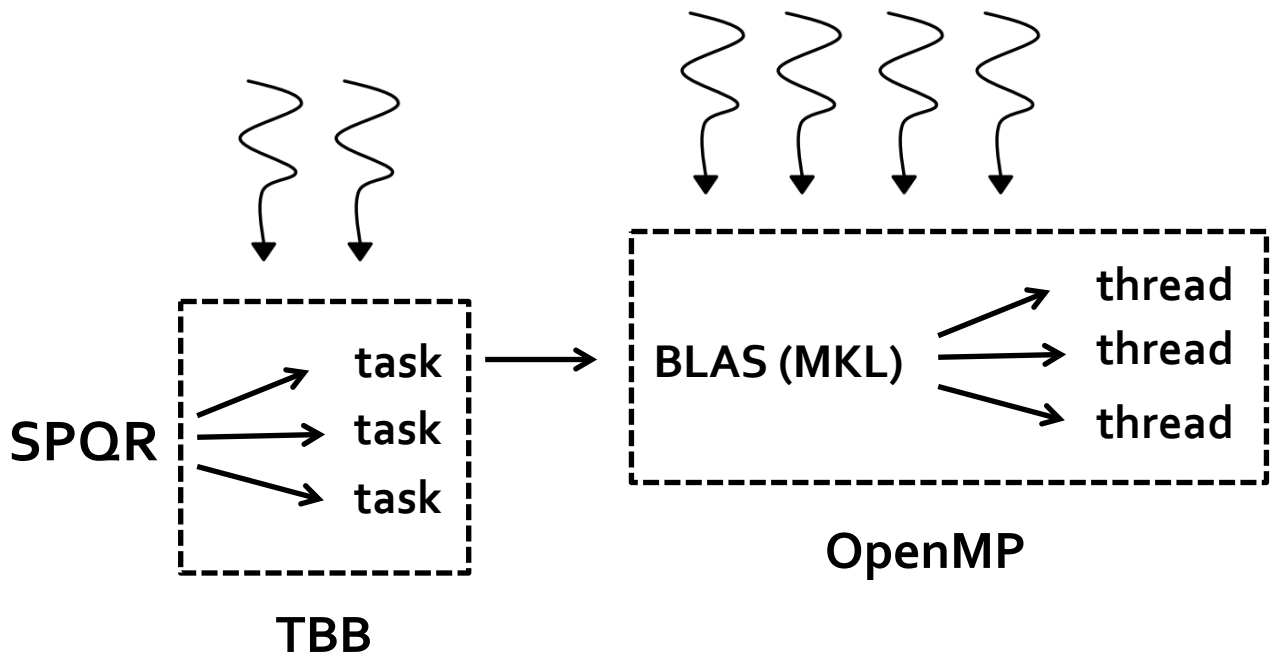
MKL function is called from a critical section of a parallel region - set MKL_NUM_THREADS=N, where N is the number of physical cores in the system (or use mkl_set_num_thread(N) routine) .

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

recommendation: static partitioning

static partitioning

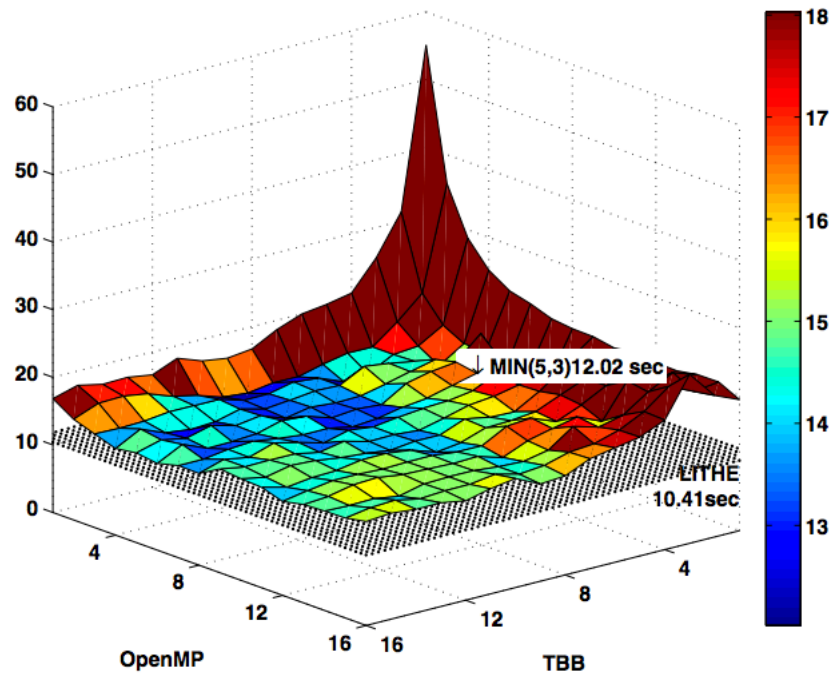
fix the number of CPUs (threads) allocated between libraries



optimal allocation?

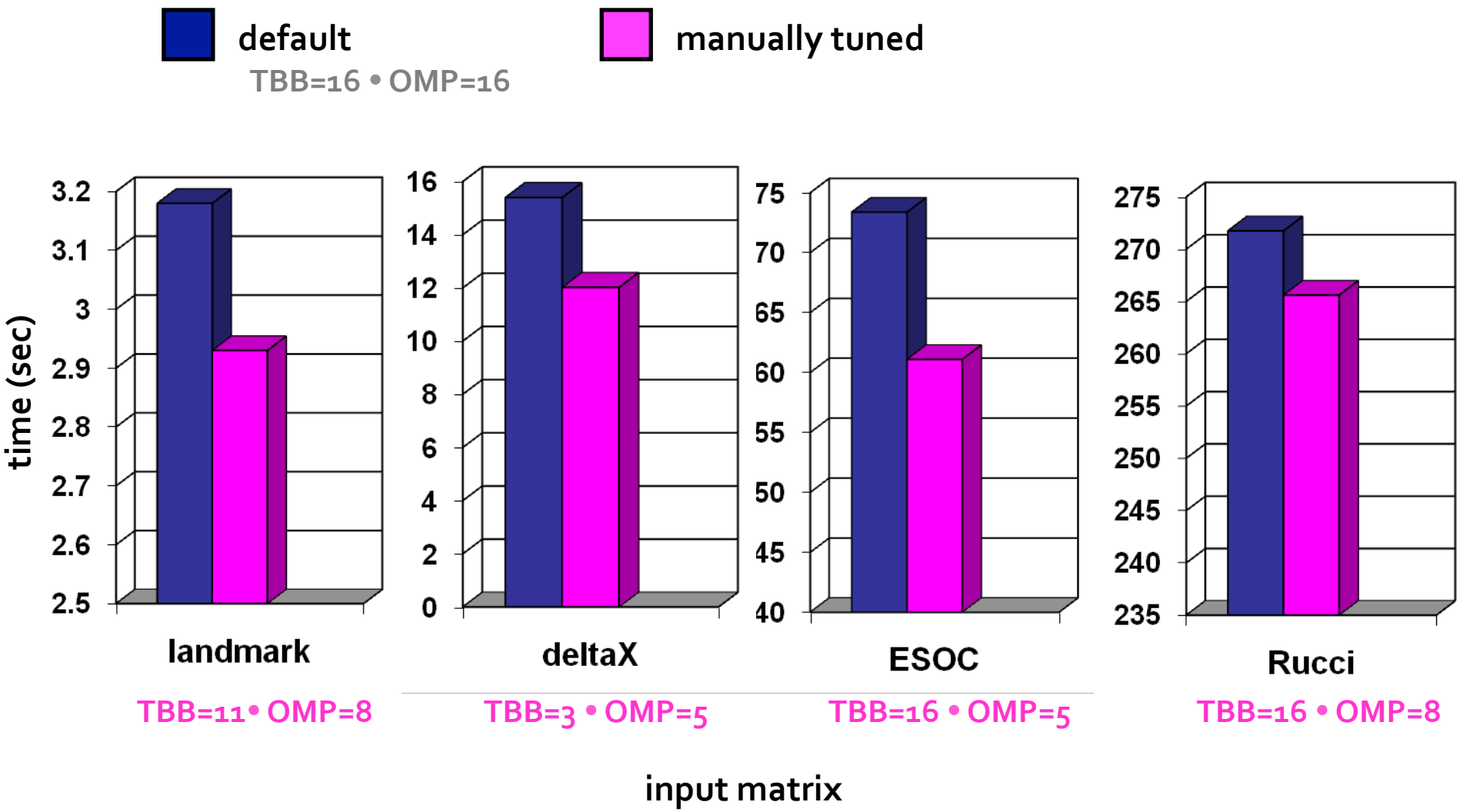
recommendation: manual tuning

static partitioning



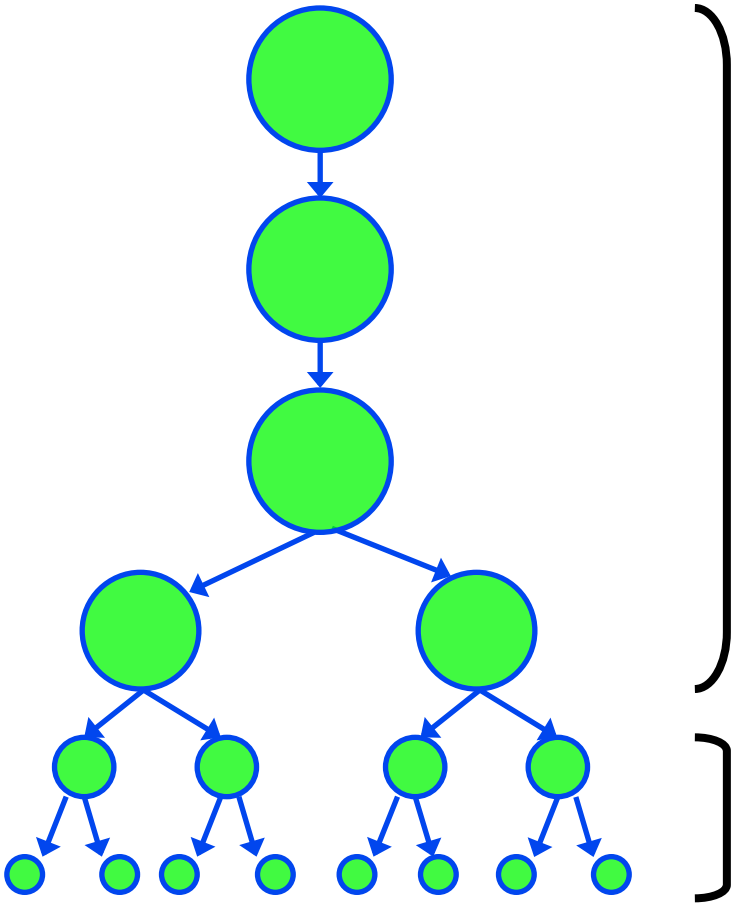
Runtime (in seconds) for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)

static partitioning



can’t we do better?

dynamic needs



Give resources to OpenMP

Give resources to TBB

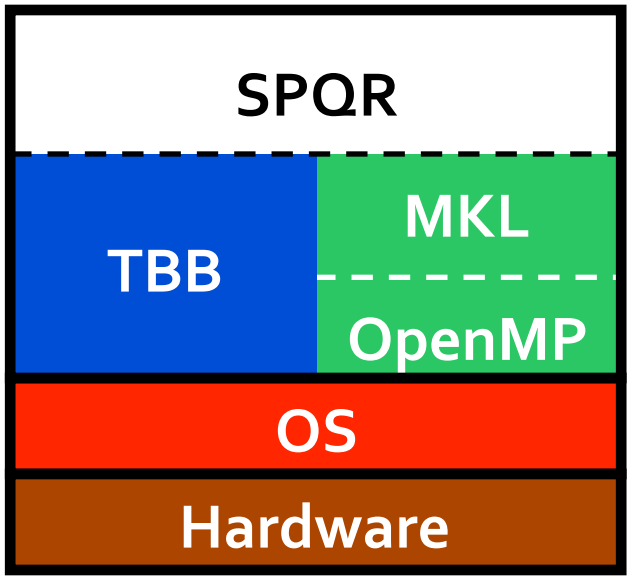
manual tuning means a fixed TBB/OMP thread allocation throughout execution

static partitioning is scheduling

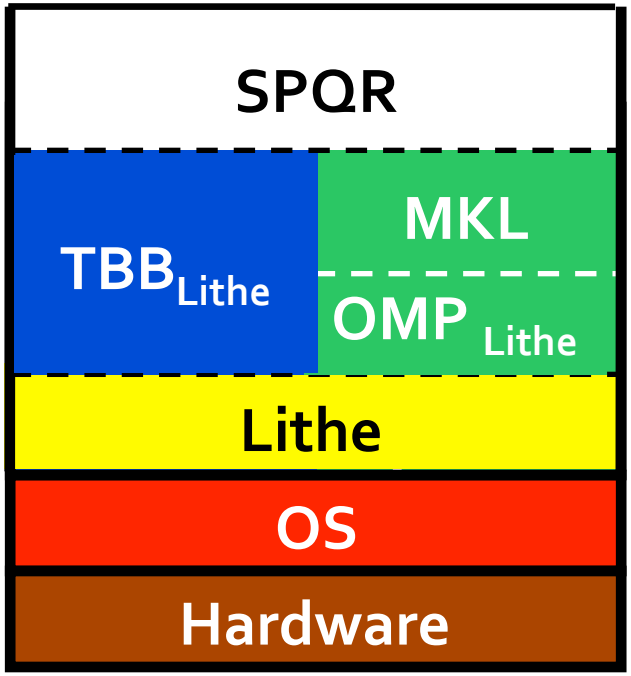
static partitioning
is scheduling;
but sub-optimal
(scheduler/human is
missing information)

Lithe: multi-level scheduling between libraries

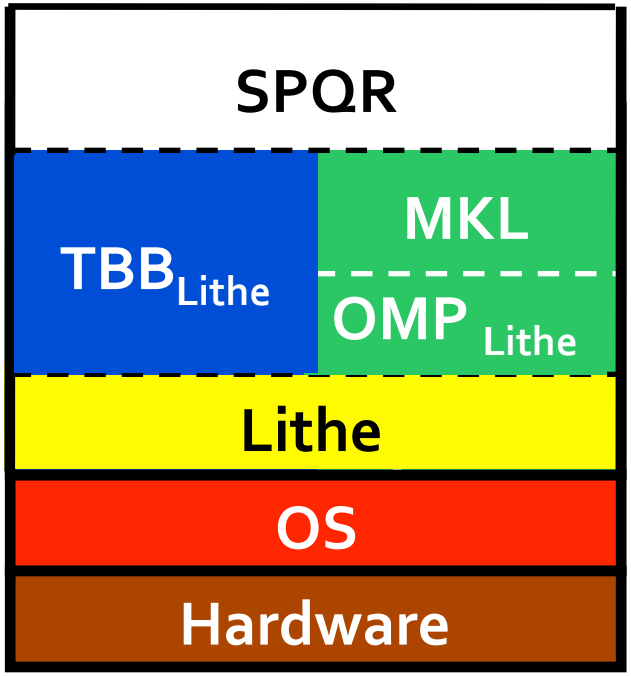
Lithe



Lithe



Lithe



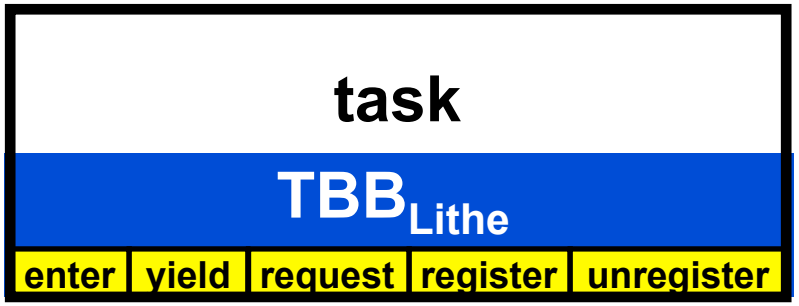
library interfaces remain the same

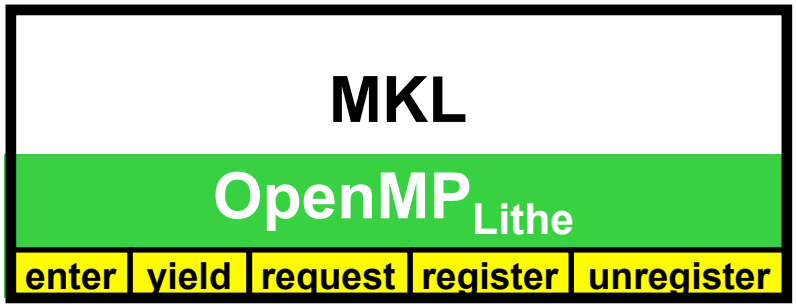
zero lines of high-level codes changed (SPQR, MKL)

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

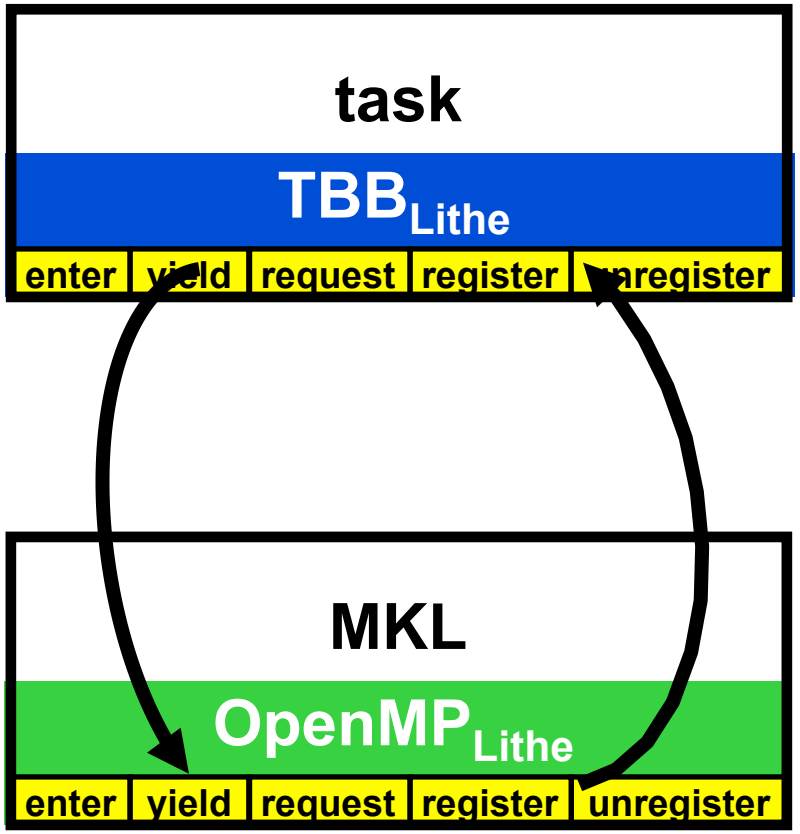
just link in Lithe runtime + Lithe versions of libraries (TBB, OpenMP)

Lithe "upcall" interface





Lithe "upcall" interface

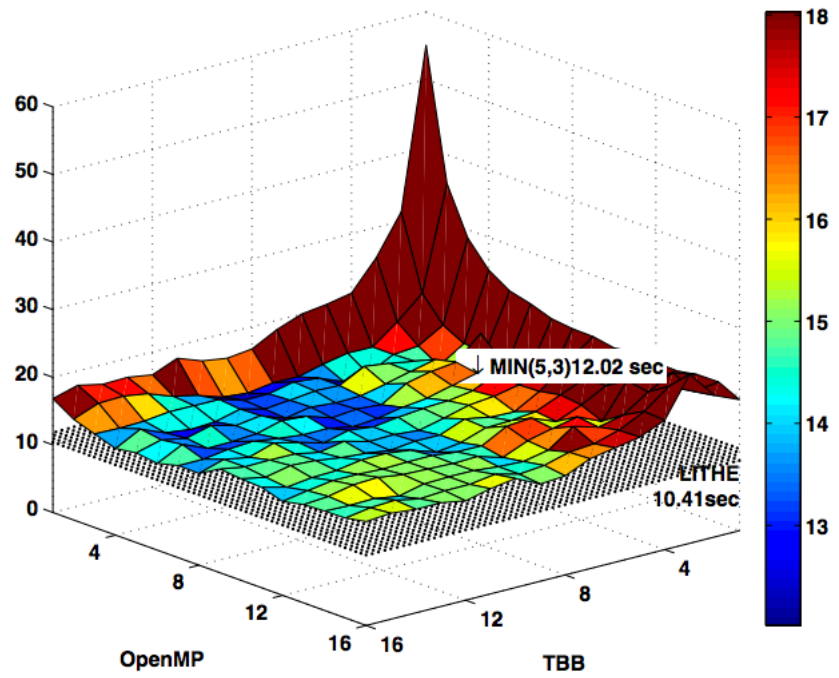


Lithe

dynamically allocate CPUs (threads) between TBB and OpenMP

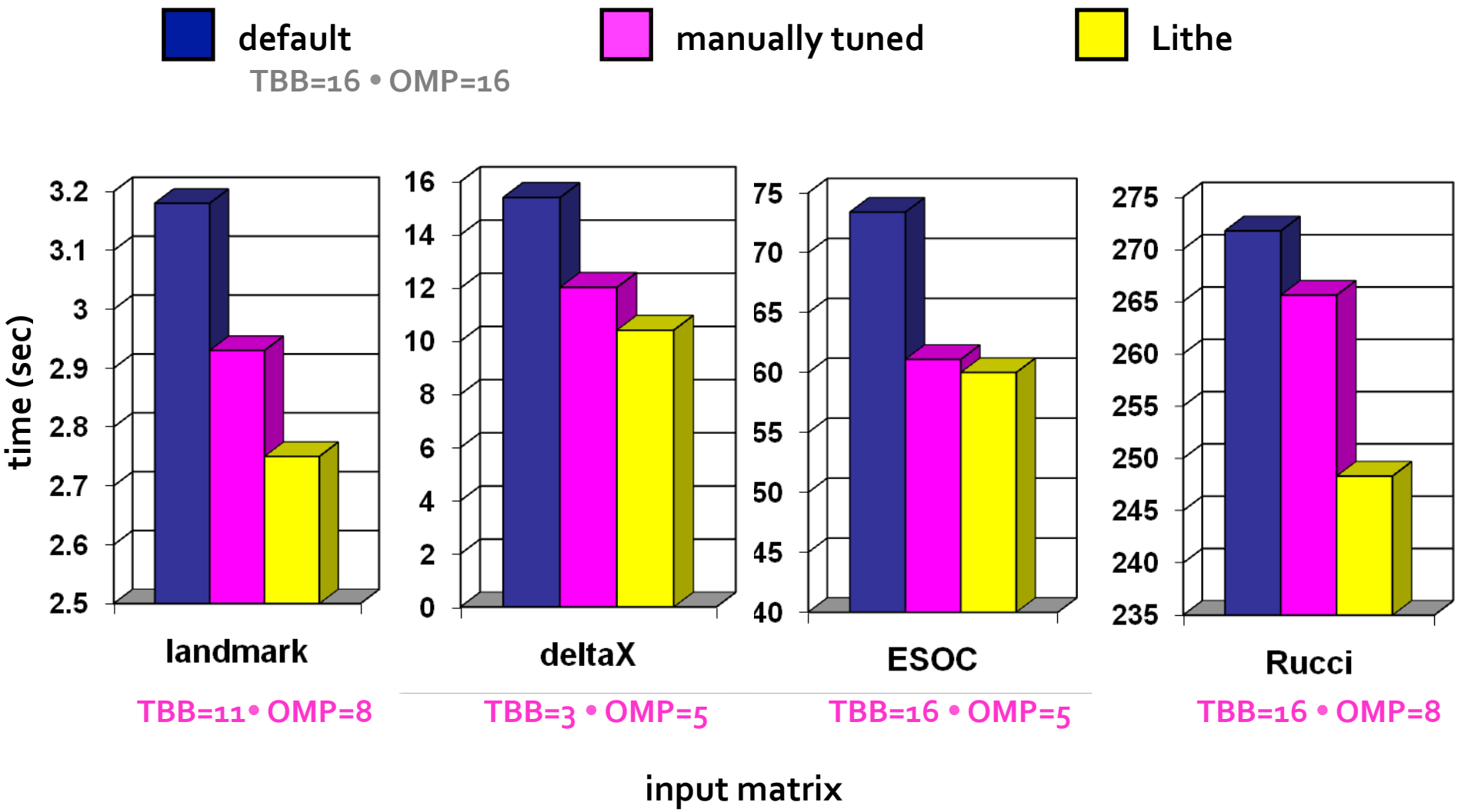
… matching demand as closely as possible

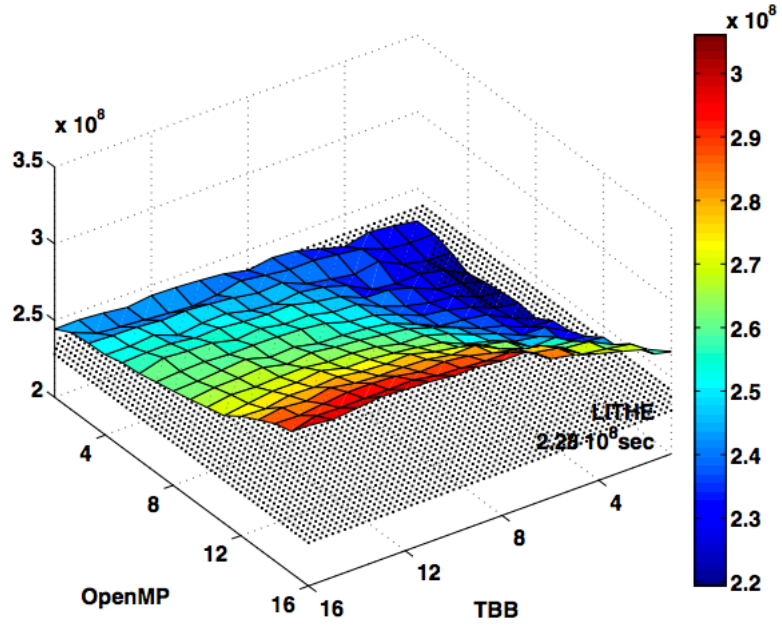
Lithe



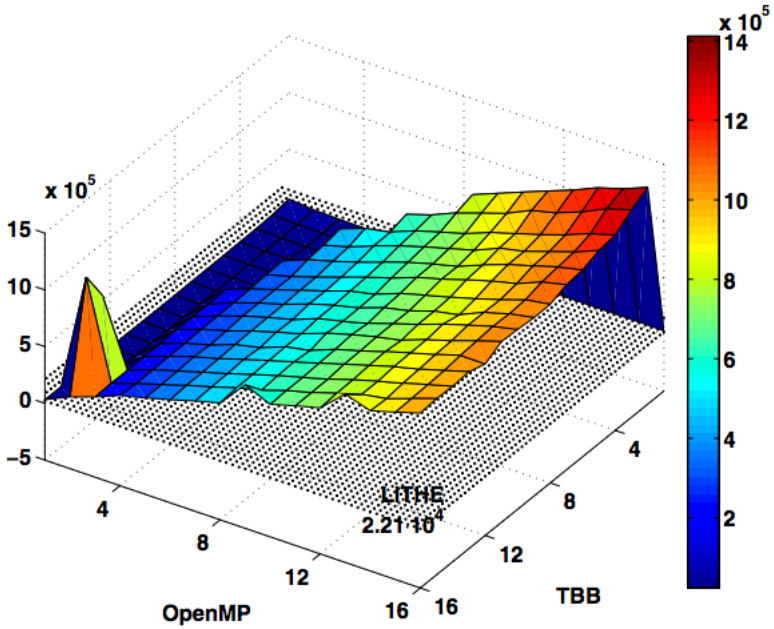
Runtime (in seconds) for varying allocations of CPUs between TBB and OpenMP (lower/bluer is better)

Lithe


default
TBB=16 • OMP=16
manually tuned
Lithe
time (sec)
3.2
3.1
3
2.9
2.8
2.7
2.6
2.5
16
14
12
10
8
6
4
2
0
75
70
65
60
55
50
45
40
275
270
265
260
255
250
245
240
235
landmark
deltaX
ESOC
Rucci
TBB=11• OMP=8
TBB=3 • OMP=5
TBB=16 • OMP=5
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

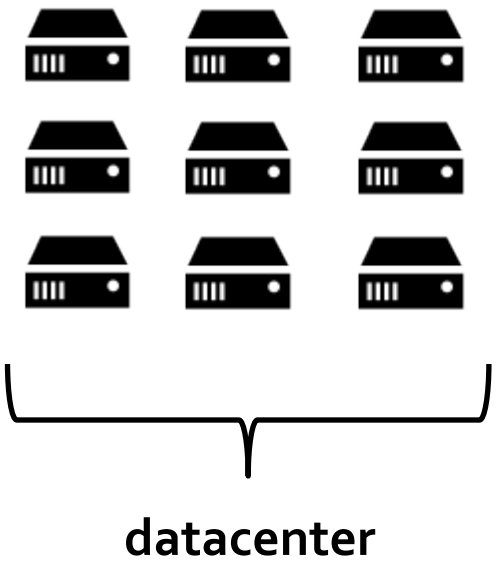
Lithe → Mesos

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

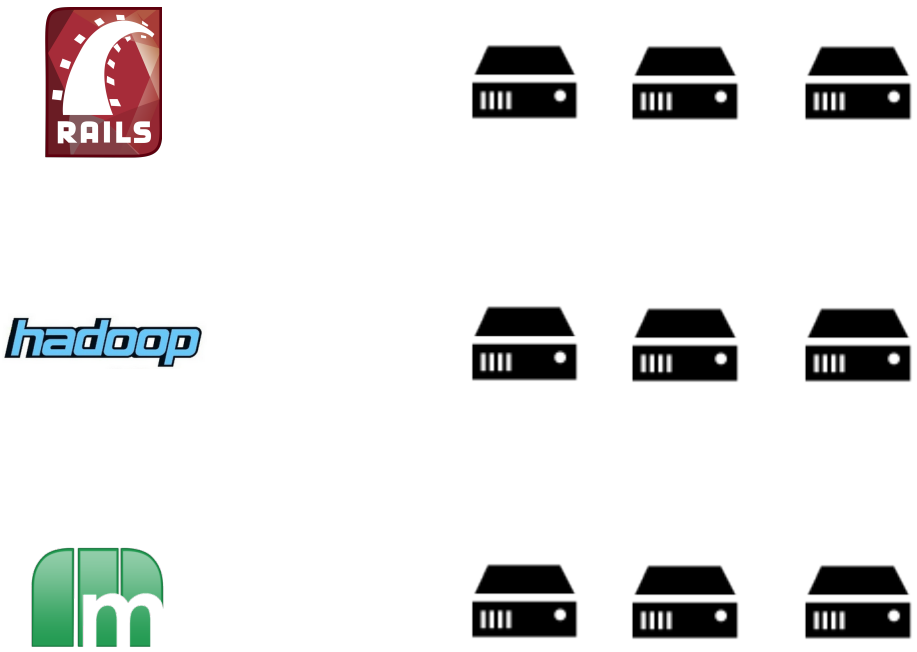
Mesos: sharing a datacenters resources between distributed systems via multi-level scheduling

static partitioning status quo in datacenters

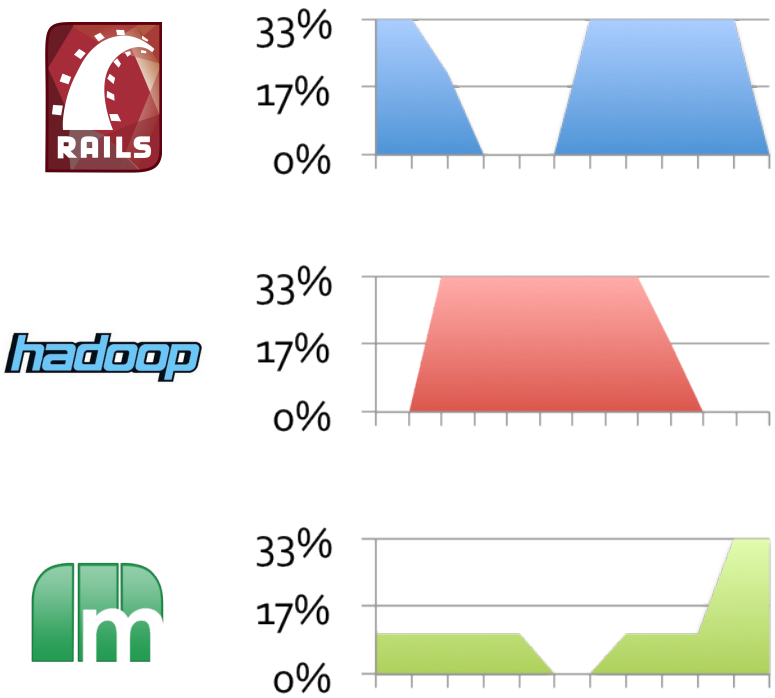
static partitioning considered harmful



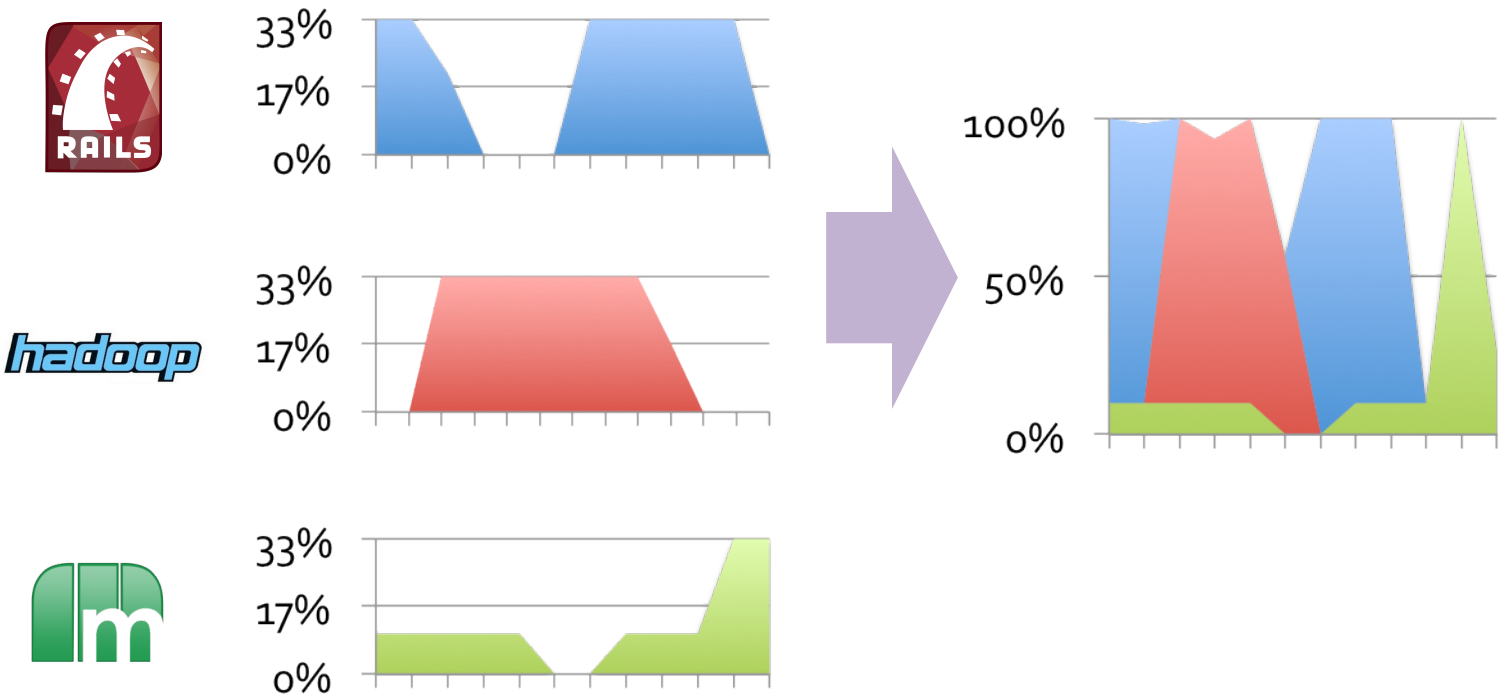
static partitioning considered harmful



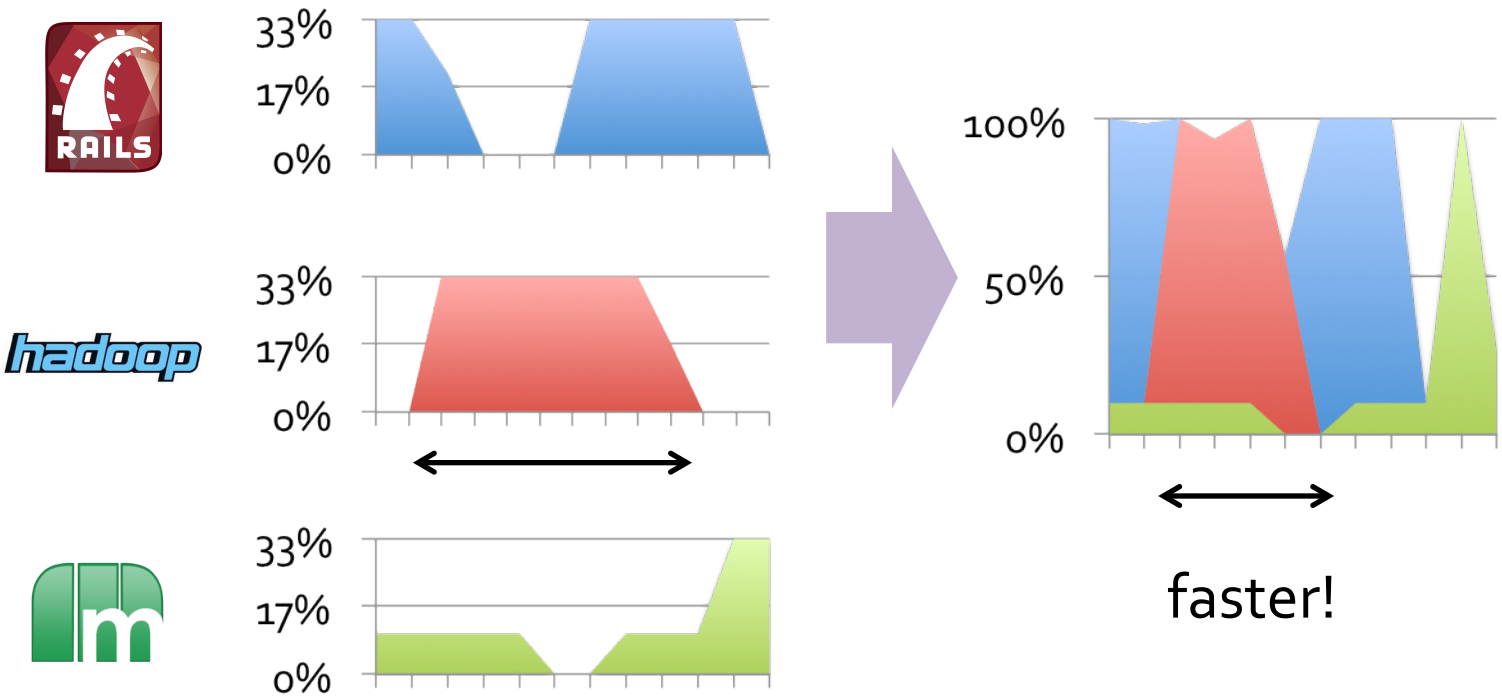
static partitioning considered harmful



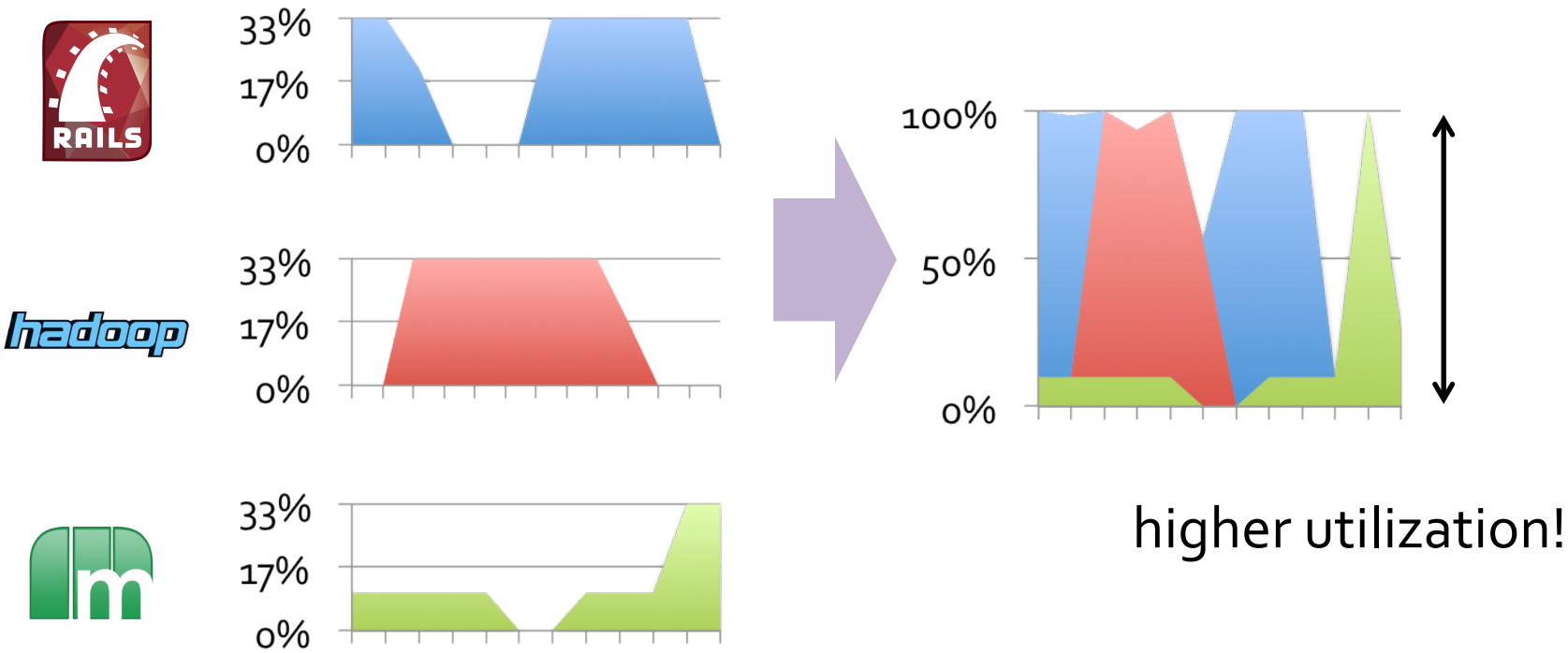
static partitioning considered harmful



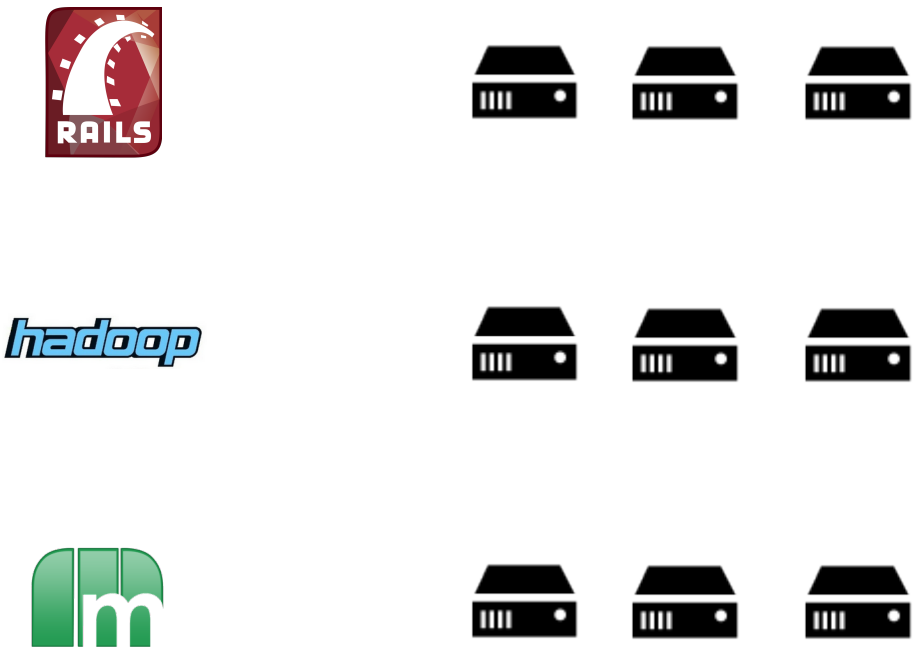
static partitioning considered harmful



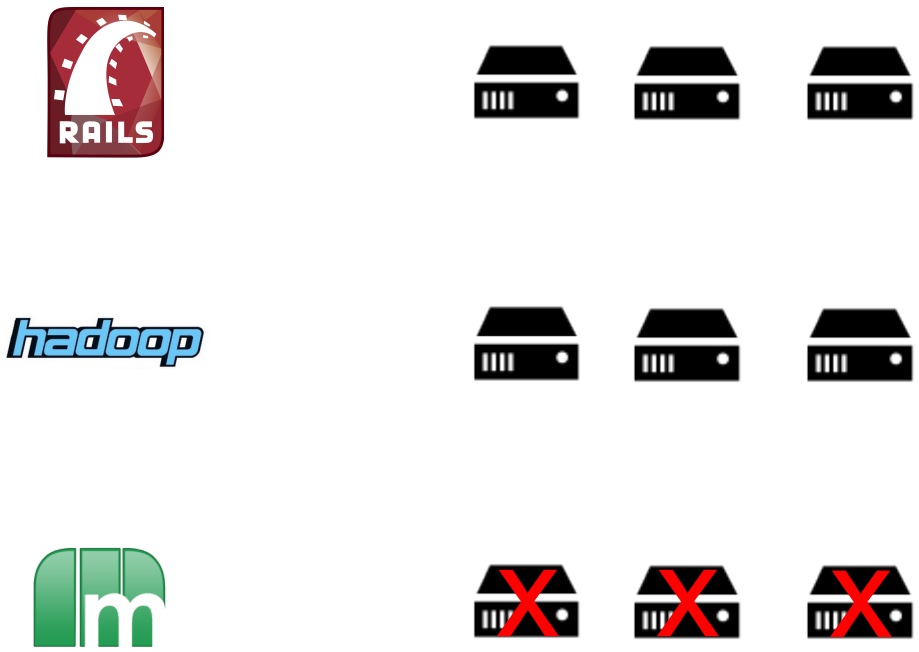
static partitioning considered harmful



static partitioning considered harmful



static partitioning considered harmful



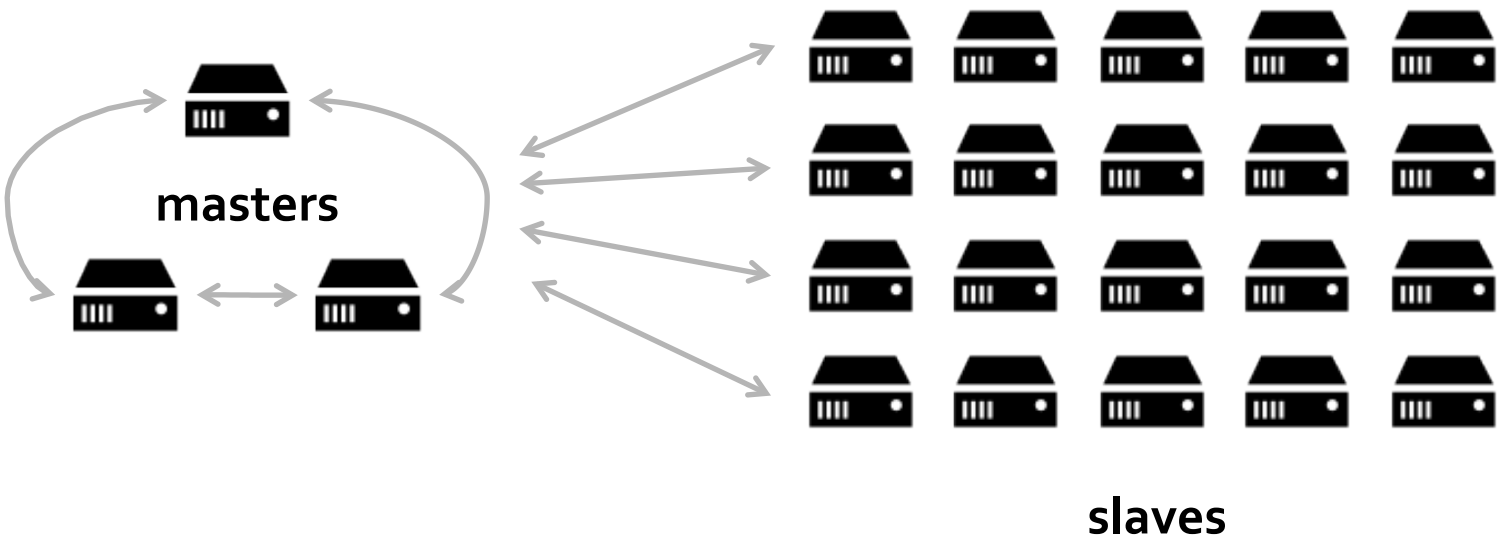
what if …

we built something to *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

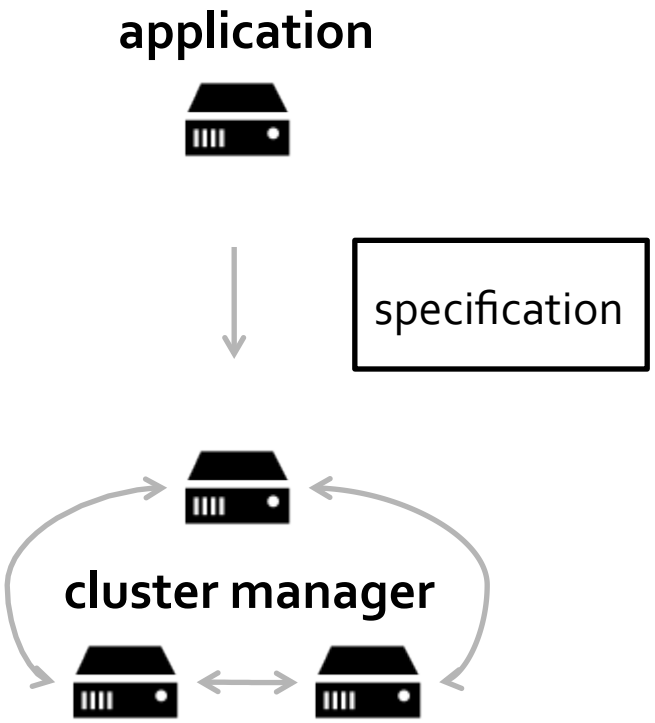
Apache Mesos: cluster manager based on multi-level scheduling

Mesos is a cluster manager with a master/slave architecture



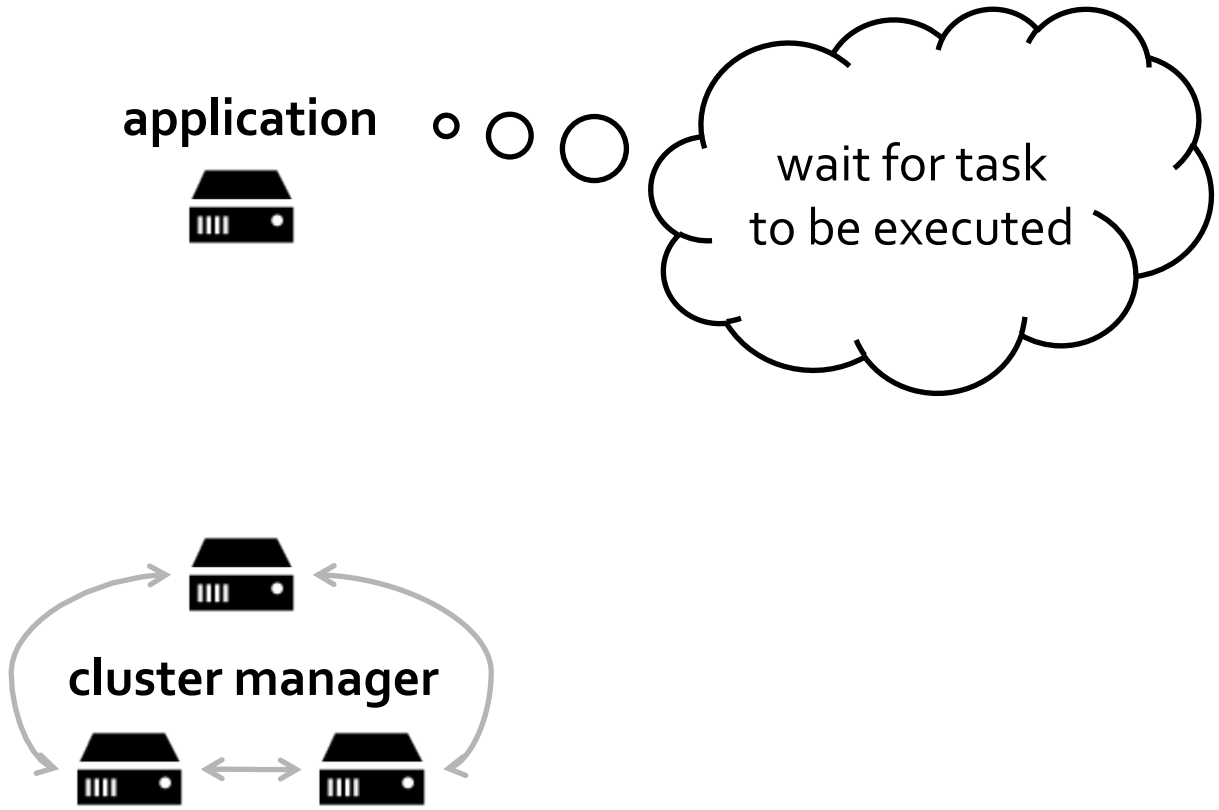
Mesos challenged the status quo of cluster managers

cluster manager status quo

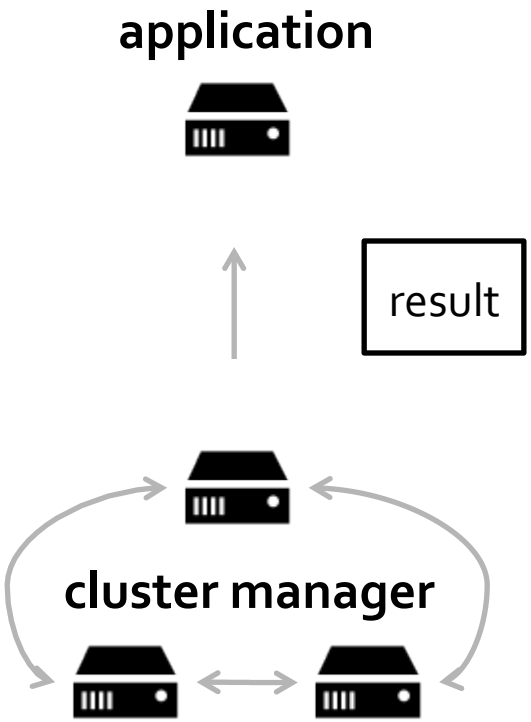


the specification includes as much information as possible to assist the cluster manager in scheduling and execution

cluster manager status quo



cluster manager status quo

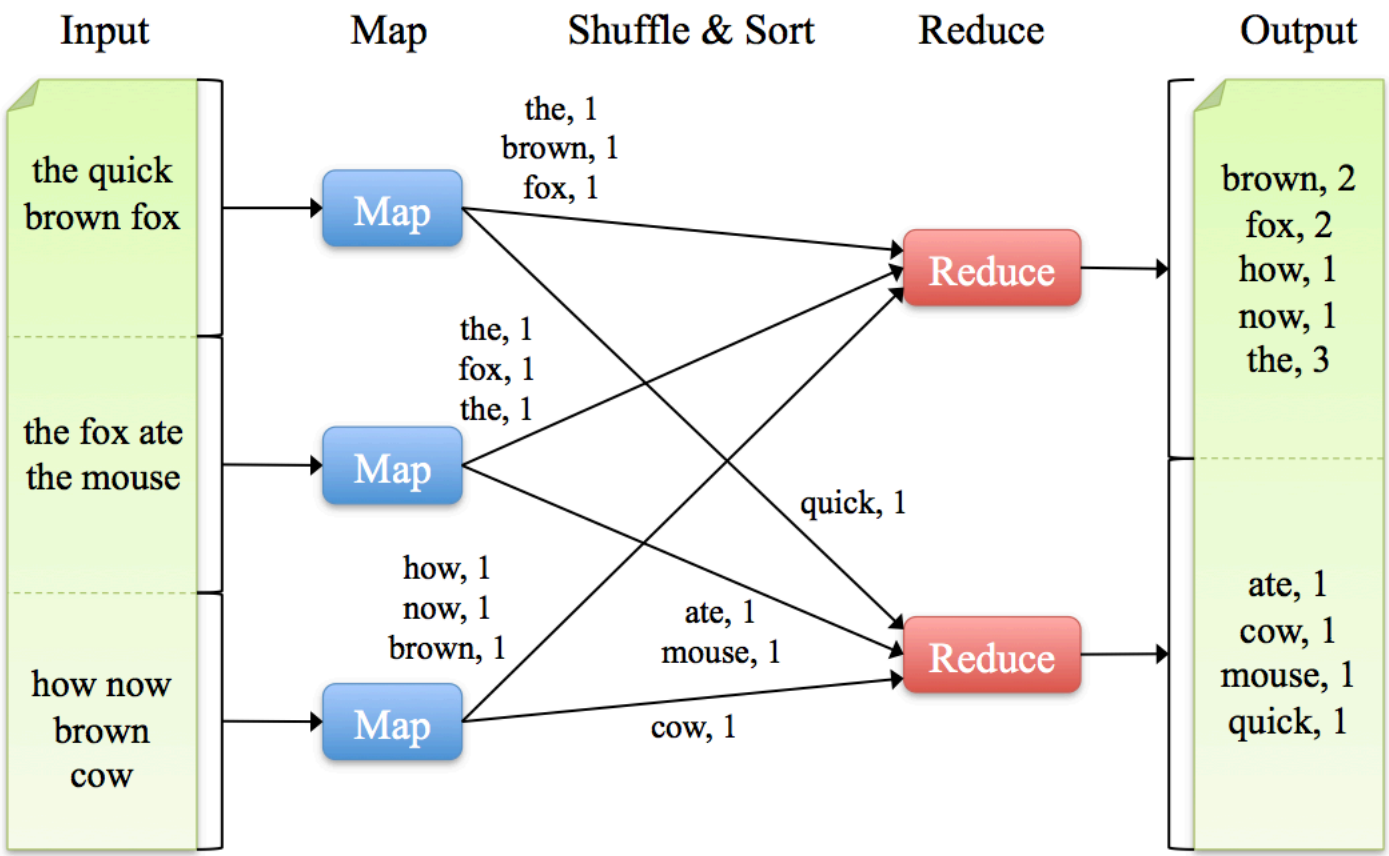


problems with specifications

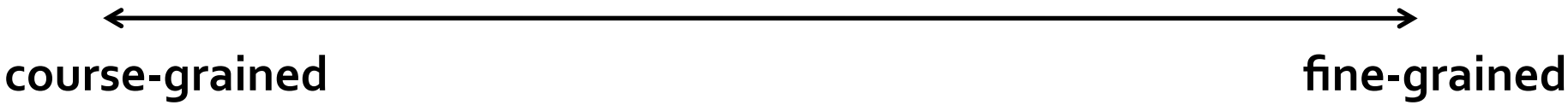
① hard to specify certain desires or constraints

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

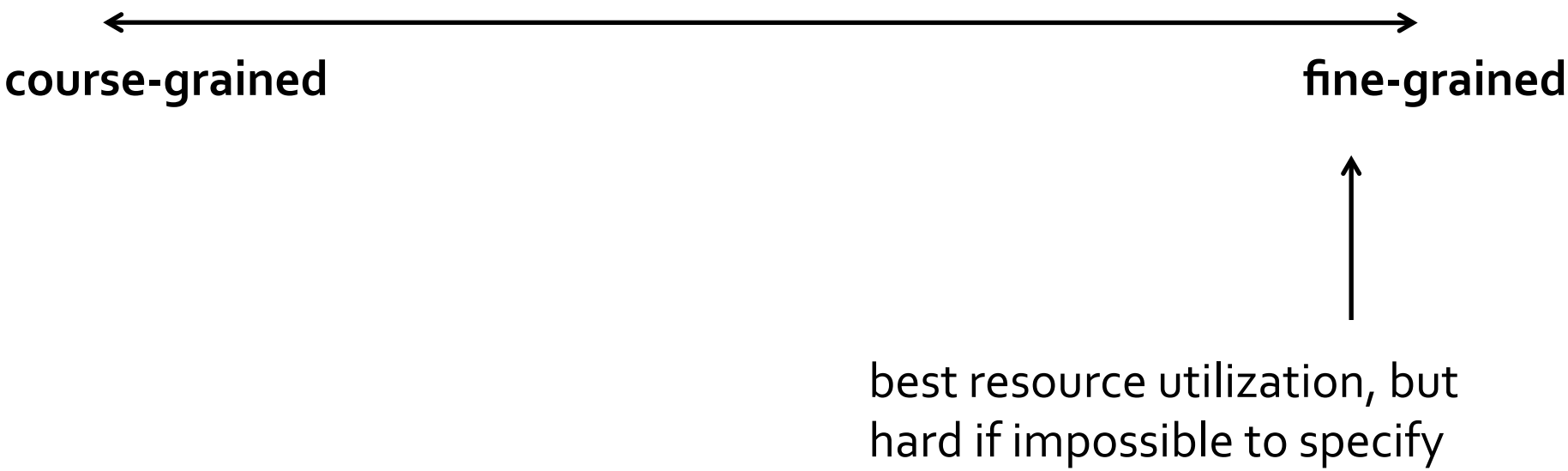
MapReduce specification?



MapReduce specification?



MapReduce specification?



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

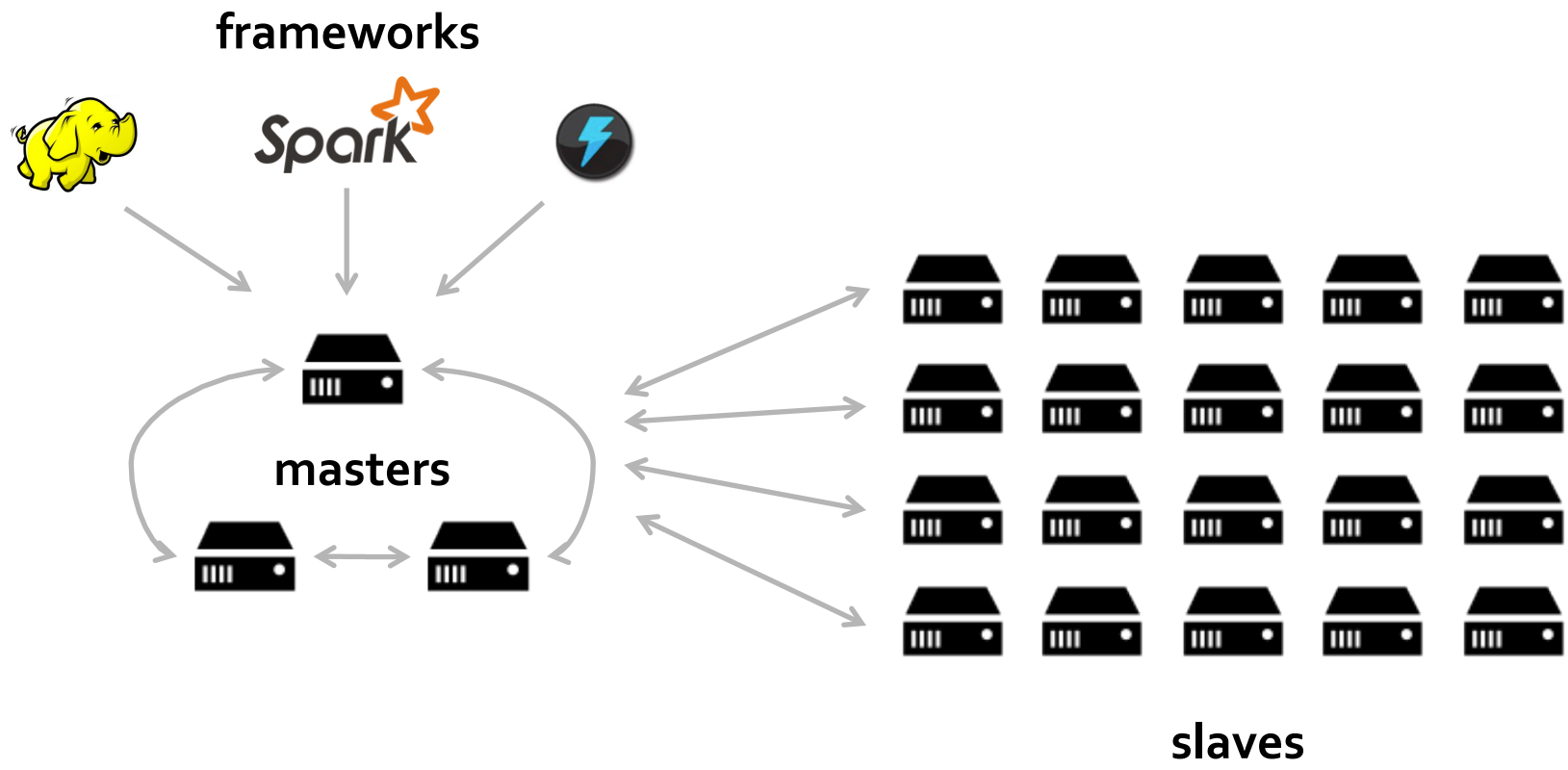
multi-level scheduling

motivation:

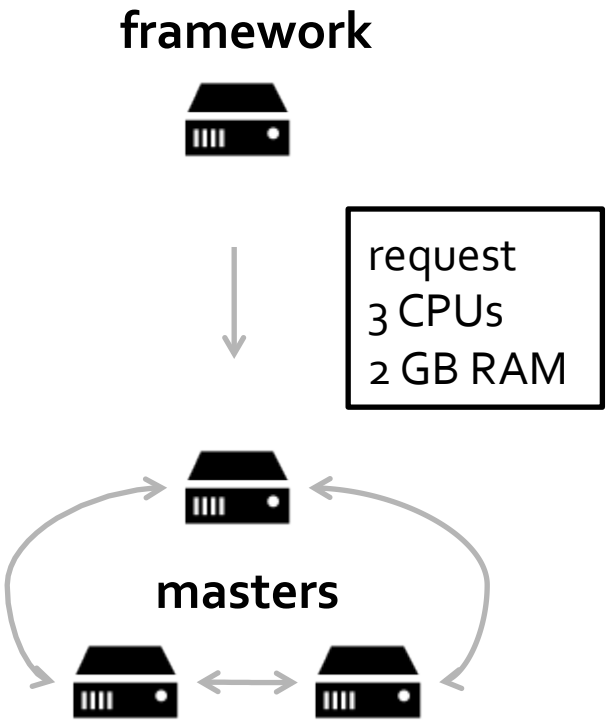
scheduler has *inadequate knowledge* of application/ framework's execution needs/semantics to make optimal decisions

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 …**

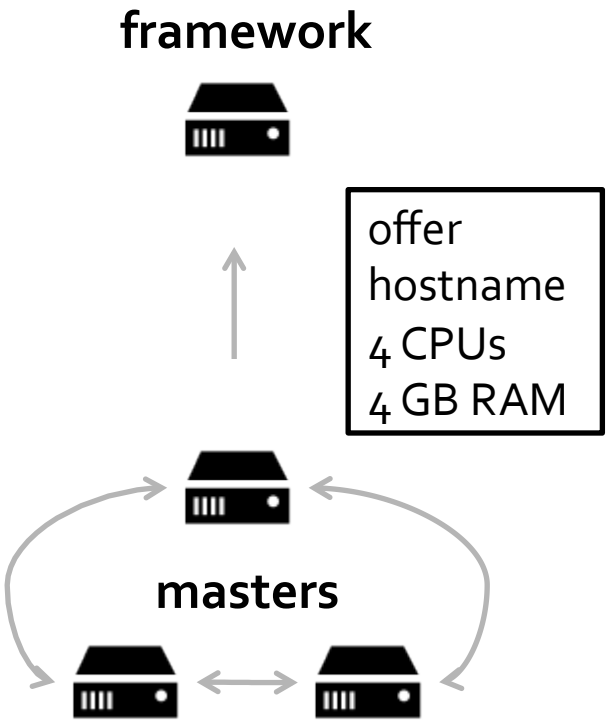
② ***offer* best you can <u>immediately</u>**

question: what should you do if you can't satisfy a request?

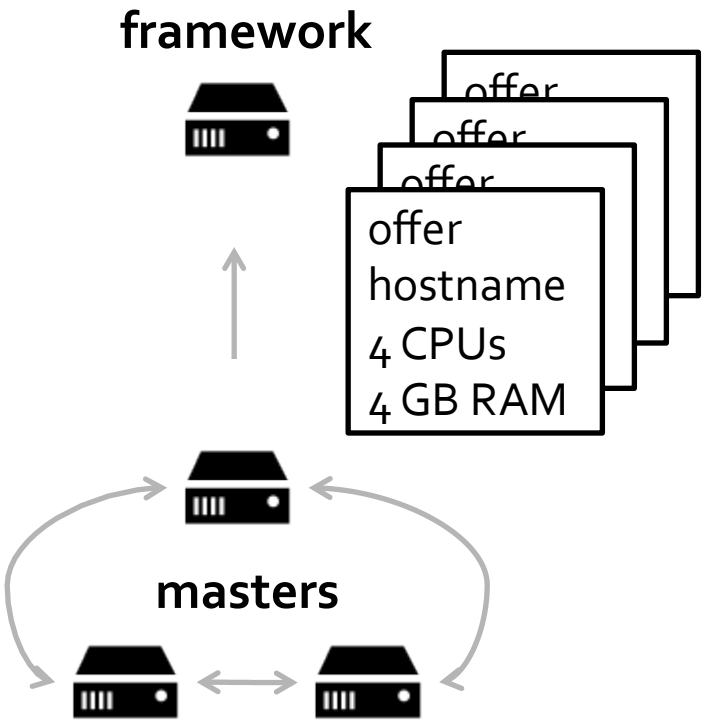
① wait until you can …

② *offer* best you can <u>immediately</u>

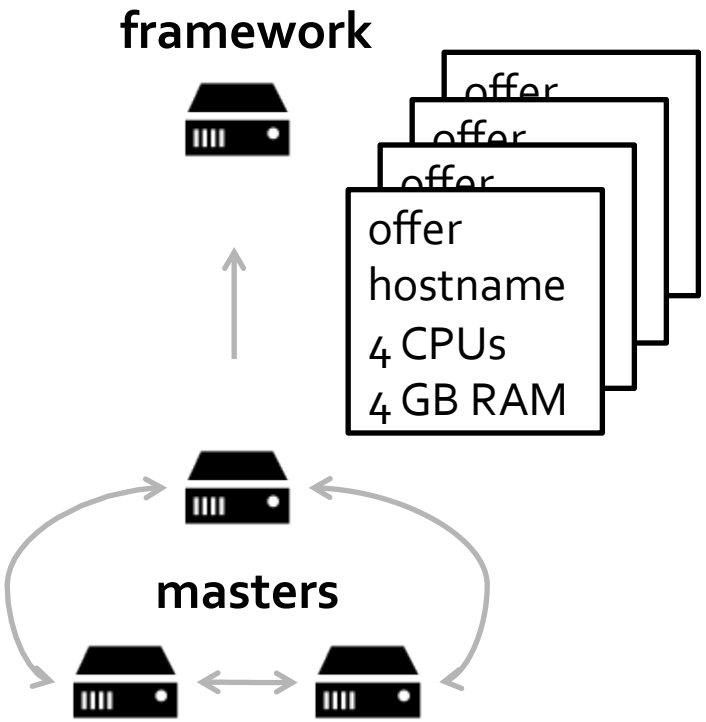
Mesos model



Mesos model

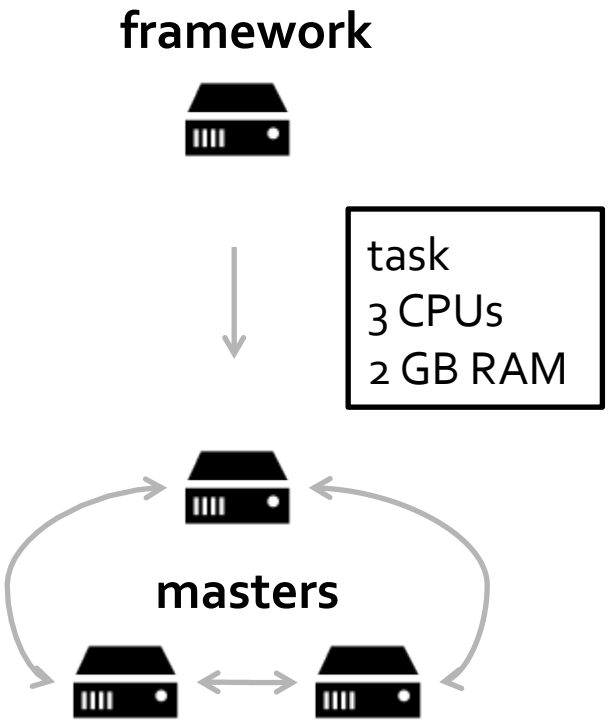

framework
offer
offer
offer
offer
hostname
4 CPUs
4 GB RAM
masters

Mesos model



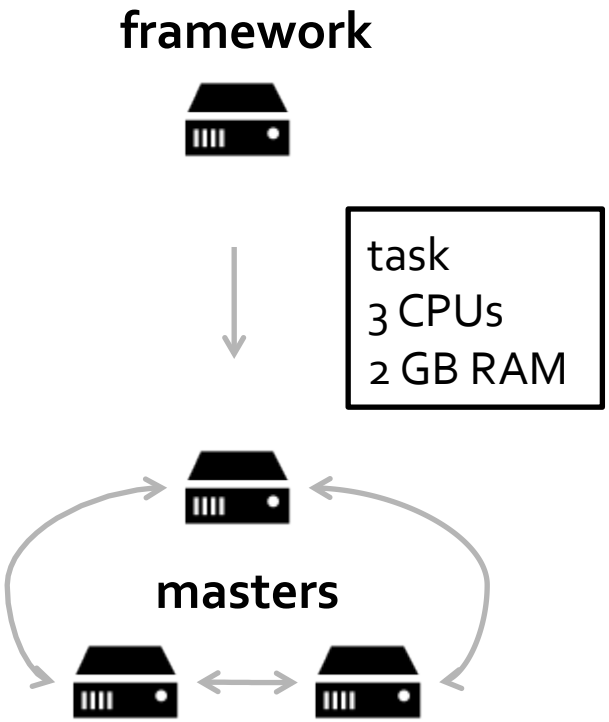
framework uses the offers to perform it's own *scheduling*

Mesos model



framework uses the offers to perform it's own *scheduling*

Mesos model



framework uses the offers to perform it's own *scheduling*

multi-level scheduling

1st level: master makes allocations via offers

2nd level: frameworks schedule tasks using offers

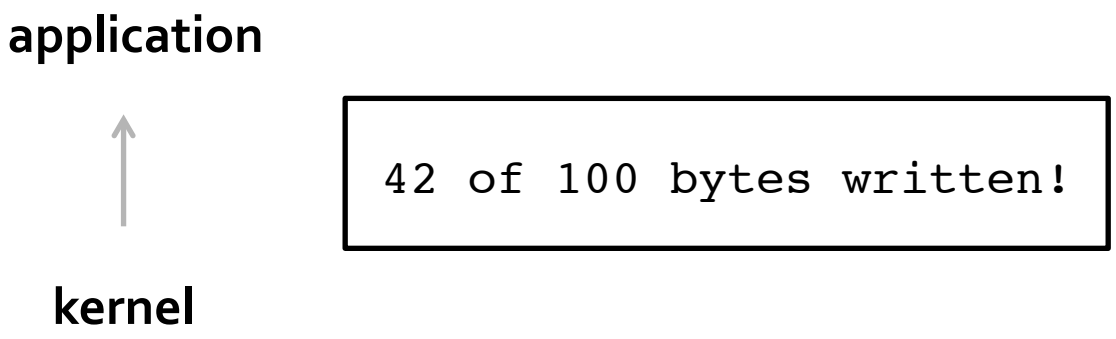
an analogue: non-blocking sockets

application

kernel

```
write(s, buffer, size);
```

an analogue: non-blocking sockets

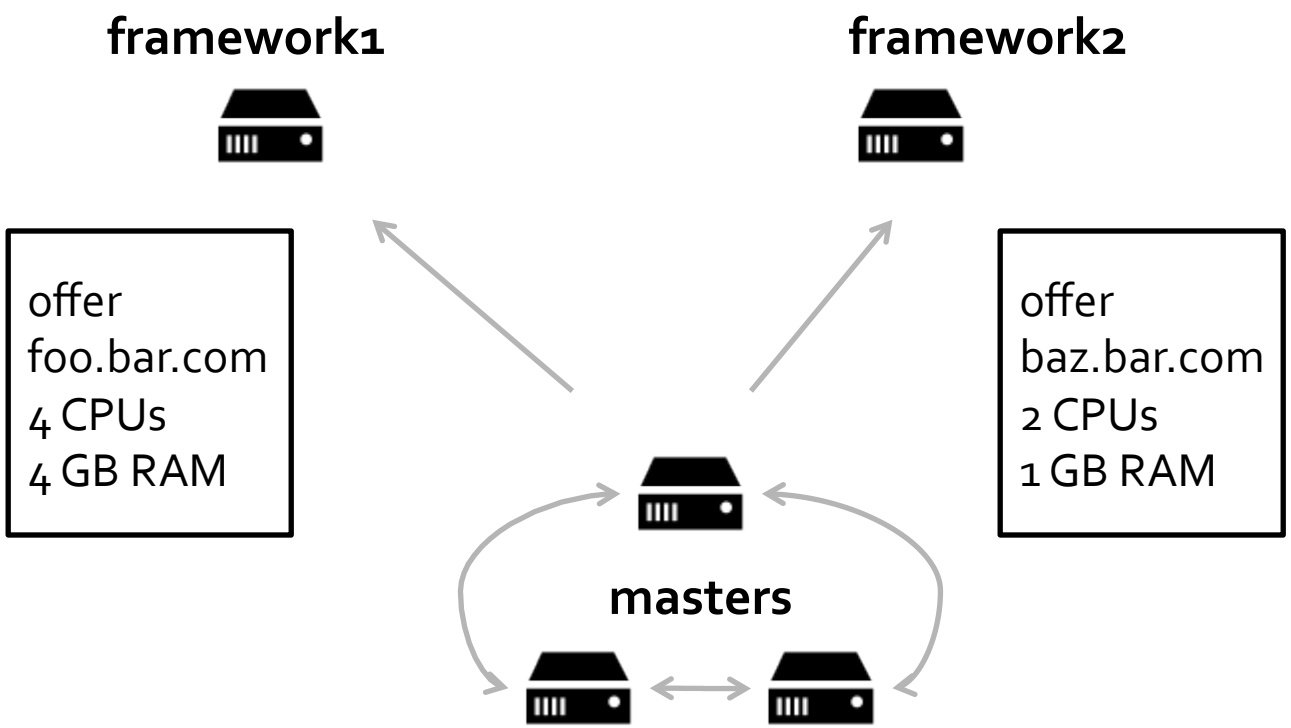


agenda

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

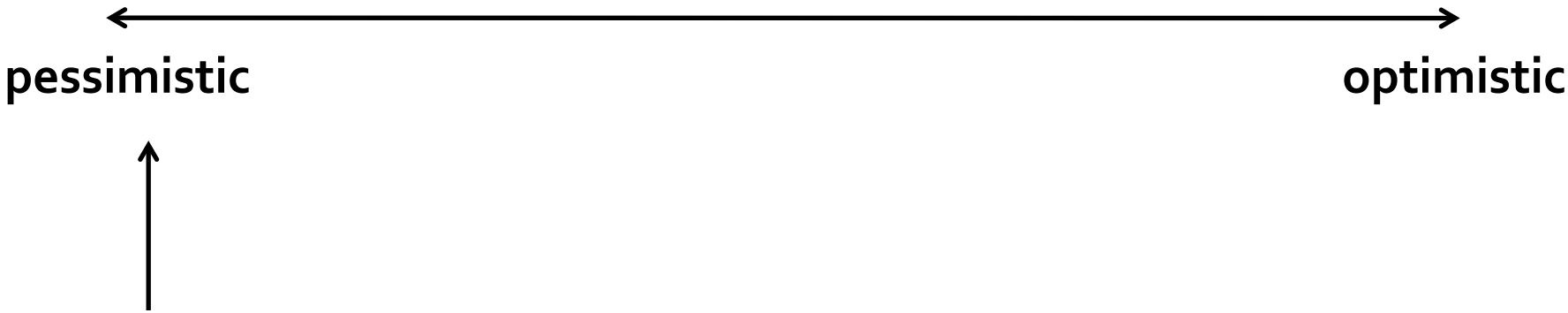
multiple offers outstanding



question: should resources within offers be disjoint?

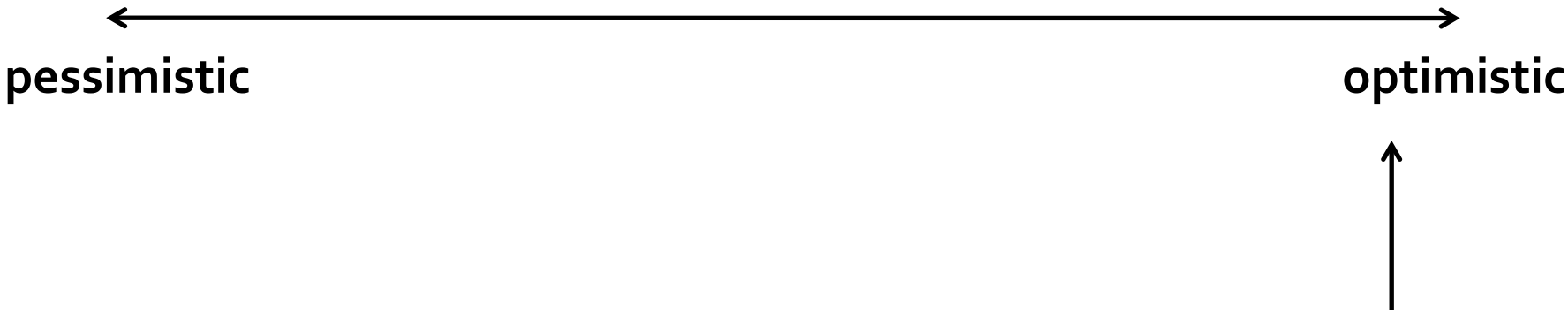
question: should multiple frameworks be able to compete for available resources?

concurrency control



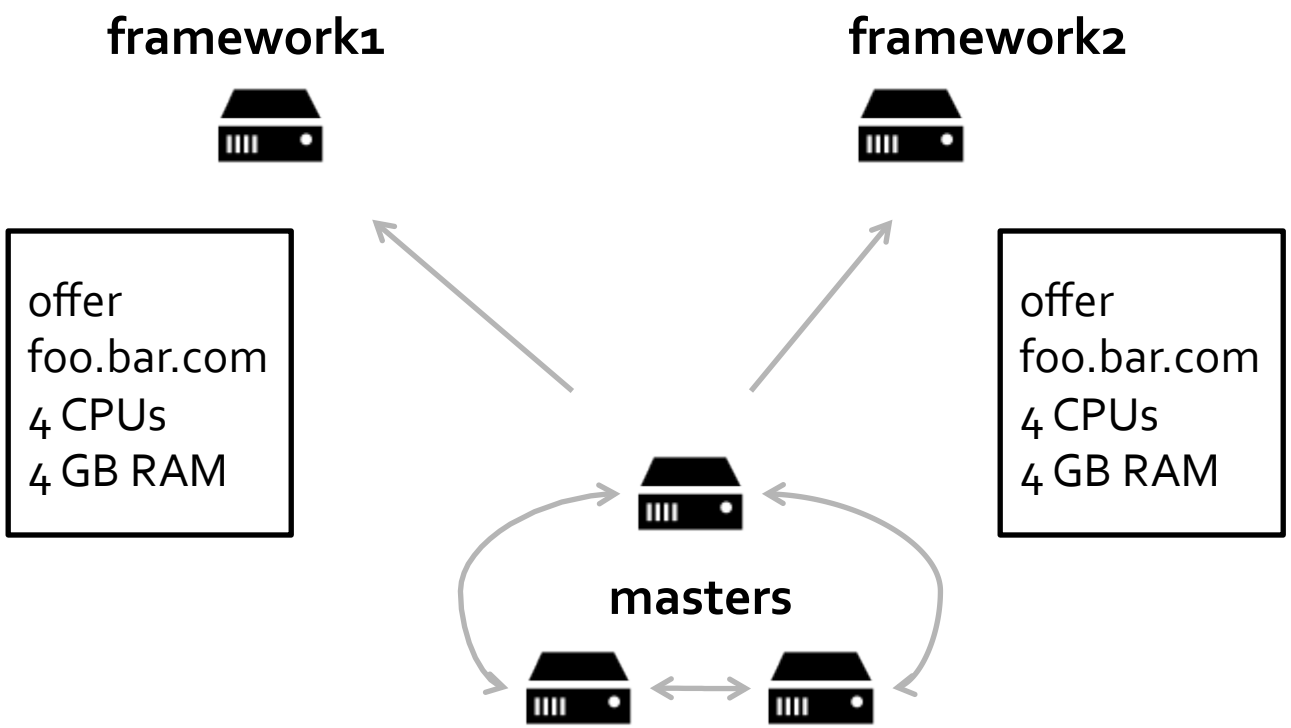
offers made to different frameworks are *disjoint*

concurrency control



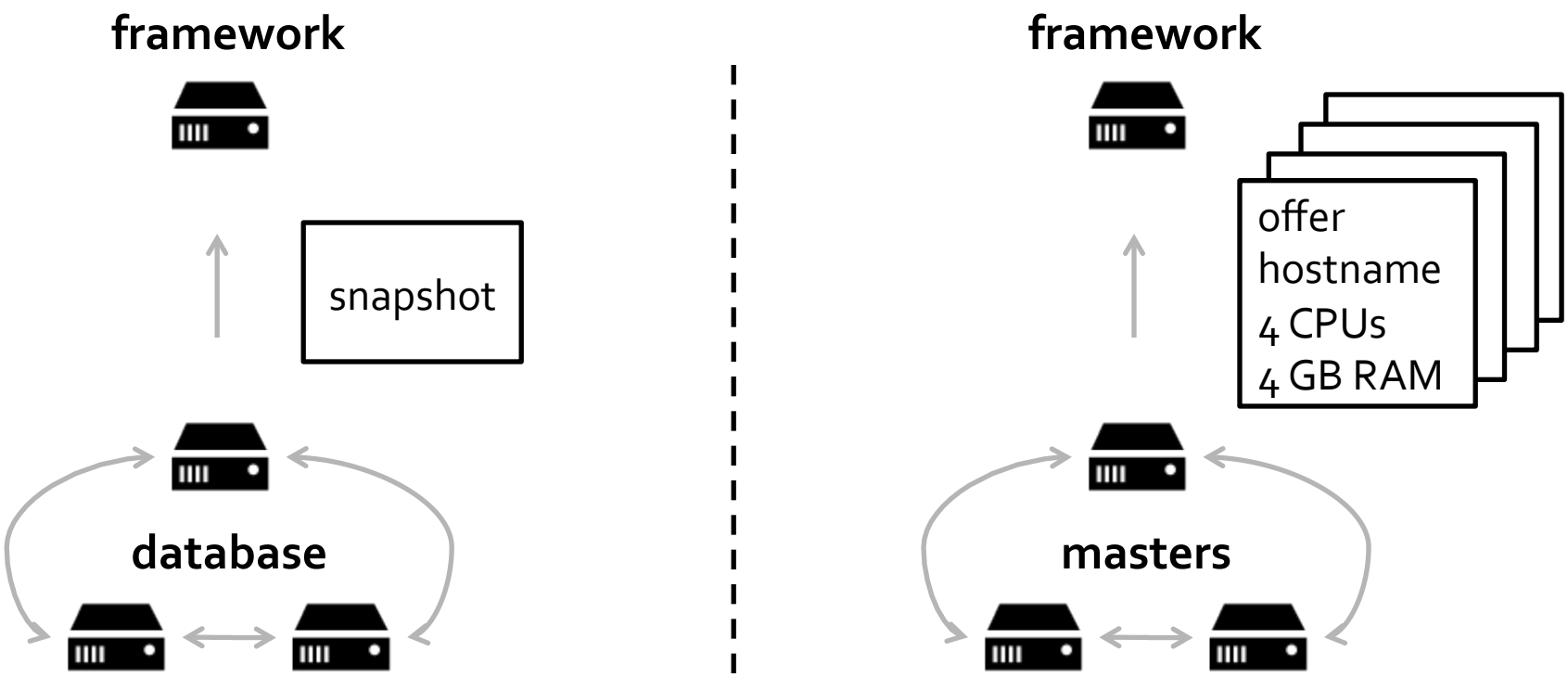
all offers *overlap* with one another, thus causing frameworks to "compete" first-come-first-served

optimistic offers

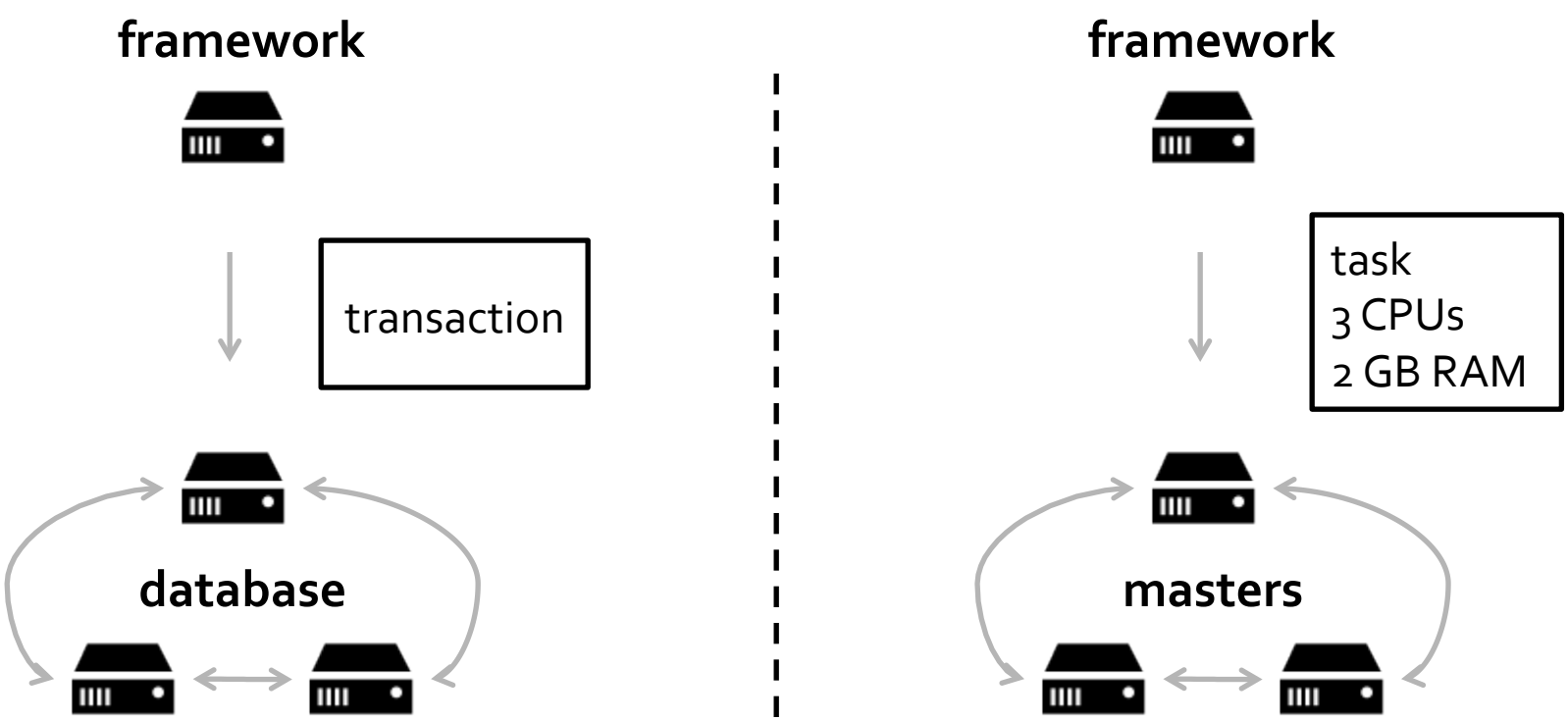


isomorphism:
Google Omega's snapshots and Mesos' optimistic offers

Omega and Mesos



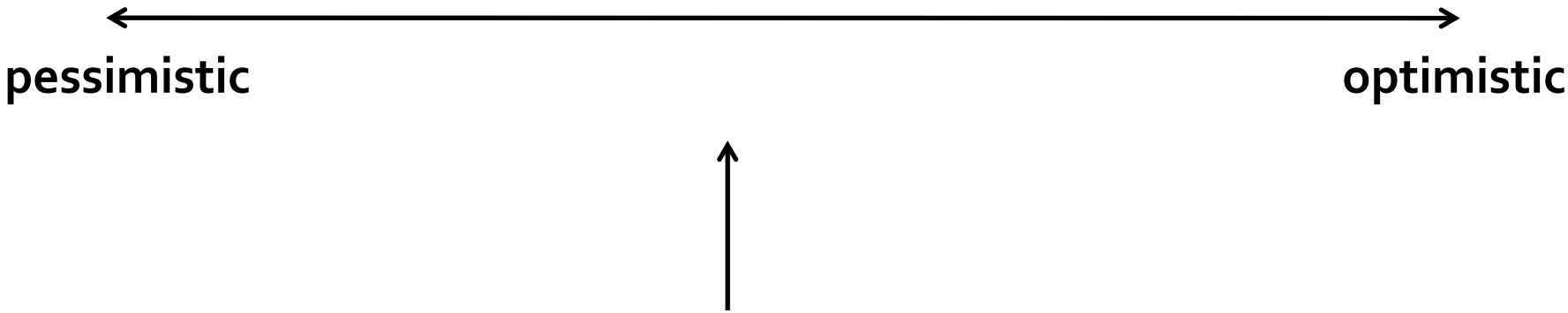
Omega and Mesos



Google's Omega: multi-level scheduler optimistic by default

Mesos:
multi-level scheduler
pessimistic by default;
optimistic via optimistic offers

concurrency control



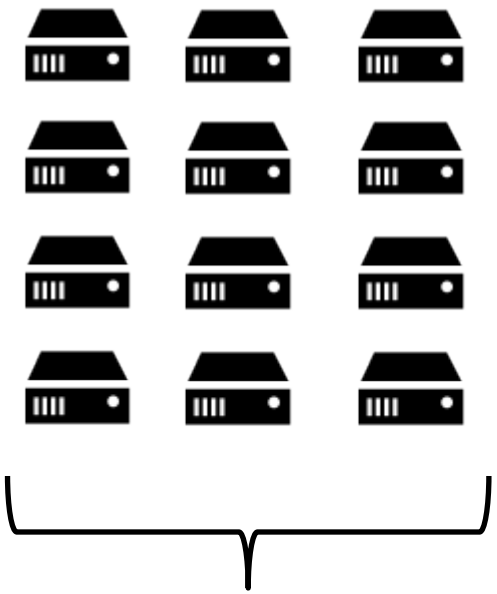
some offers *overlap* with one another, thus causing only *some* of the frameworks to "compete" first-come-first-served

agenda

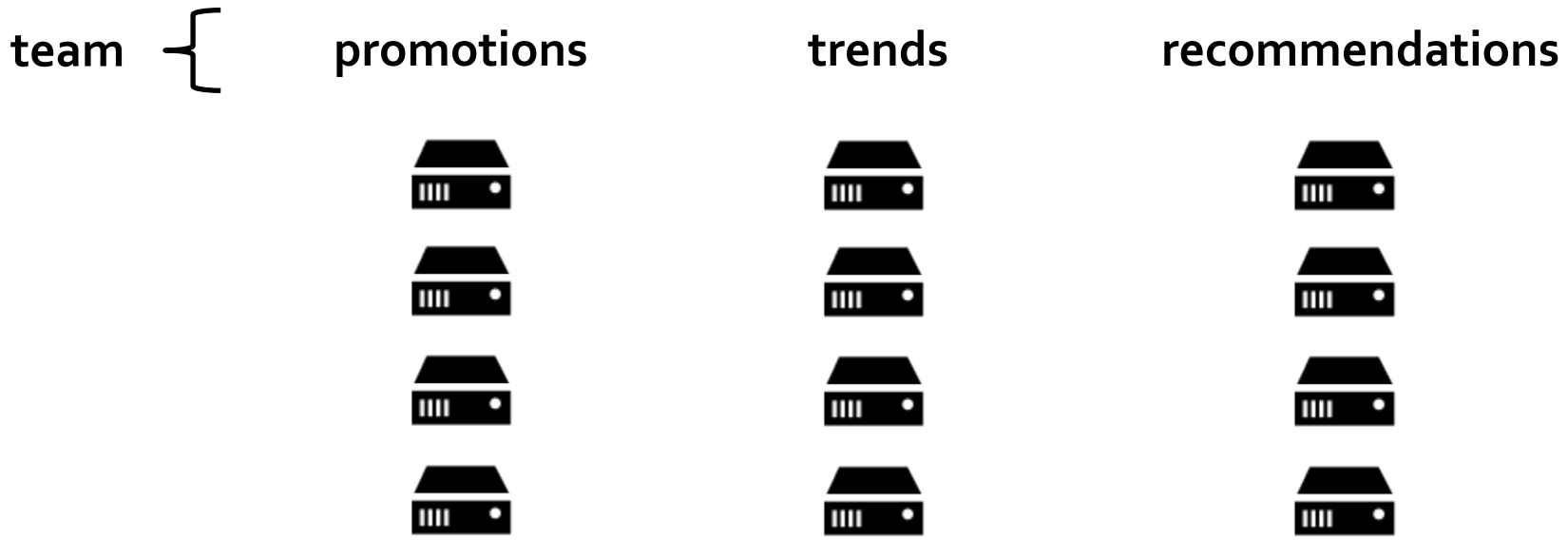
1. multi-level scheduling (scheduler activations)
2. intra-process multi-level scheduling (Lithe)
3. distributed multi-level scheduling (Mesos)
    1. optimistic multi-level scheduling
    2. Dominant Resource Fairness
    3. reservations and revocable resources
    4. persistent volumes
    5. inverse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

Mesos 1st-level scheduler allocates resources to frameworks using a fair-sharing algorithm we created called Dominant Resource Fairness (DRF)

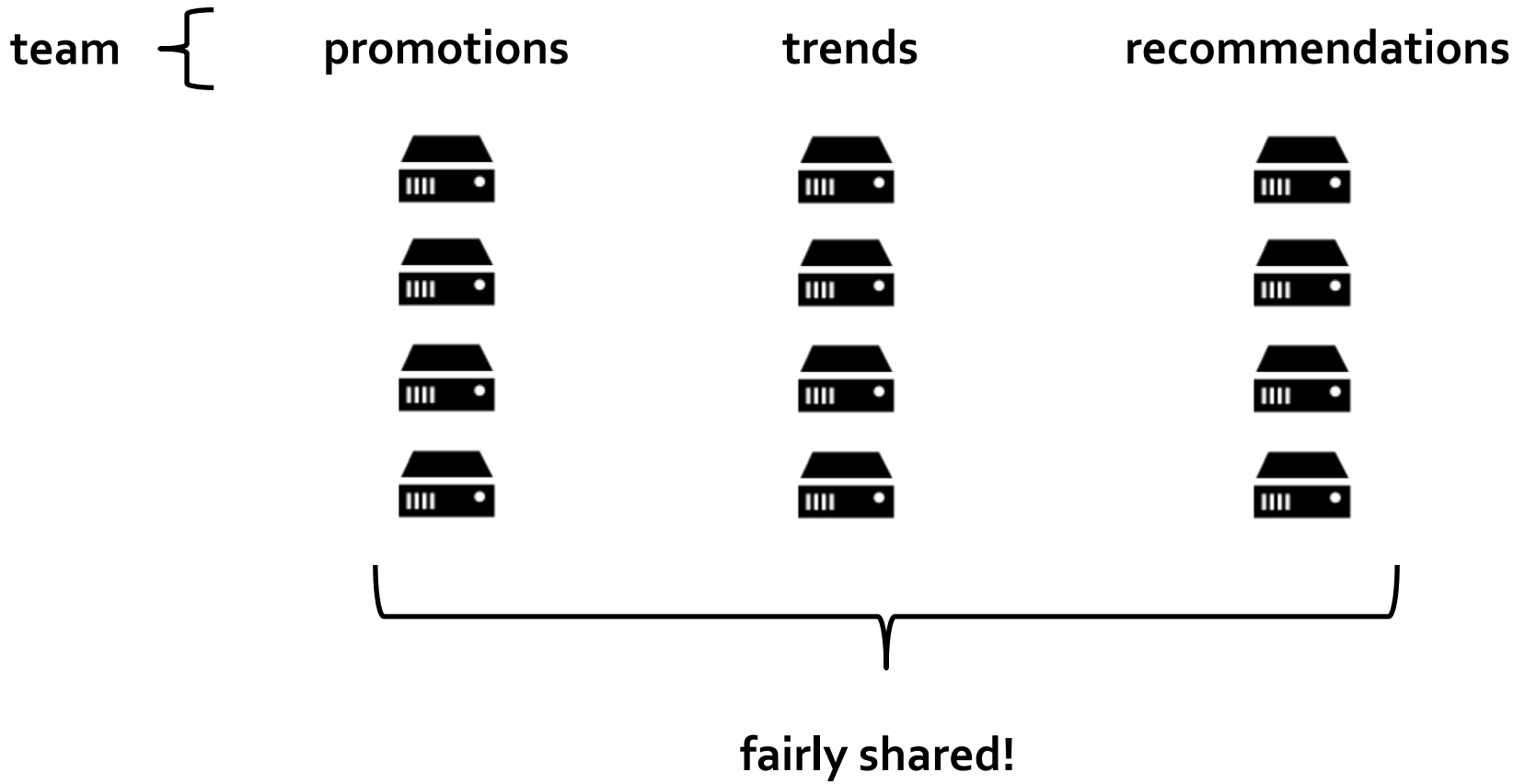
DRF, born of static partitioning



static partitioning across teams

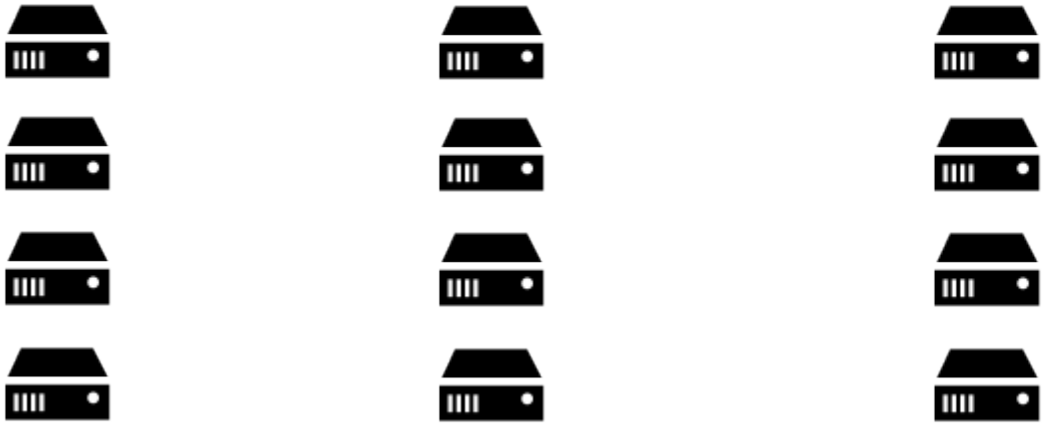


static partitioning across teams



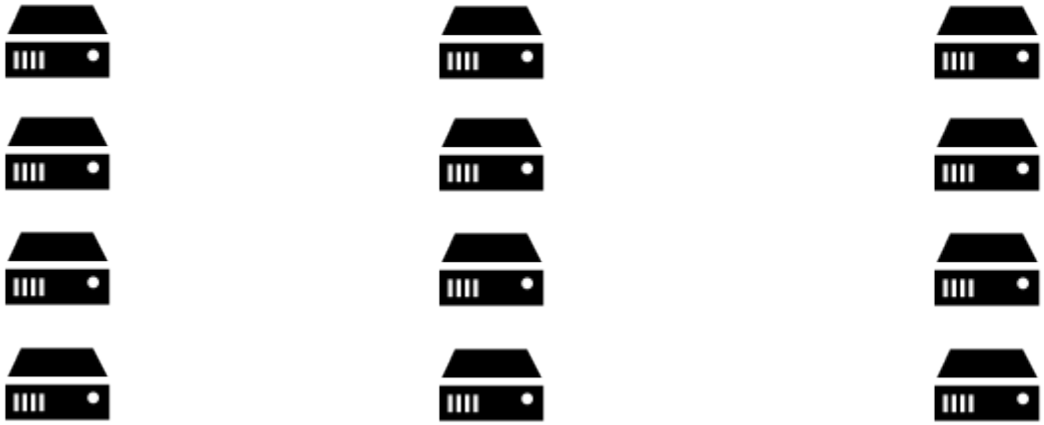
goal: fairly share the resources
without static partitioning

partition utilizations

team	promotions	trends	recommendations
			
utilization	45% CPU 100% RAM	75% CPU 100% RAM	100% CPU 50% RAM

observation: a <u>dominant</u> resource *bottlenecks* each team from running any more jobs/tasks

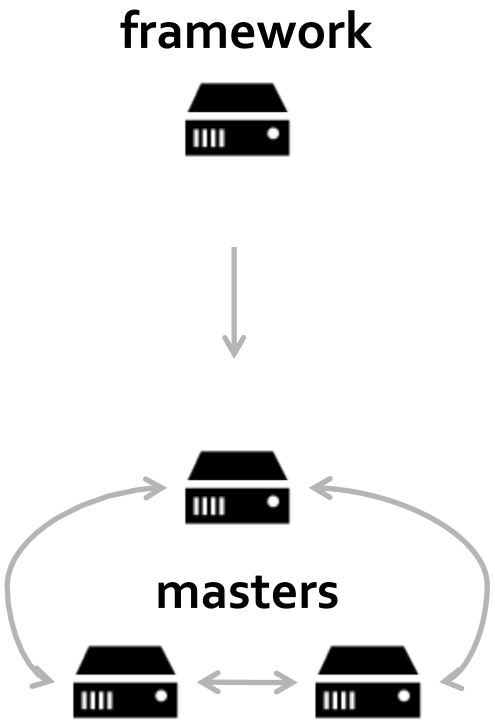
dominant resource bottlenecks

team	promotions	trends	recommendations
			
utilization	45% CPU 100% RAM	75% CPU 100% RAM	100% CPU 50% RAM
bottleneck	RAM	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!

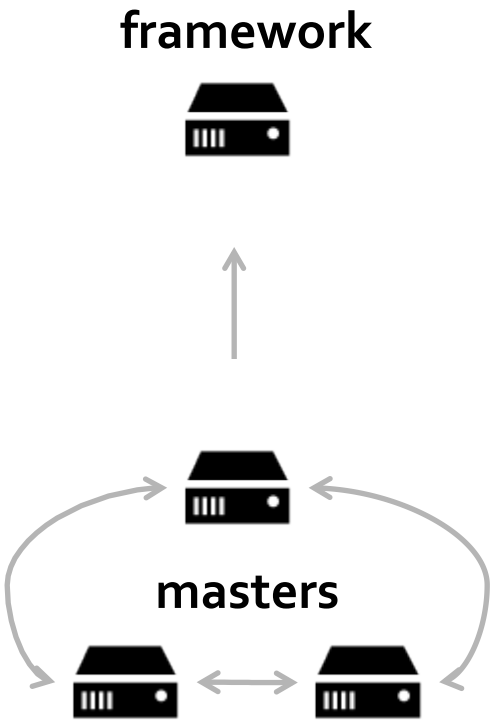
… if my team gets at least 1/N of my dominant resource I will do no worse than if I had my own cluster, but I might do better when resources are available!

DRF in Mesos



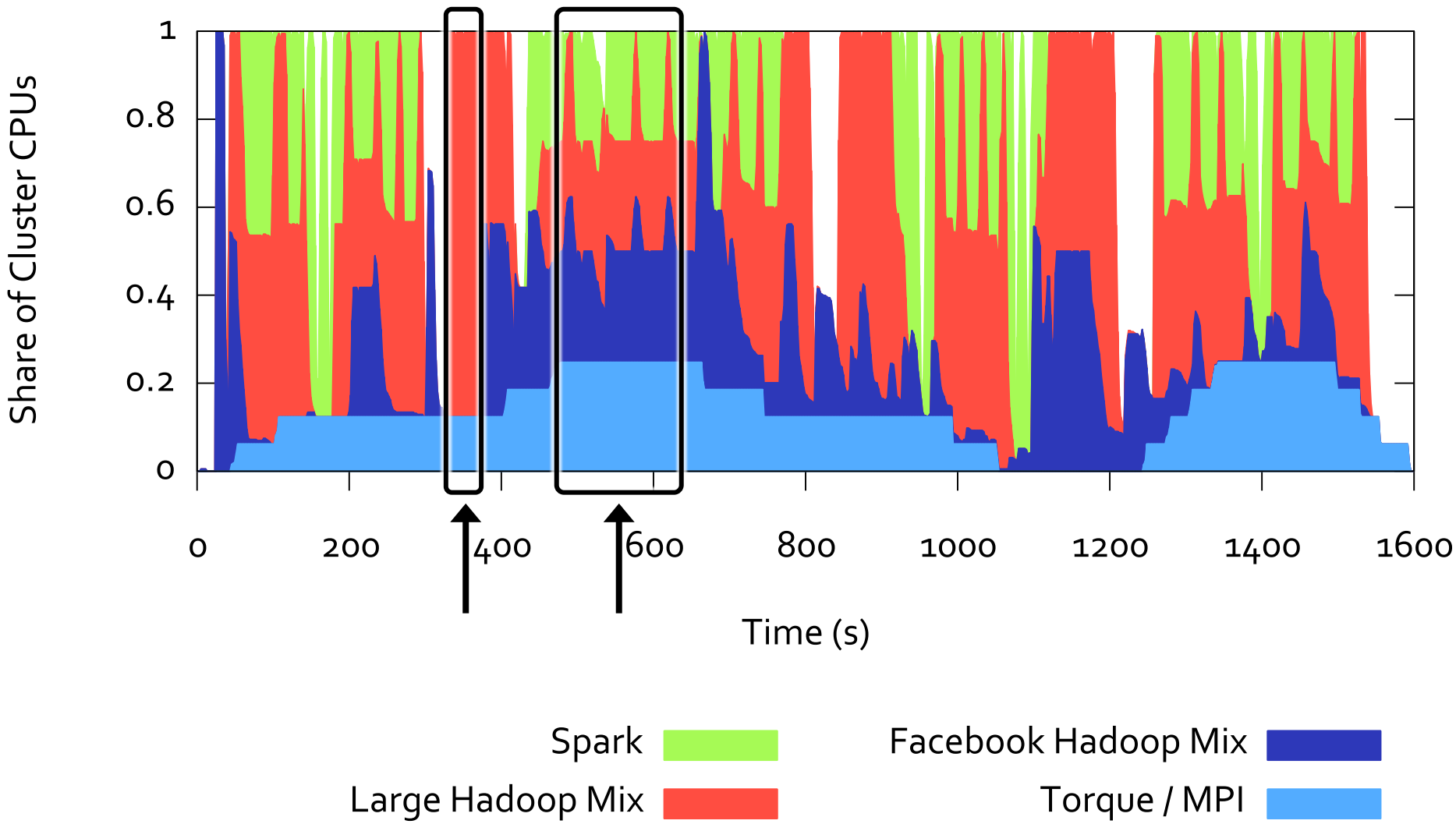
① frameworks specify a ***role*** when they register (i.e., the team to charge for used resources)

DRF in Mesos



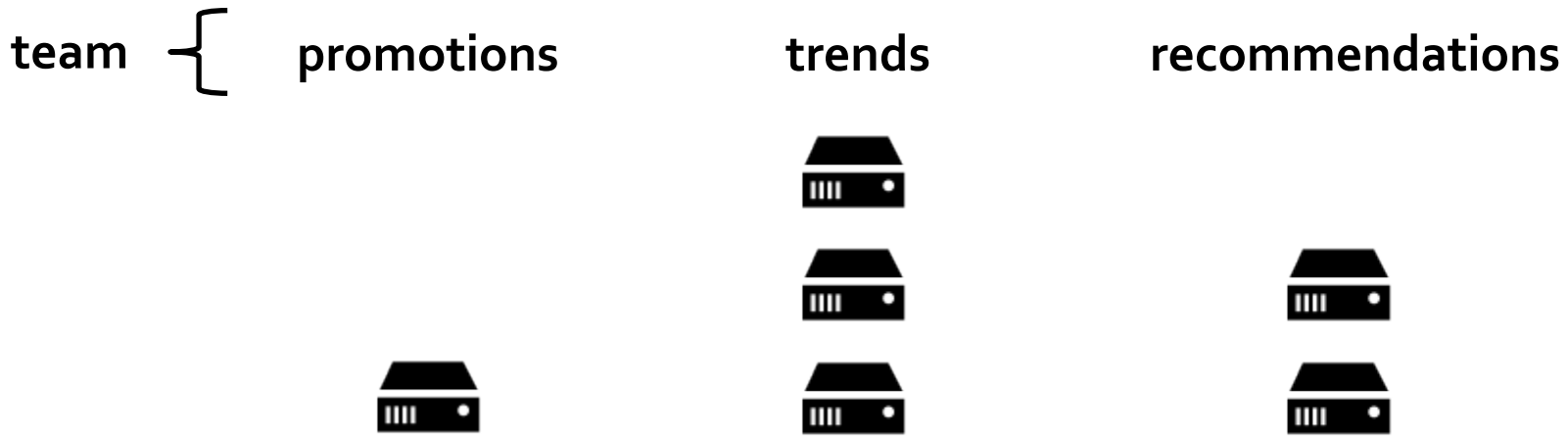
① frameworks specify a ***role*** when they register (i.e., the team to charge for used resources)

② master calculates each role's dominant resource (dynamically) and allocates appropriately

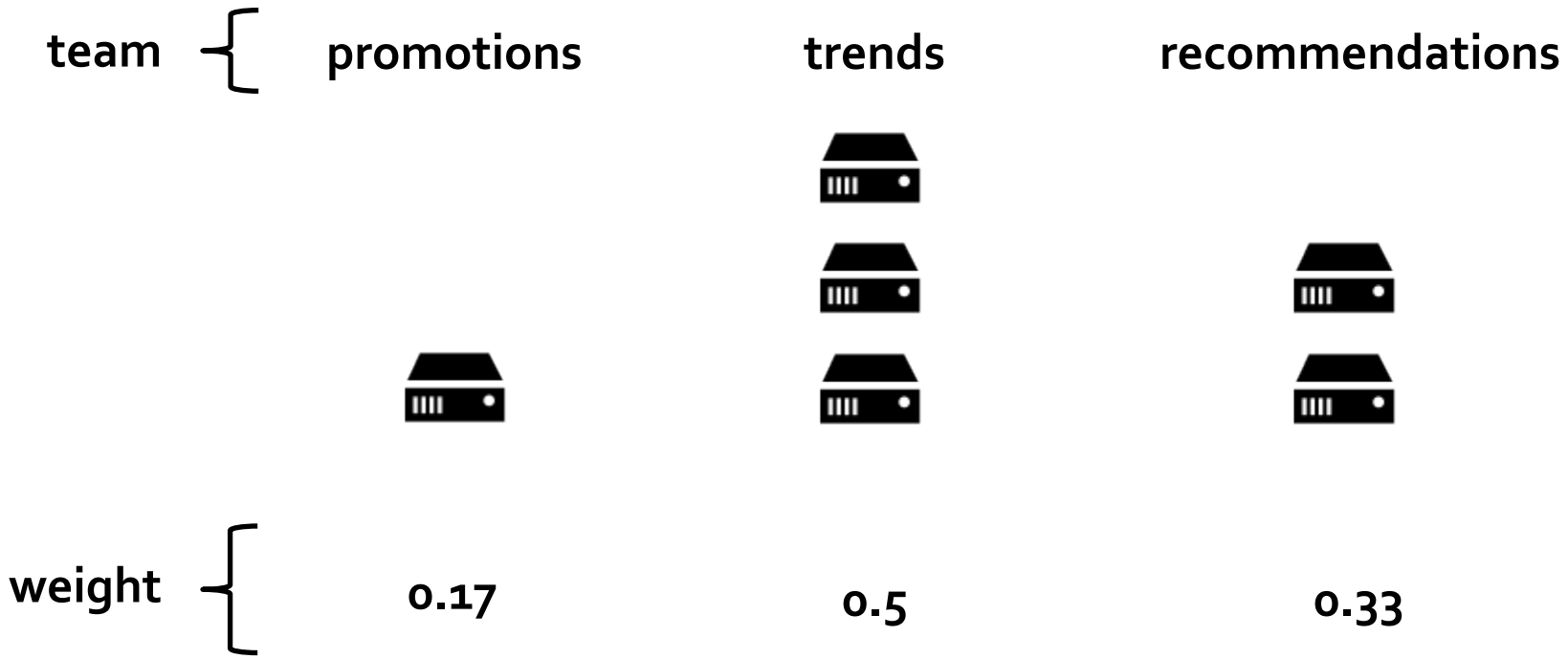

Share of Cluster CPUs
1
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
1400
1600
Time (s)
Spark
Large Hadoop Mix
Facebook Hadoop Mix
Torque / MPI

in practice,
***fair* sharing is insufficient**

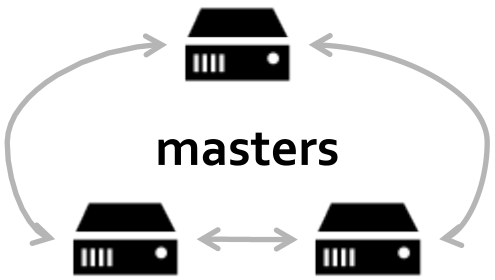
weighted fair sharing



weighted fair sharing



Mesos implements weighted DRF



masters can be configured with weights *per role*

resource allocation decisions incorporate the weights to determine dominant fair shares

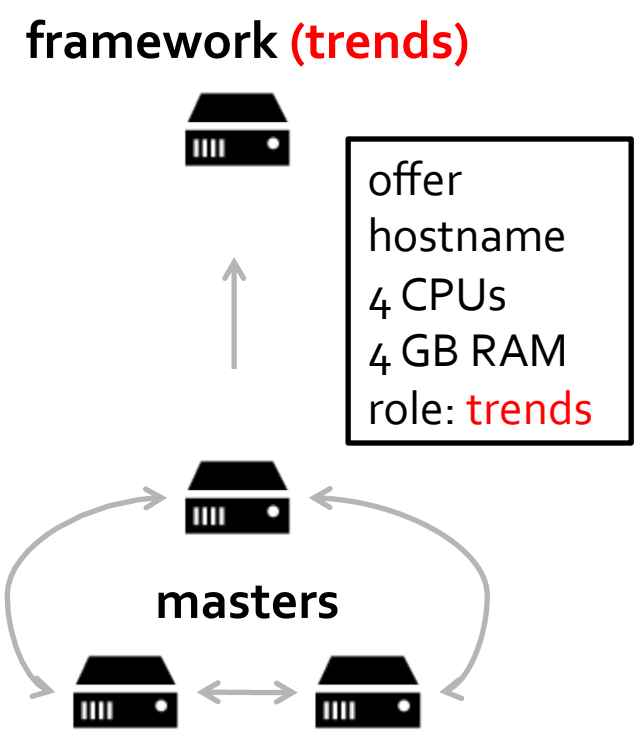
agenda

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

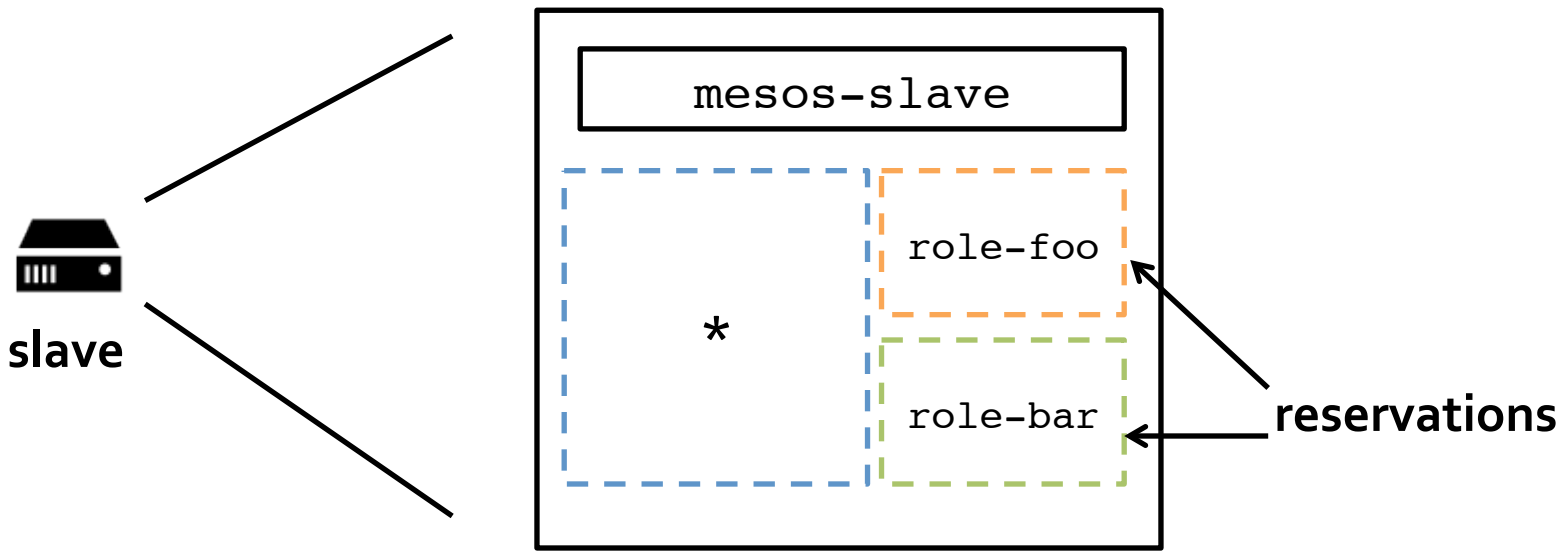
resource *reservations*



resources on individual slaves can be reserved for particular roles

resource offers include the reservation role (if any)

reservations



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

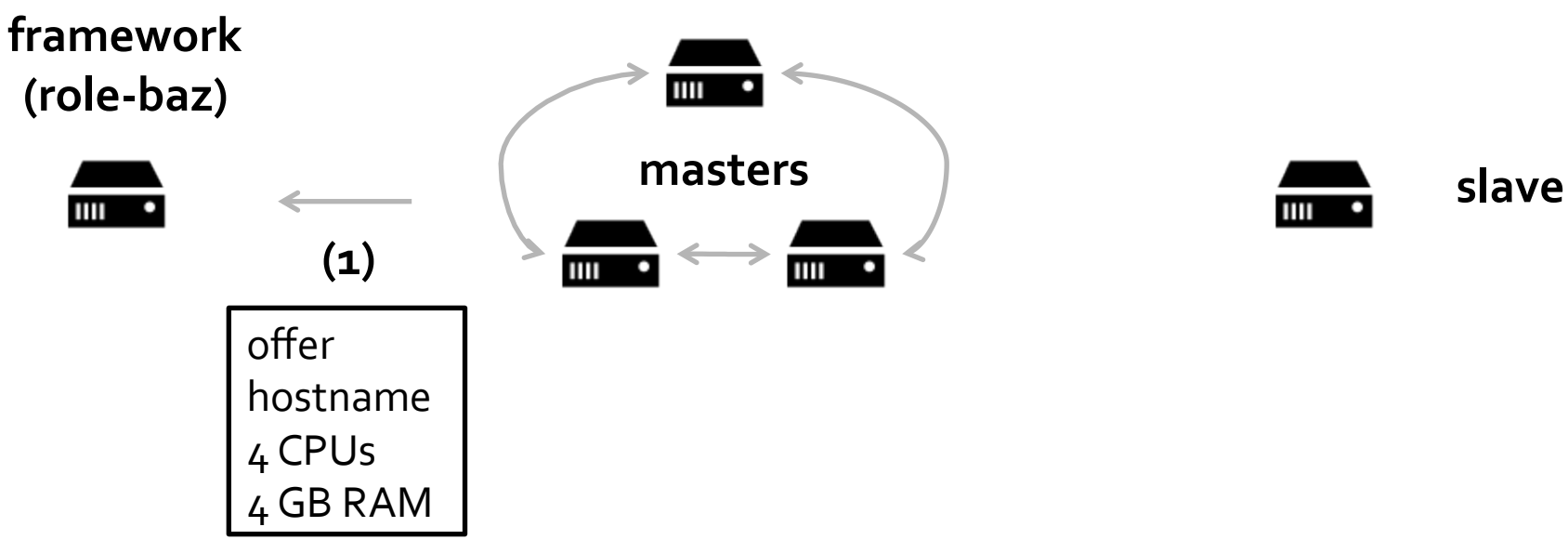
+ strong guarantees

- set up by an operator when starting slave

- immutable (must drain/restart the slave)

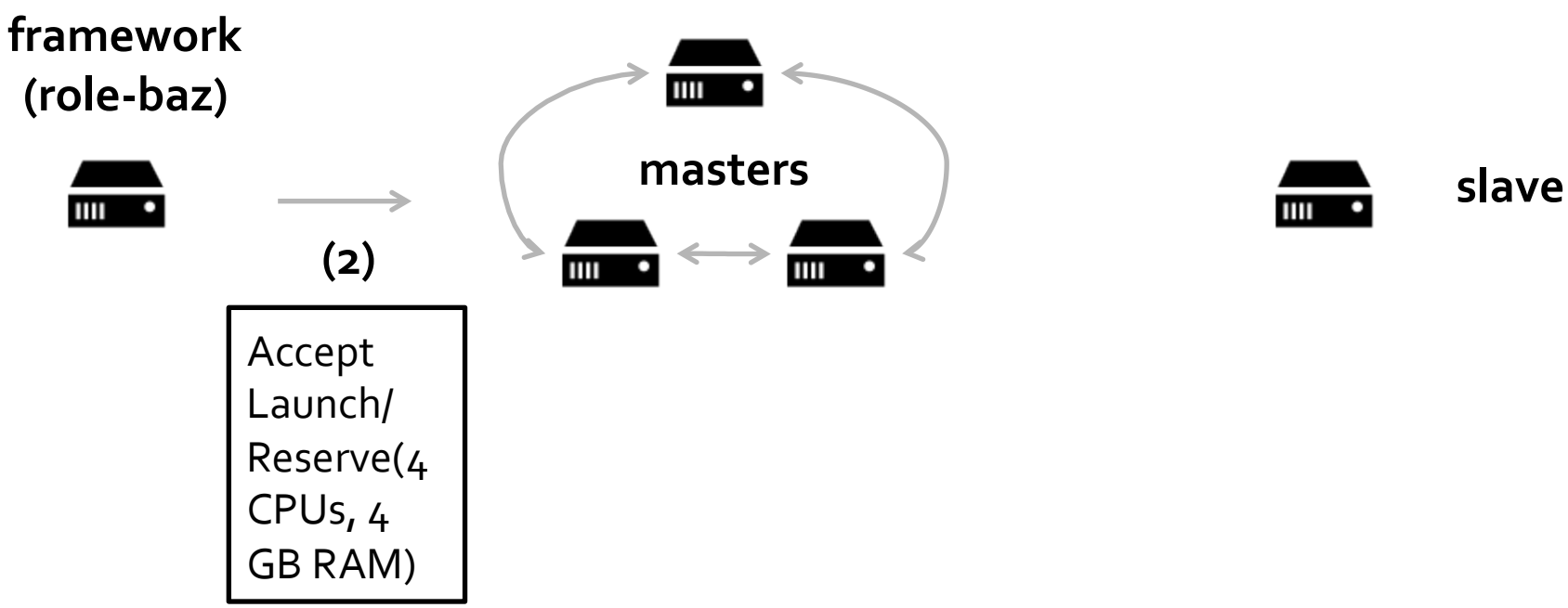
dynamic reservations

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



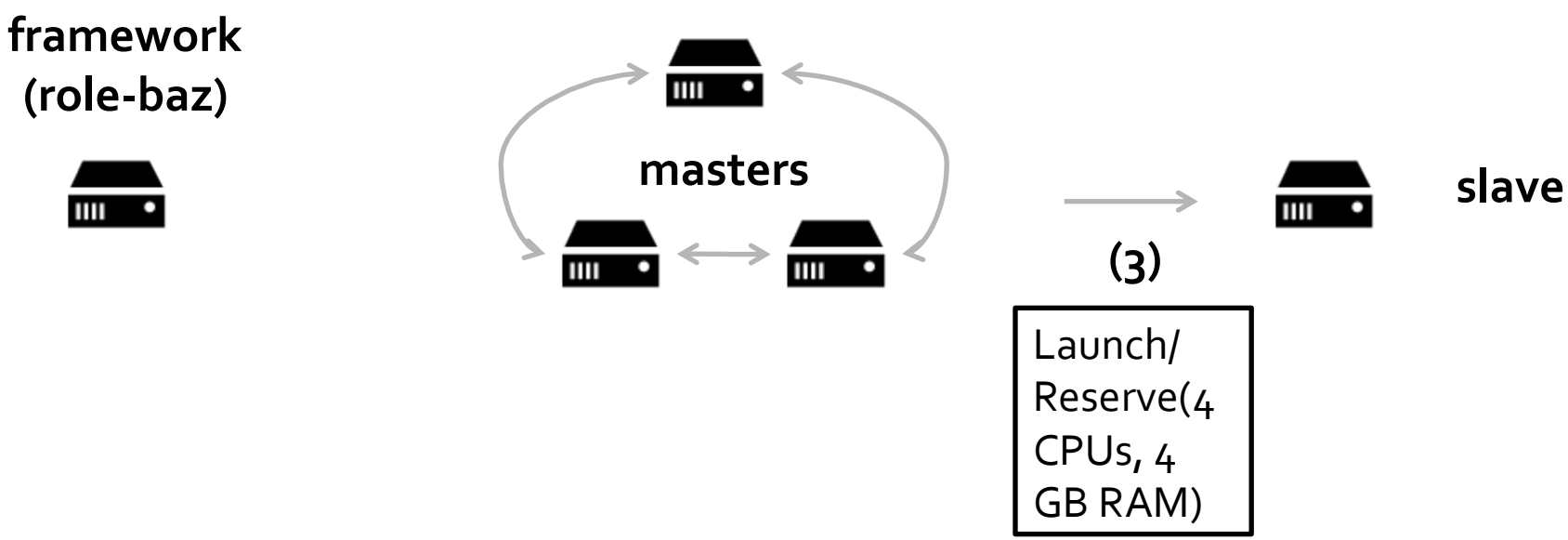
dynamic reservations

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

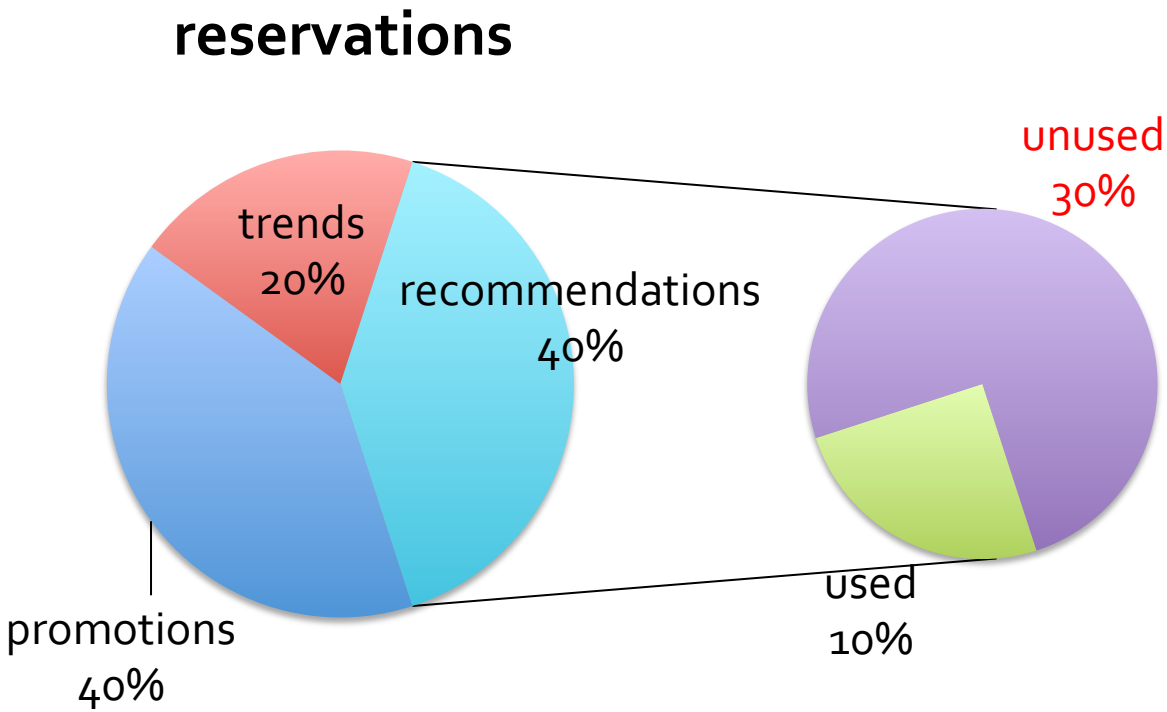


dynamic reservations

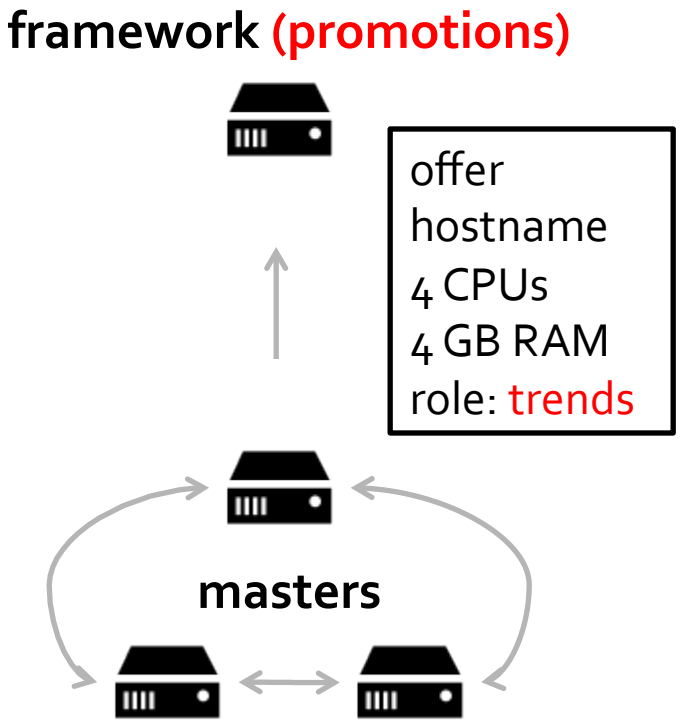
framework scheduler *reserves* resources at runtime when it *accepts* an offer (allocation)



reservations provide guarantees, but at the cost of utilization

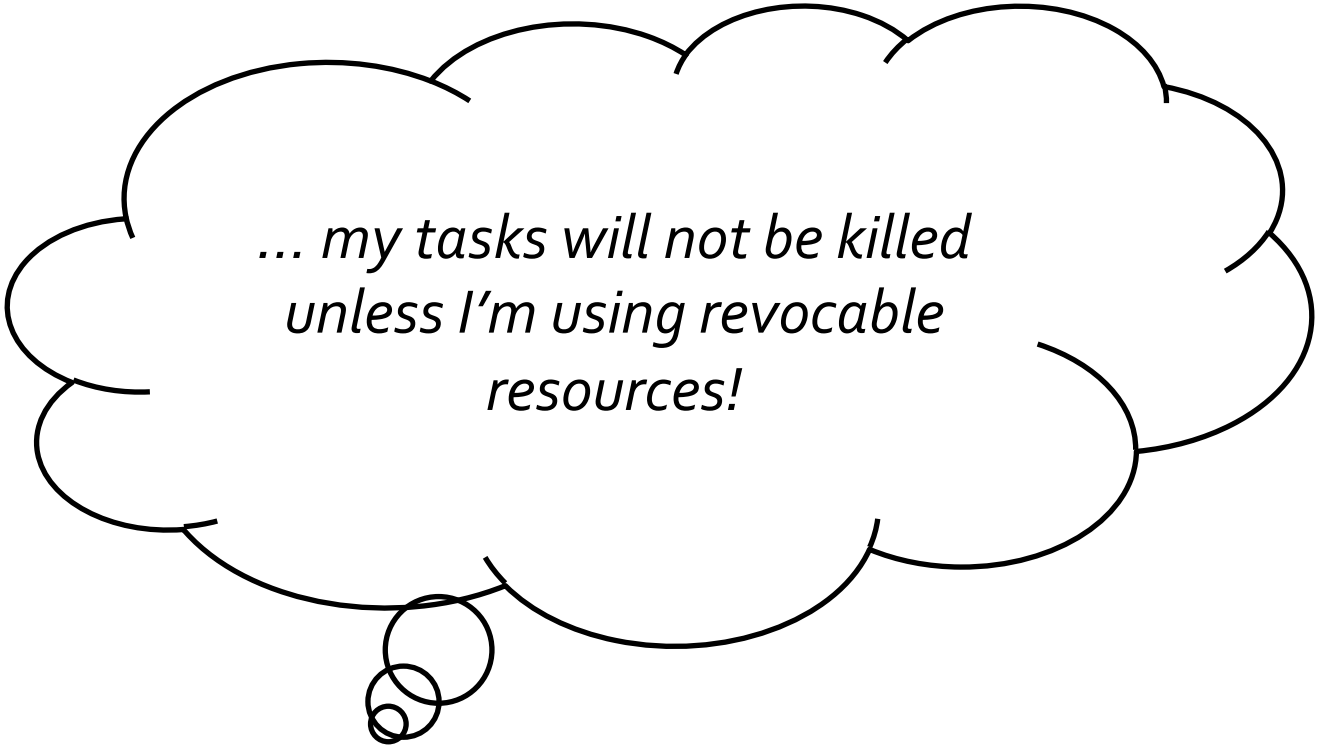


revocable resources

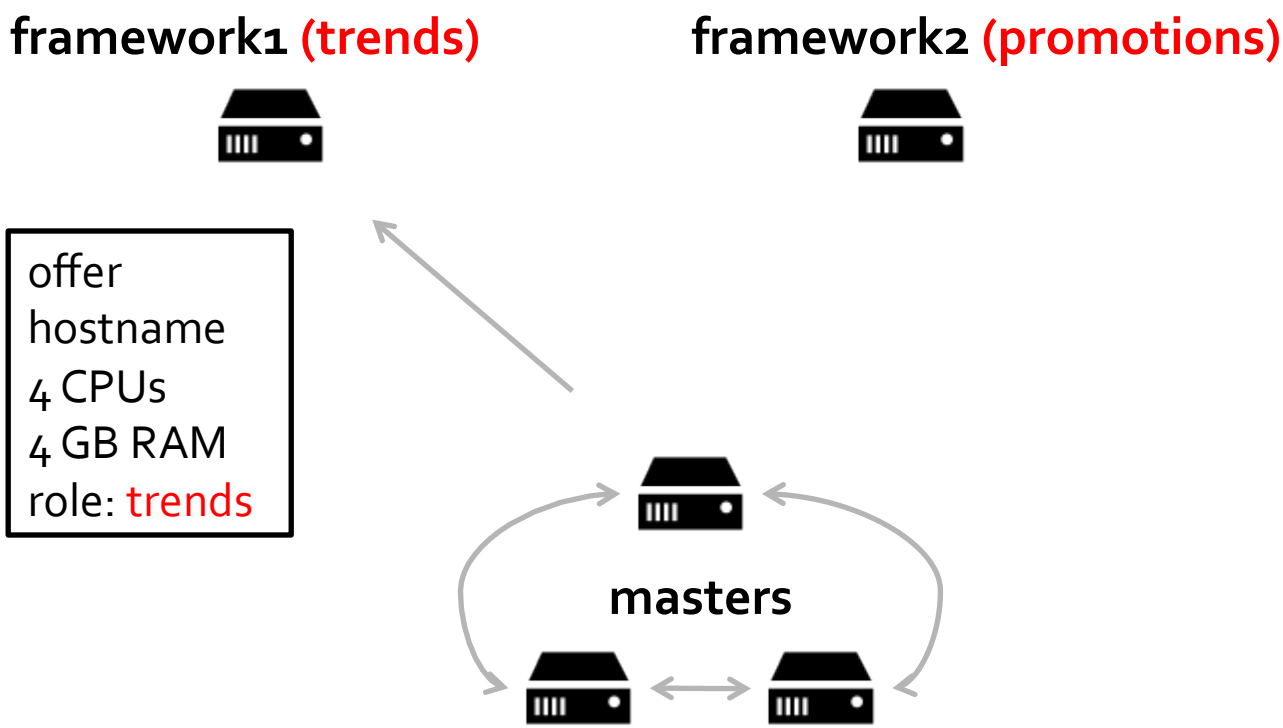


resources that are reserved for another role and thus may be *revoked* at any time

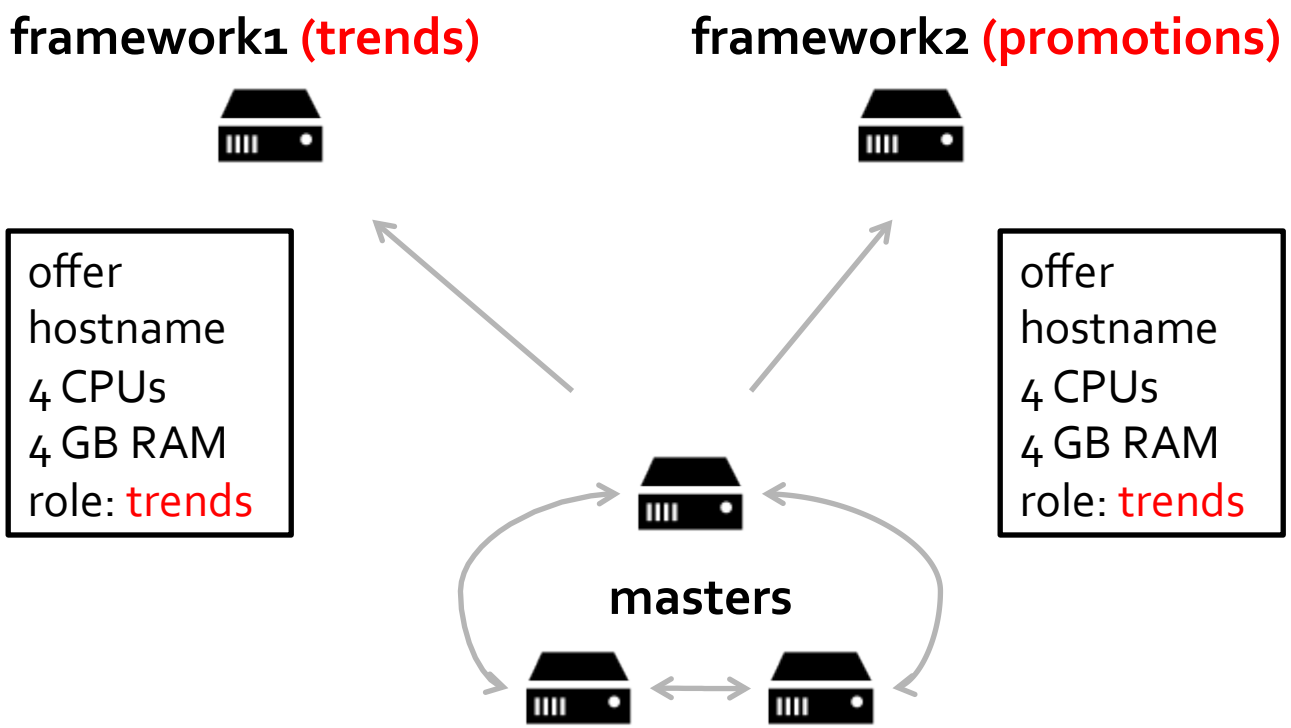
preemption via revocation



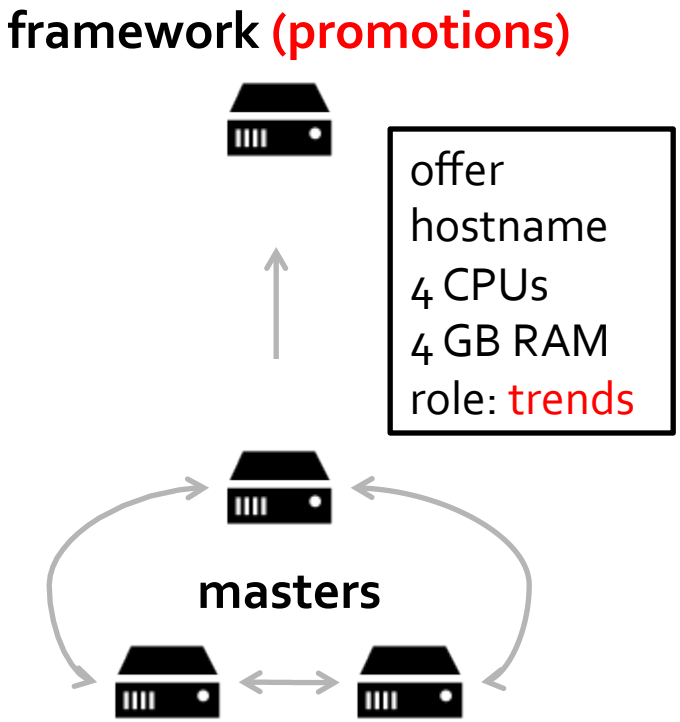
optimistic offers of revocable resources



optimistic offers of revocable resources



oversubscription via revocable resources



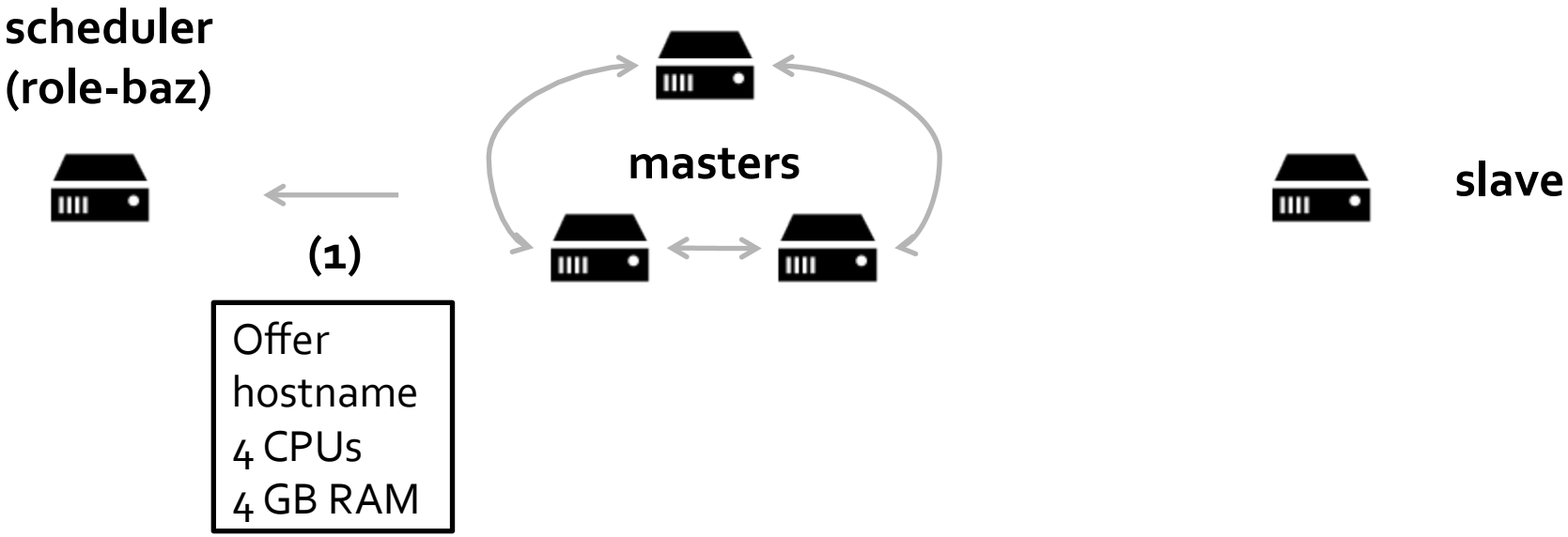
oversubscribe resources by allocating unused resources as revocable!

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

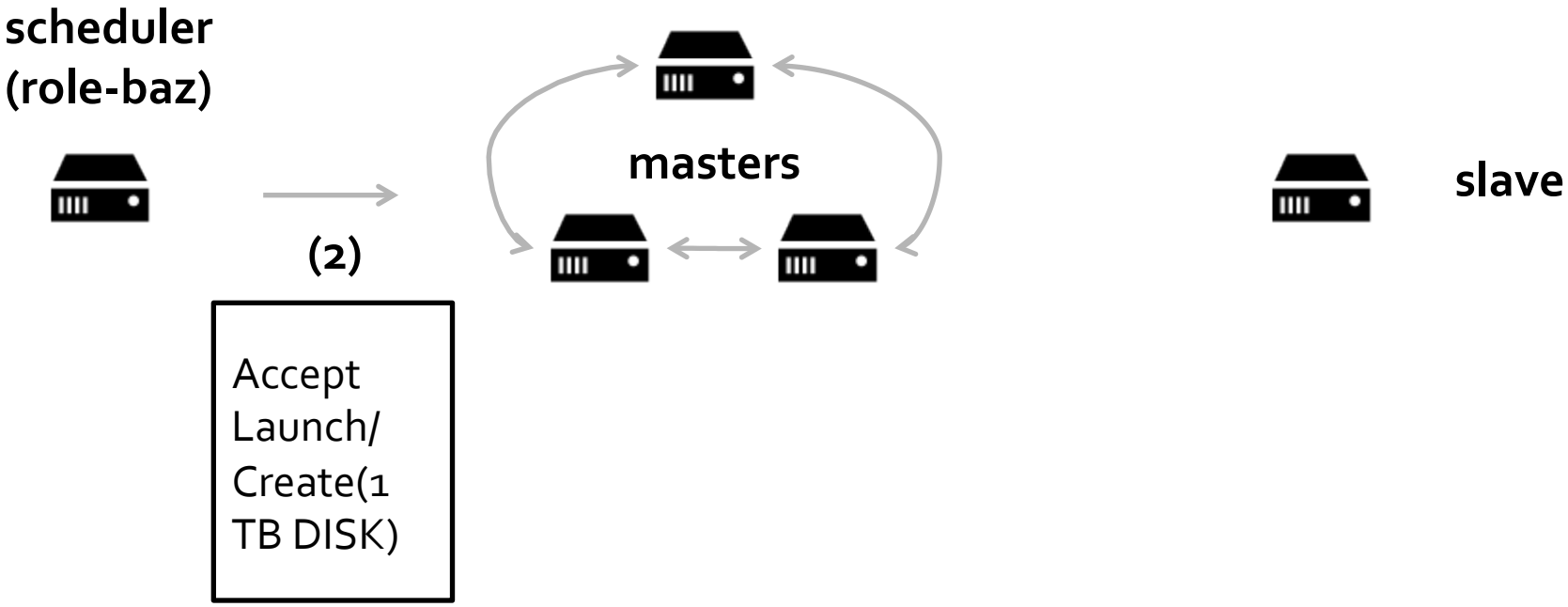
persistent volumes

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



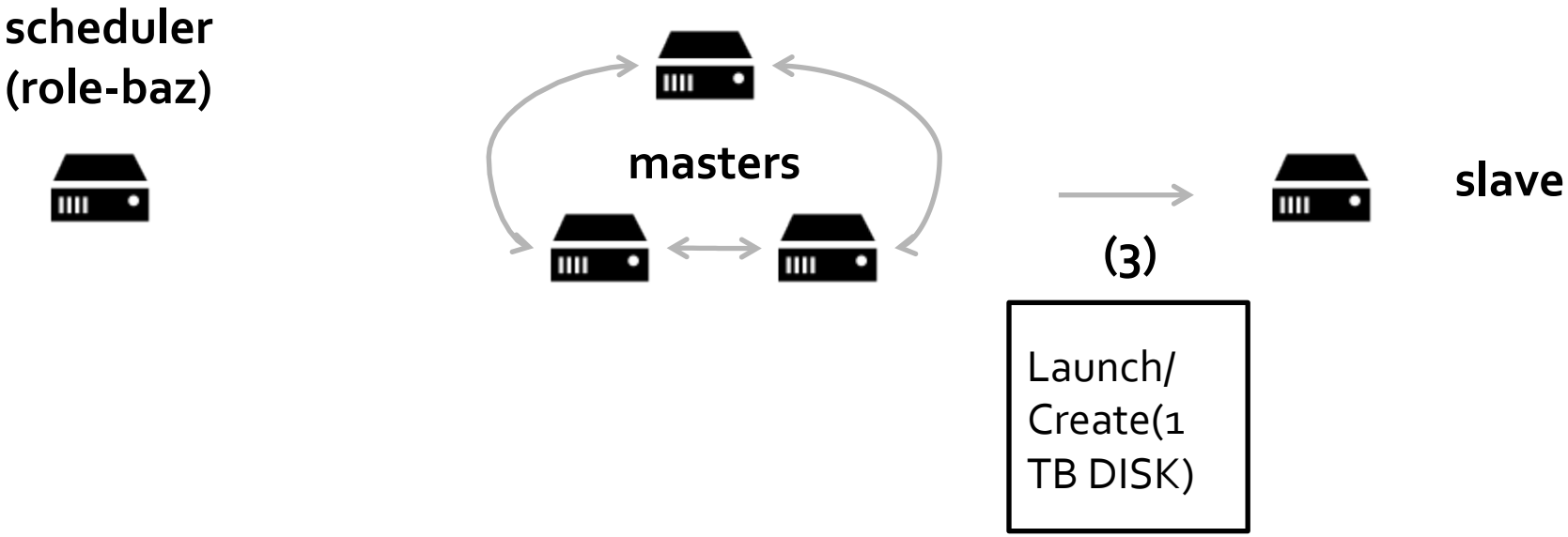
persistent volumes

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



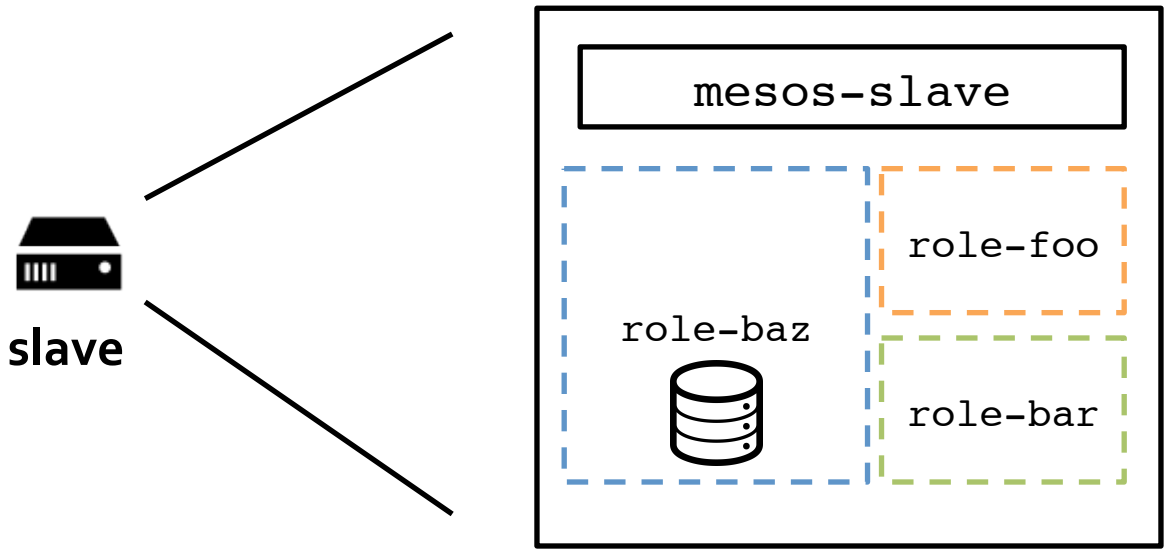
persistent volumes

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



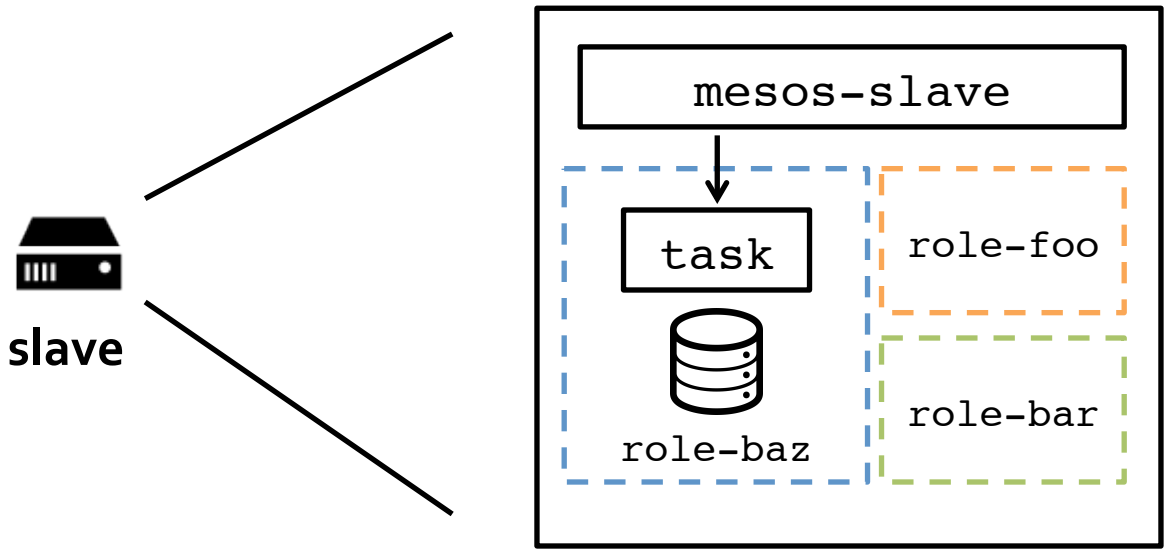
persistent volumes

volumes are created before launching any tasks or executors



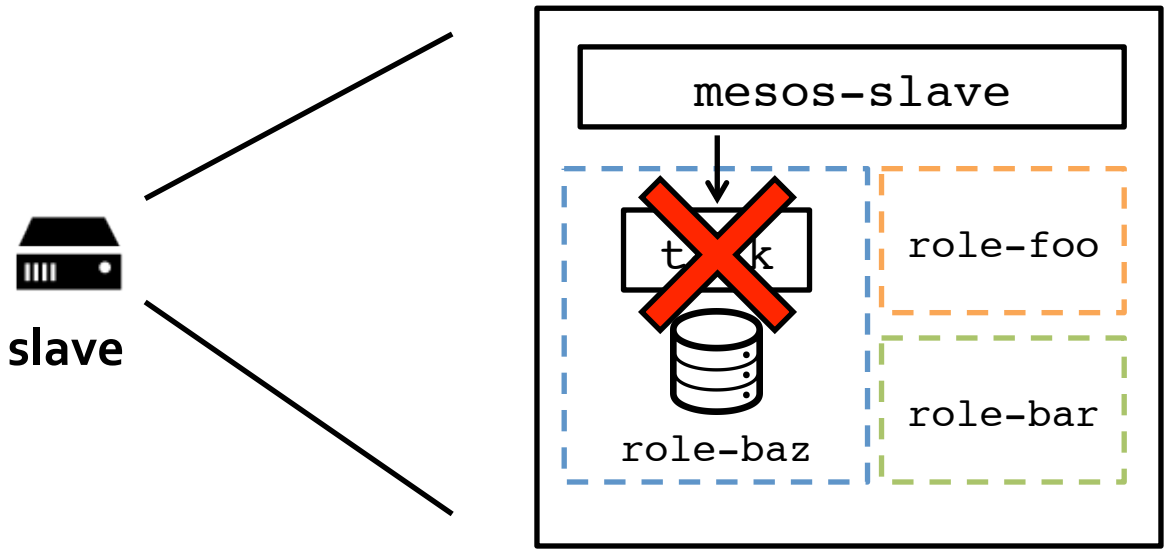
persistent volumes

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



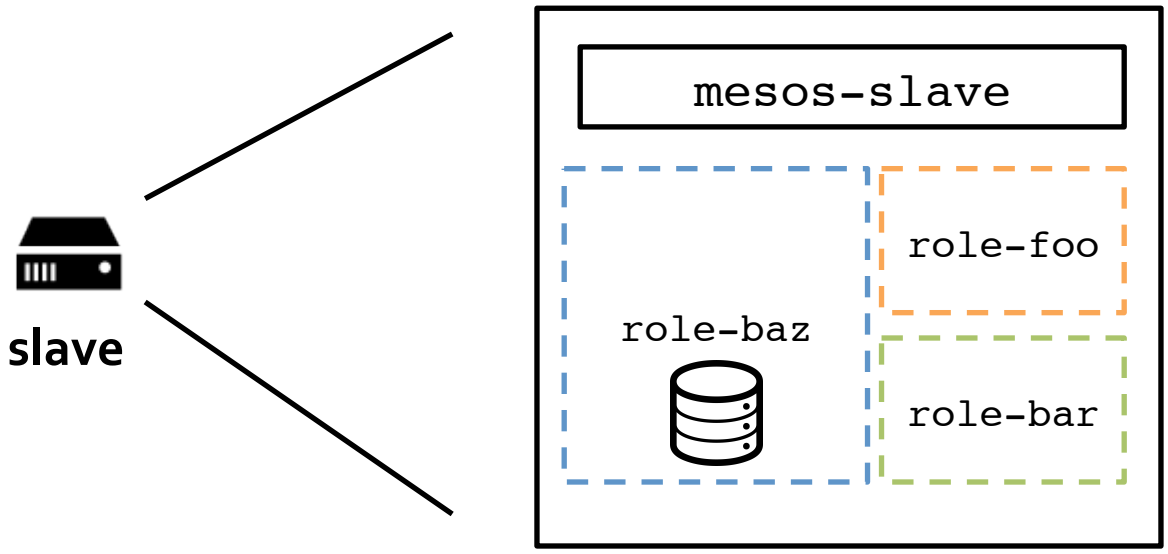
persistent volumes

volumes persist even after task or executor terminate!



persistent volumes

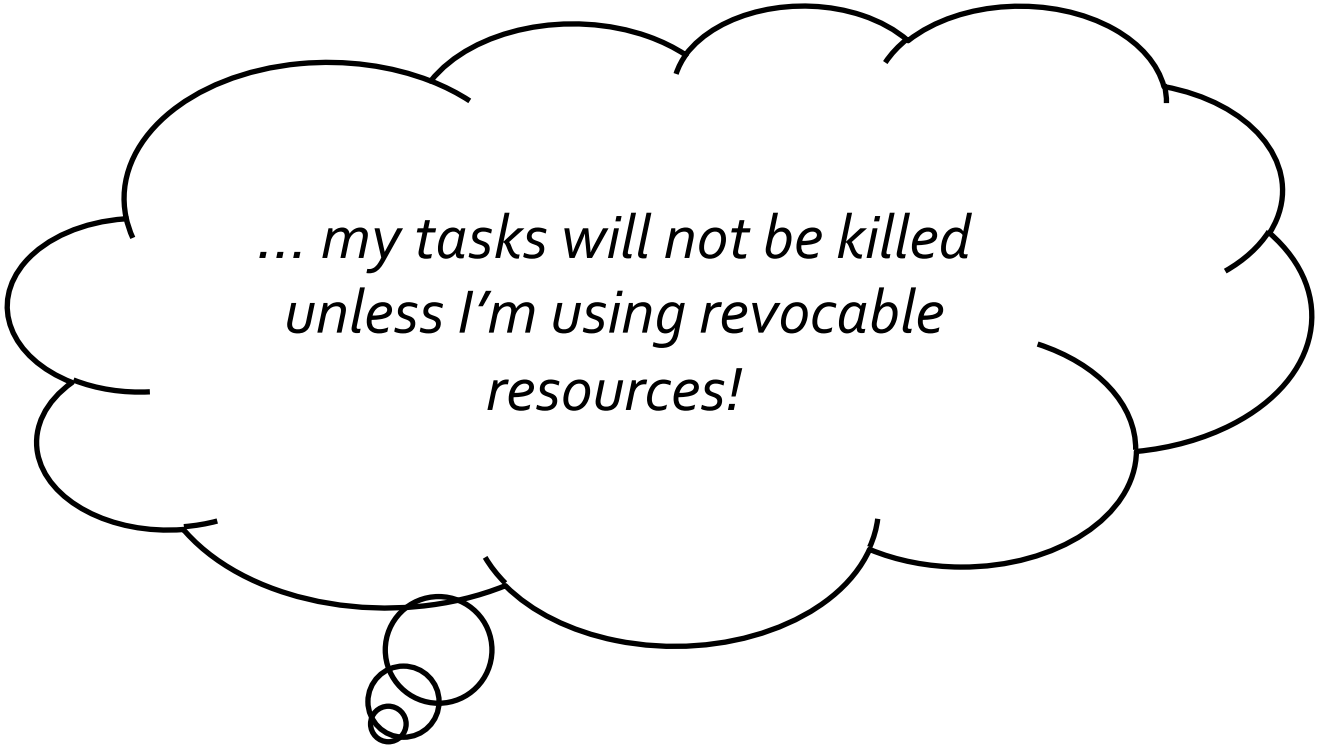
volumes persist even after task or executor terminate!



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”



revocation “guarantee”

what about when:

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

② maintenance

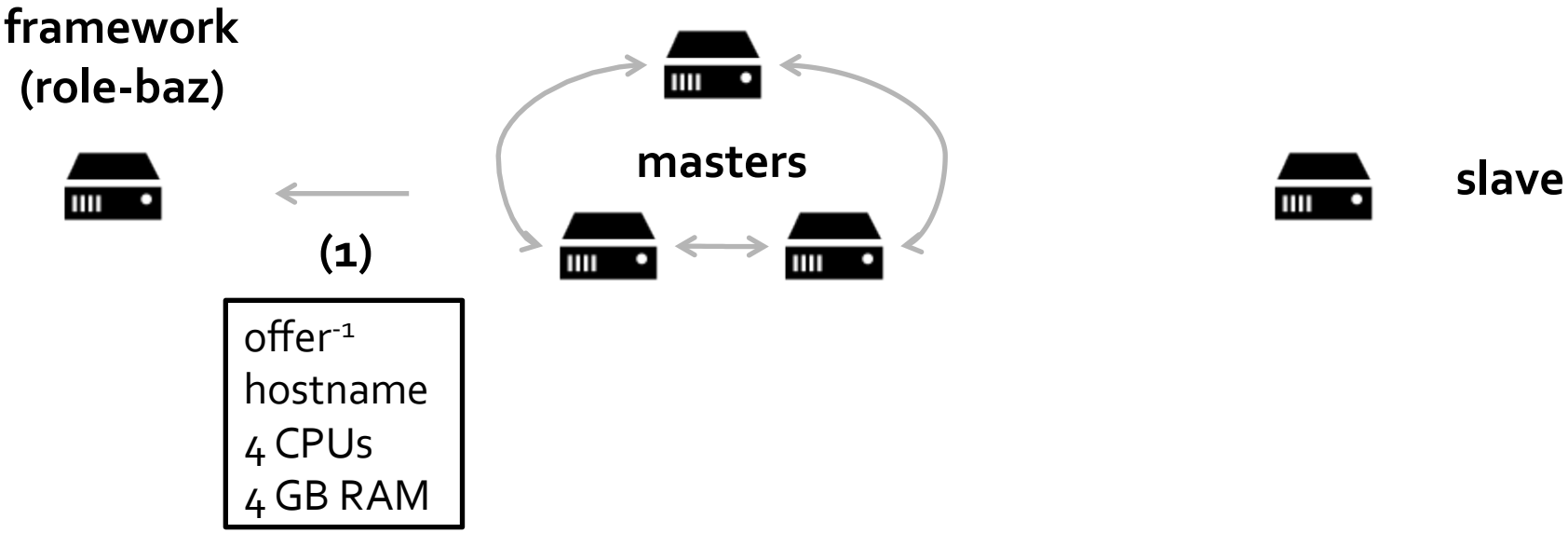
mechanism for deallocation

possible solution: introduce failures to deallocate resources

but why not communicate explicitly!?

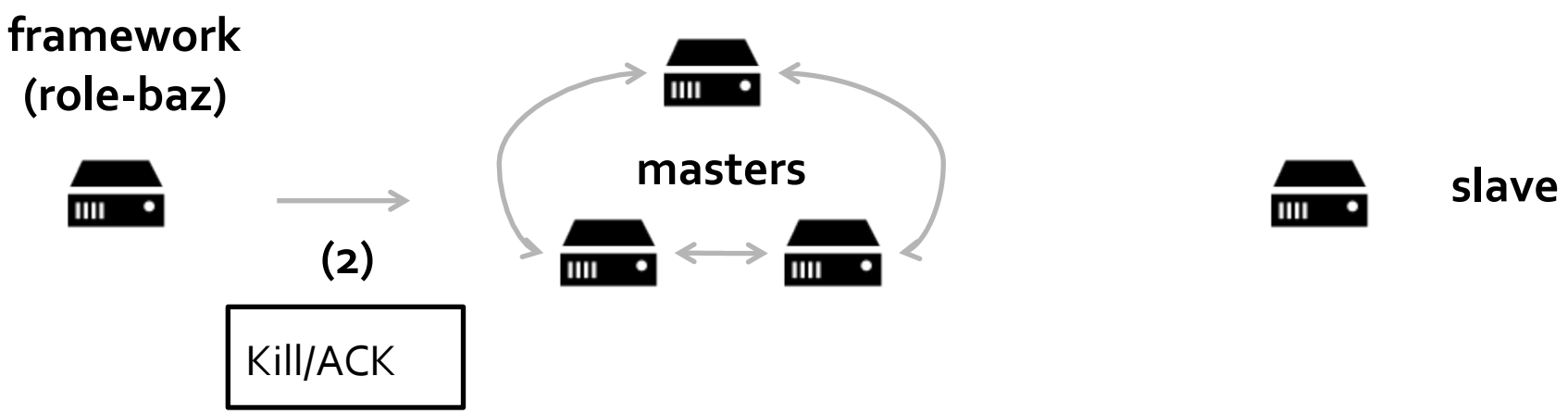
inverse offer

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

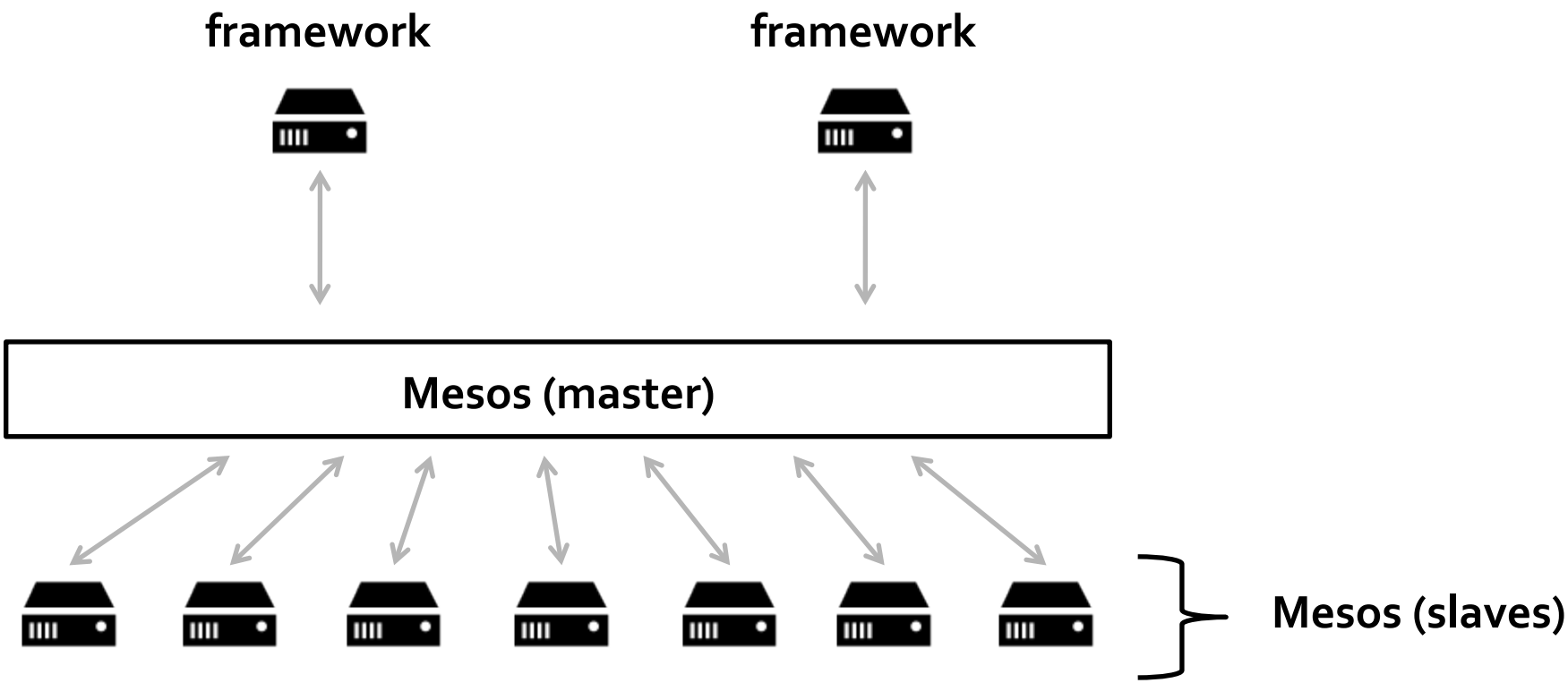
reservations +

persistent volumes +

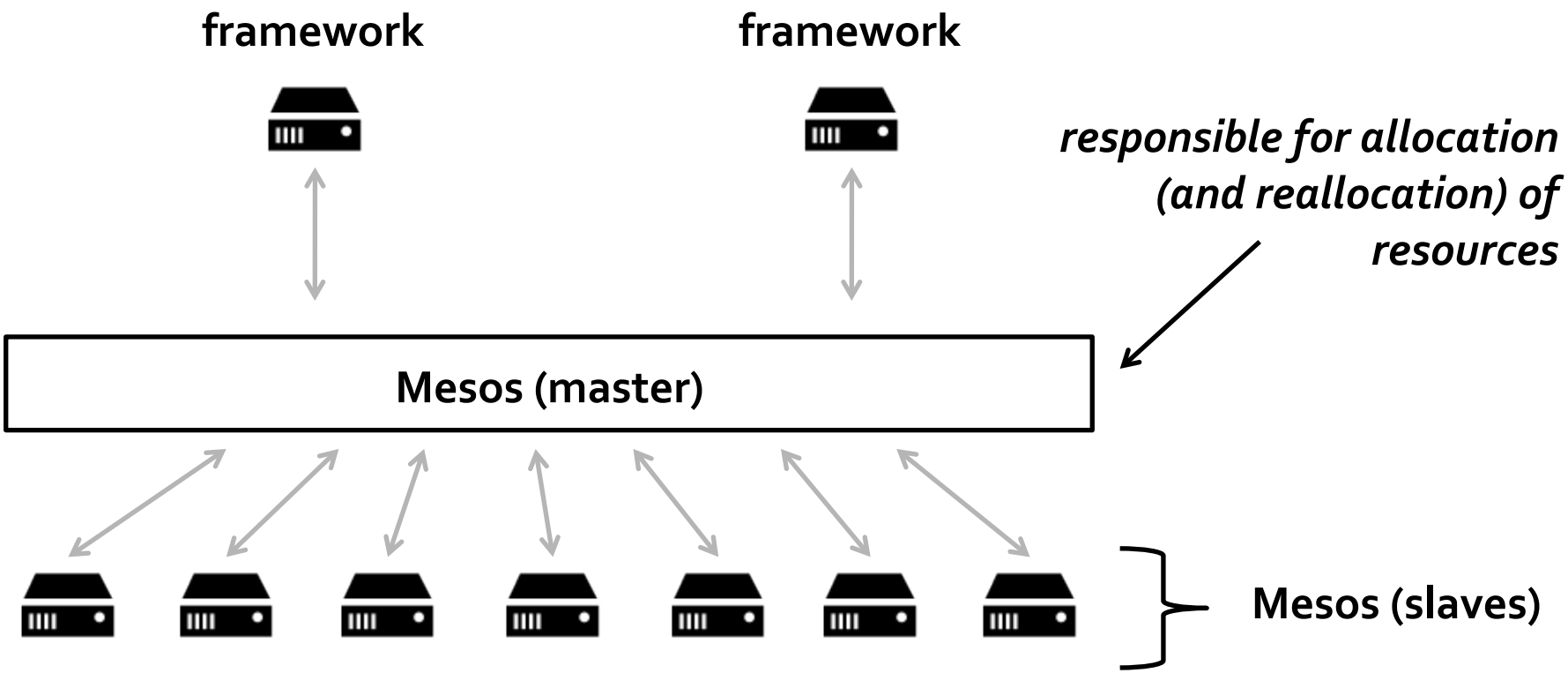
inverse offers =

long-lived stateful frameworks!

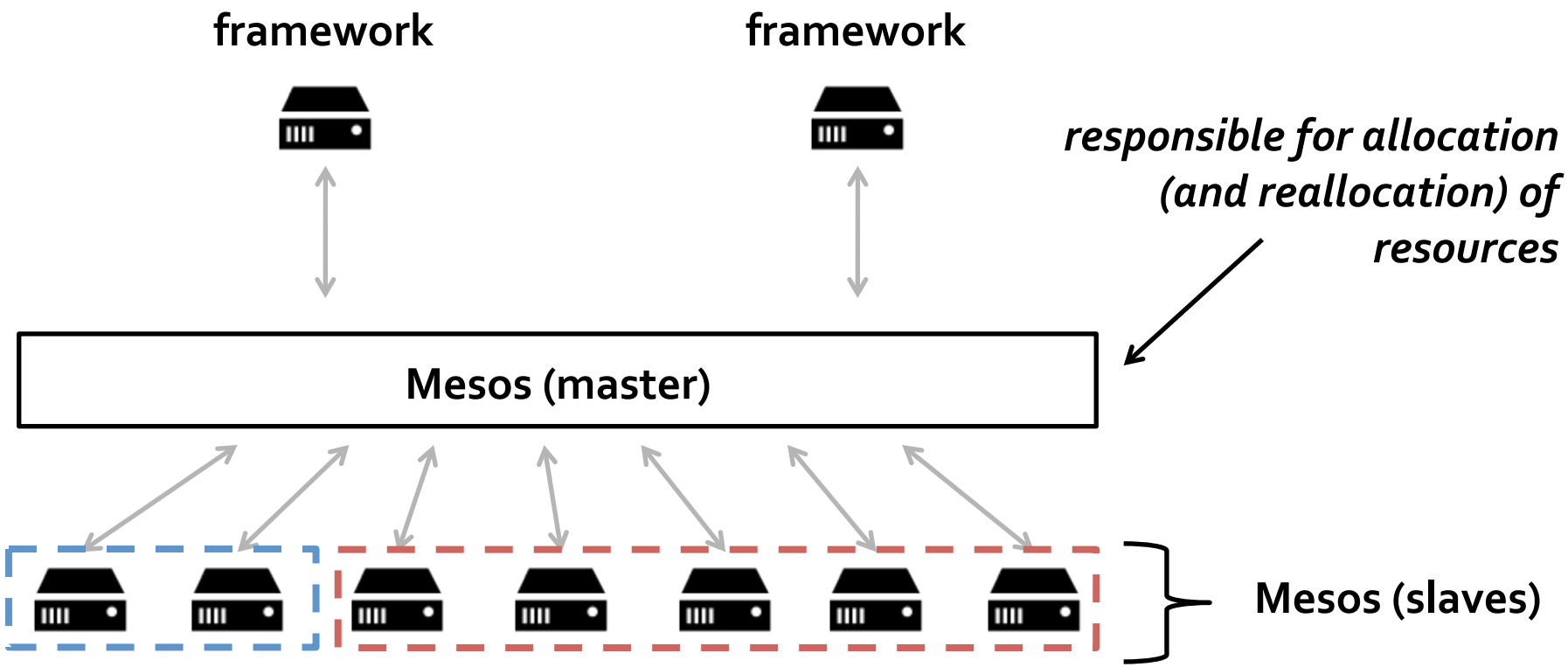
Mesos: level of abstraction



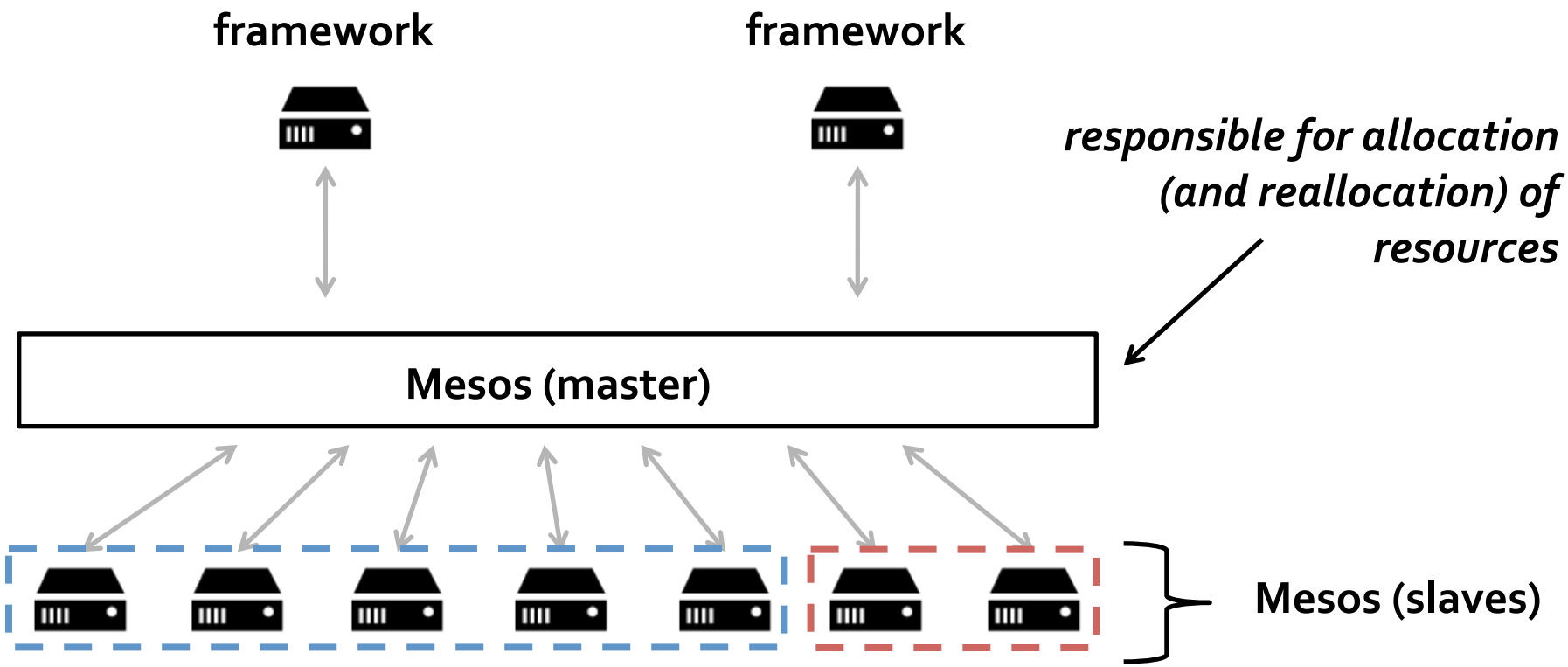
Mesos: level of abstraction



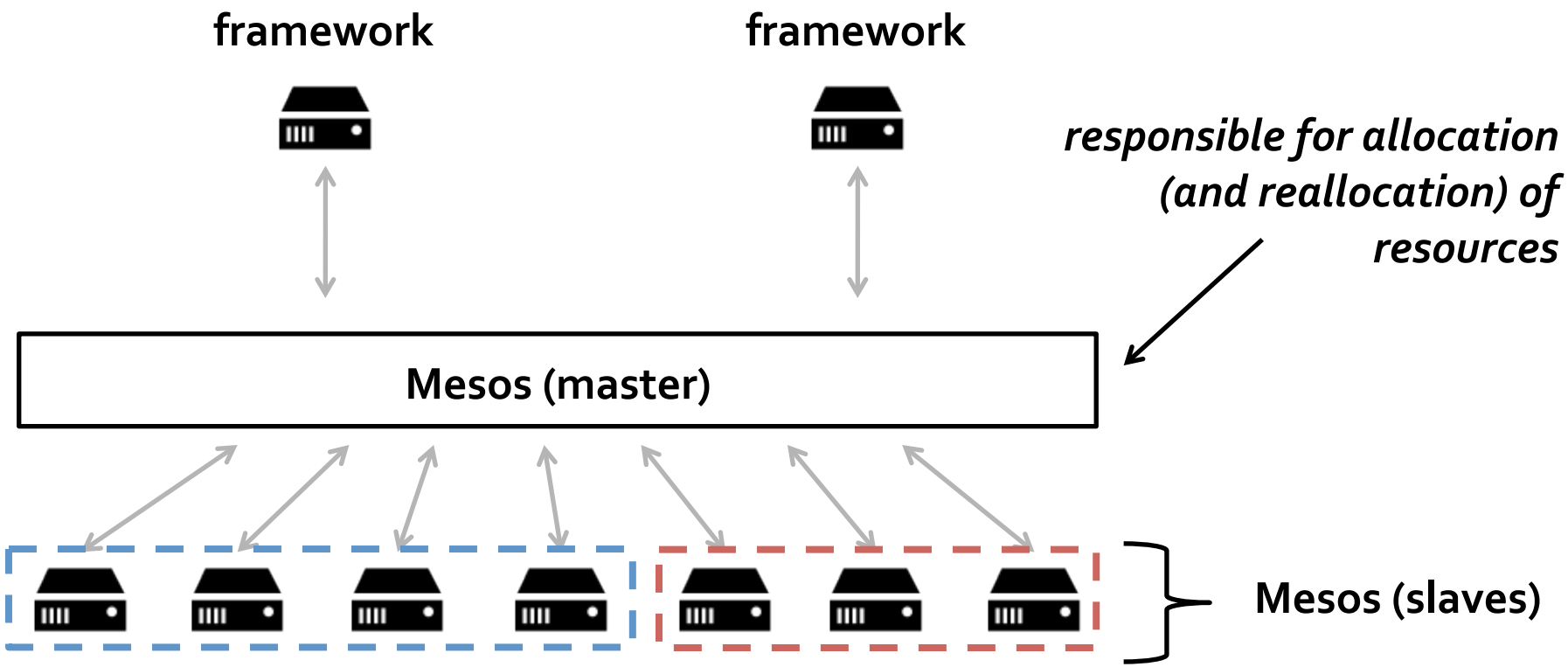
Mesos: level of abstraction



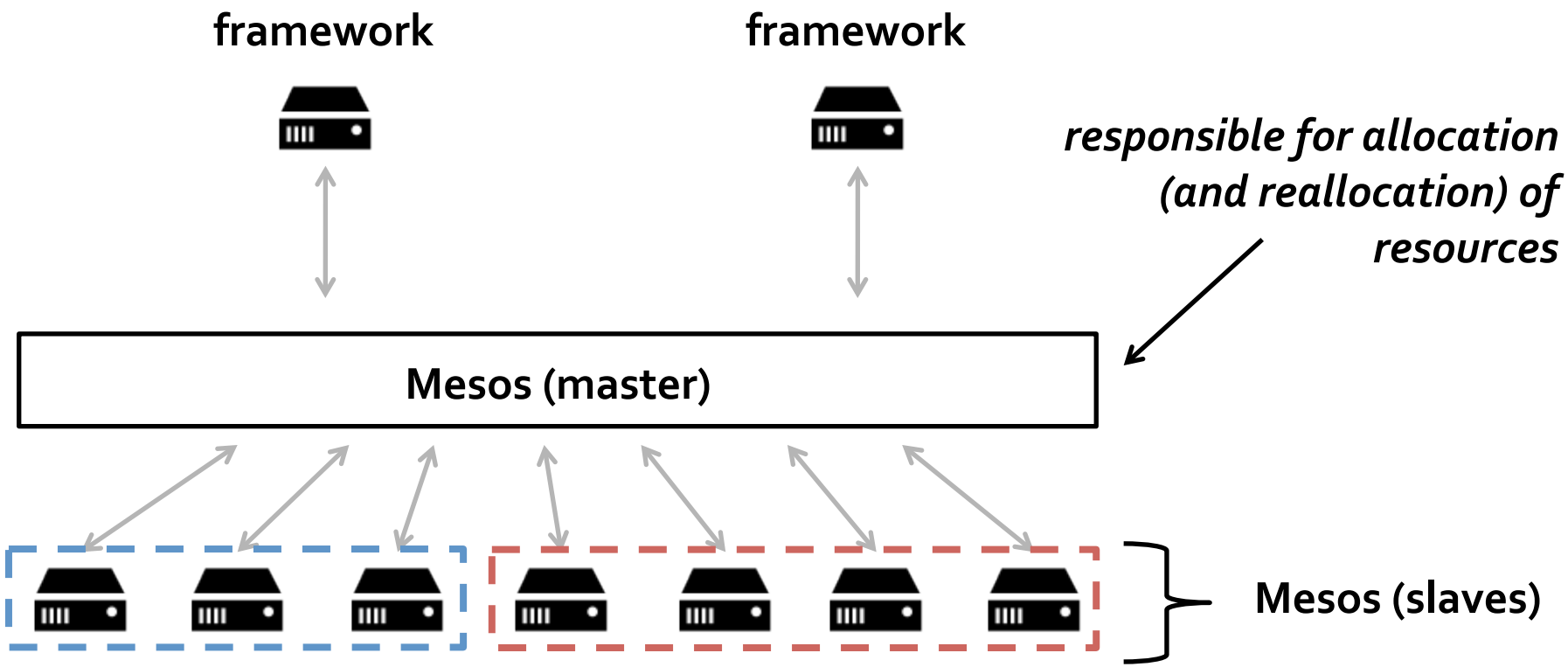
Mesos: level of abstraction



Mesos: level of abstraction



Mesos: level of abstraction



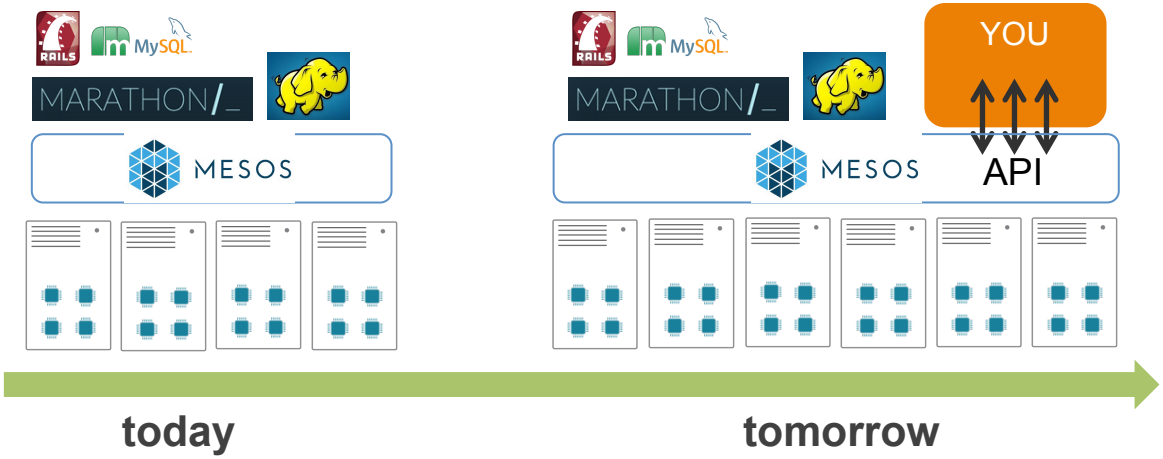
Mesos: level of abstraction

+ 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*

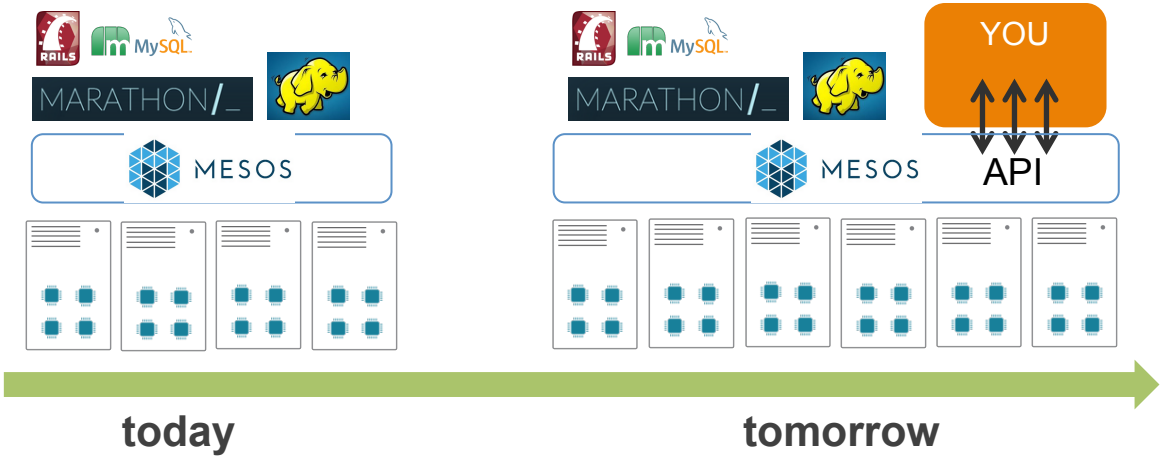
Mesos: datacenter kernel



provides common functionality every new distributed system *re-implements*:

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

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):

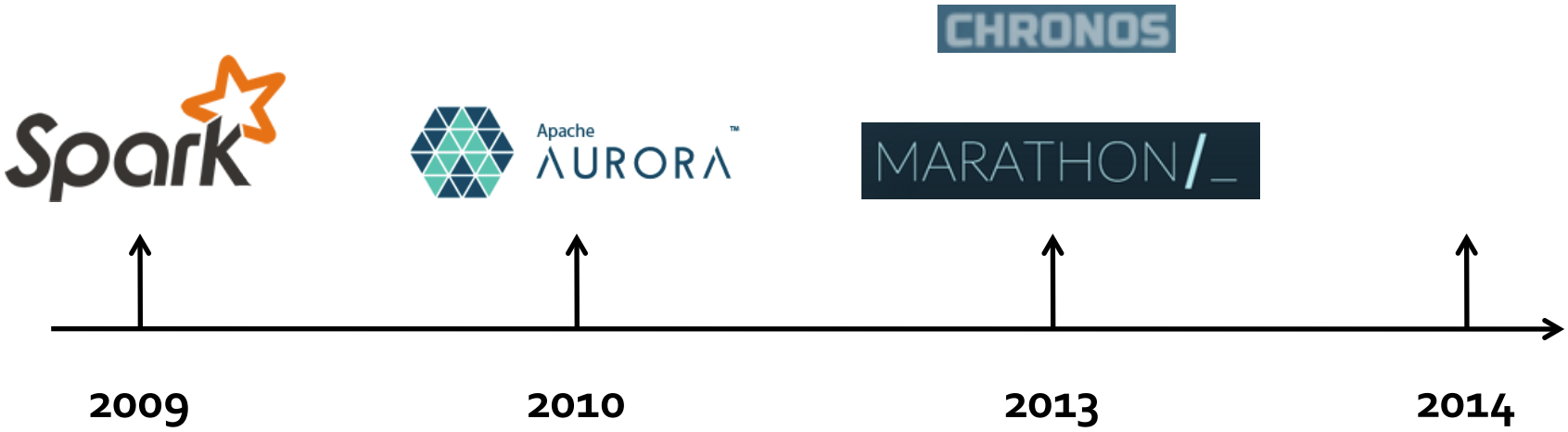
1) a virtual processor has been allocated or deallocated from you
2) this activation (i.e., context) has blocked or unblocked

Mesos

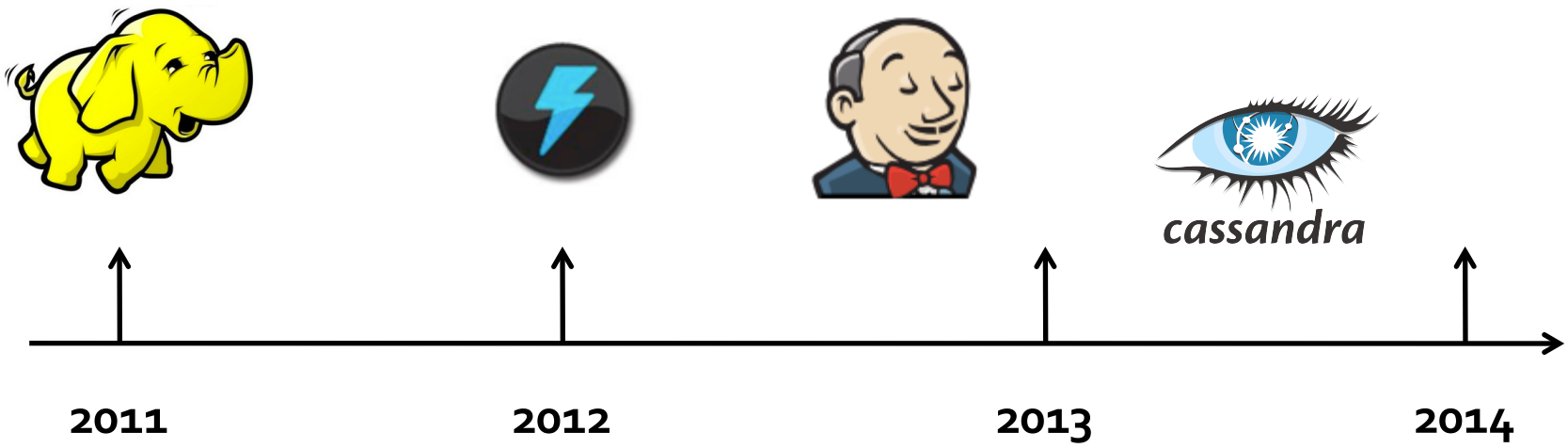
1st-level scheduler *communicates* with framework-level scheduler via "upcalls" (i.e., events):

1) resources have been allocated or deallocated from you
2) this task has completed, been preempted, etc
3) …

built on Mesos:



ported to Mesos:



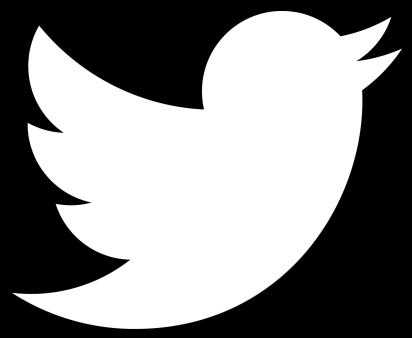
agenda

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



NETFLIX

ebay

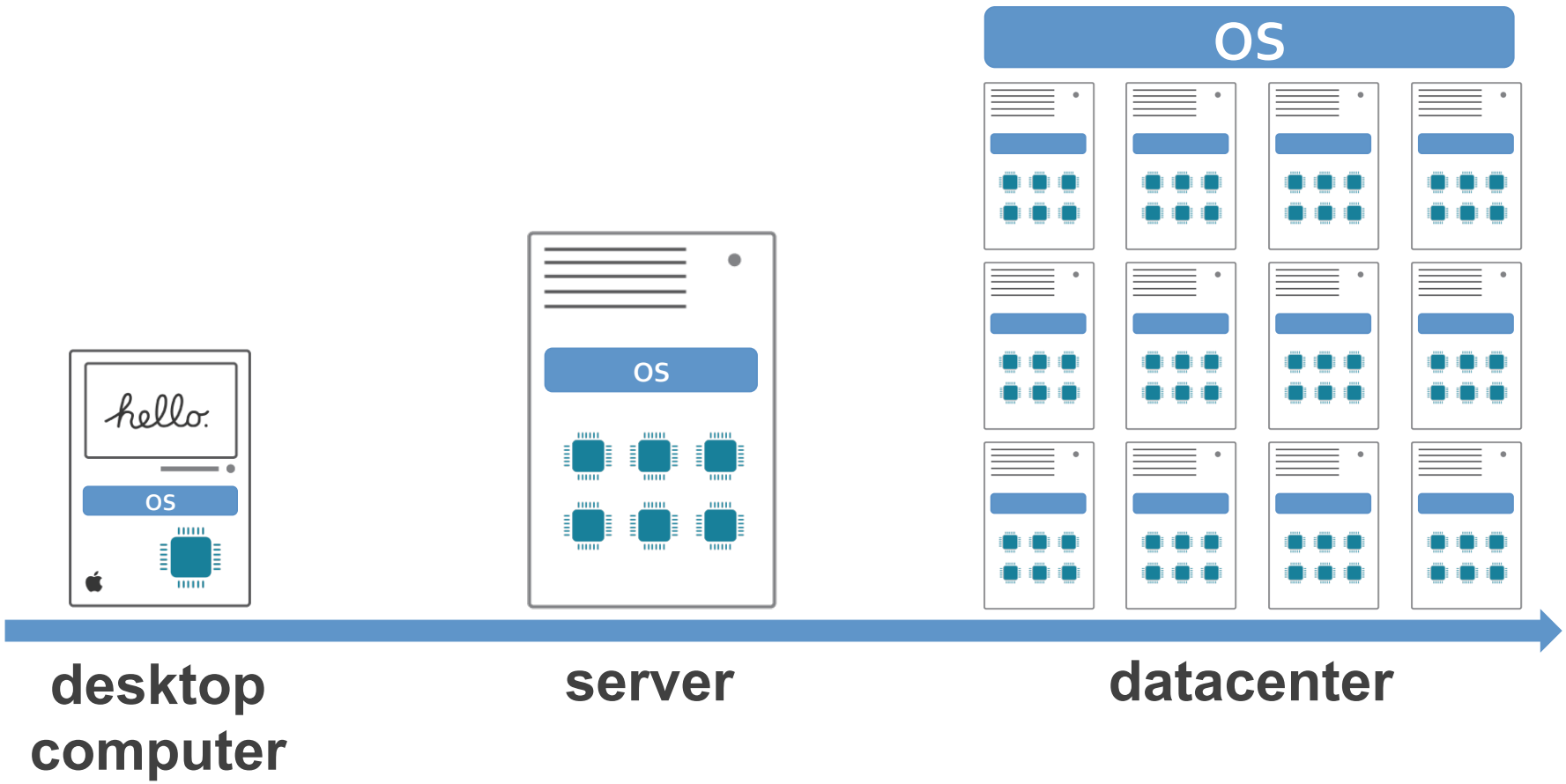
GROUPON

PayPal™

yelp

OpenTable™

the datacenter computer needs an operating system



build a distributed system in one organization and easily deploy it in another!

MESOSPHERE

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!

<u>Composing parallel software efficiently with Lithe</u>

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

<u>Mesos: a platform for fine-grained resource sharing in the data center</u>

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