Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

Benjamin Hindman, Andy Konwinski, Matei Zaharia,
Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica
University of California, Berkeley

Thursday 30th September, 2010, 12:57

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To support the sophisticated schedulers of today's frameworks, Mesos introduces a distributed two-level scheduling mechanism called resource offers. Mesos decides *how many* resources to offer each framework, while frameworks decide *which* resources to accept and which computations to run on them. Our results show that Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks, can scale to 50,000 (emulated) nodes, and is resilient to failures.

1 Introduction

Clusters of commodity servers have become a major computing platform, powering both large Internet services and a growing number of data-intensive scientific applications. Driven by these applications, researchers and practitioners have been developing a diverse array of cluster computing frameworks to simplify programming the cluster. Prominent examples include MapReduce [23], Dryad [30], MapReduce Online [22] (which supports streaming jobs), Pregel [34] (a specialized framework for graph computations), and others [33, 18, 28].

It seems clear that new cluster computing frameworks[1] will continue to emerge, and that no framework will be optimal for all applications. Therefore, organizations will want to run *multiple frameworks in the same cluster*, picking the best one for each application. Sharing a cluster between frameworks improves utilization and allows applications to share access to large datasets that may be too costly to replicate.

[1] By framework we mean a software system that manages and executes one or more jobs on a cluster.

The solutions of choice to share a cluster today are either to statically partition the cluster and run one framework per partition, or allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-grained resource sharing model, where nodes are subdivided into "slots" and jobs are composed of short *tasks* that are matched to slots [31, 44]. The short duration of tasks and the ability to run multiple tasks per node allow jobs to achieve high data locality, as each job will quickly get a chance to run on nodes storing its input data. Short tasks also allow frameworks to achieve high utilization, as jobs can rapidly scale when new nodes become available. Unfortunately, because these frameworks are developed independently, there is no way to perform fine-grained sharing *across* frameworks, making it difficult to share clusters and data efficiently between them.

In this paper, we propose Mesos, a thin resource sharing layer that enables fine-grained sharing across diverse cluster computing frameworks, by giving frameworks a common interface for accessing cluster resources.

The main design question that Mesos must address is how to match resources with tasks. This is challenging for several reasons. First, a solution will need to support a wide array of both current and future frameworks, each of which will have different scheduling needs based on its programming model, communication pattern, task dependencies, and data placement. Second, the solution must be highly scalable, as modern clusters contain tens of thousands of nodes and have hundreds of jobs with millions of tasks active at a time. Third, the scheduling system must be fault-tolerant and highly available, as all the applications in the cluster depend on it.

One approach would be for Mesos to implement a centralized scheduler that takes as input framework requirements, resource availability, and organizational policies,

and computes a global schedule for all tasks. While this approach can optimize scheduling across frameworks, it faces several challenges. The first is complexity. The scheduler would need to provide a sufficiently expressive API to capture all frameworks' requirements, and to solve an on-line optimization problem for millions of tasks. Even if such a scheduler were feasible, this complexity would have a negative impact on its scalability and resilience. Second, as new frameworks and new scheduling policies for current frameworks are constantly being developed [42, 44, 46, 32], it is not clear whether we are even at the point to have a full specification of framework requirements. Third, many existing frameworks implement their own sophisticated scheduling [31, 44], and moving this functionality to a global scheduler would require expensive refactoring.

Instead, Mesos takes a different approach: *delegating* control over scheduling to the frameworks. This is accomplished through a new abstraction, called a *resource offer*, which encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks. Mesos decides *how many* resources to offer each framework, based on an organizational policy such as fair sharing, while frameworks decide *which* resources to accept and which tasks to run on them. While this decentralized scheduling model may not always lead to globally optimal scheduling, we have found that it performs surprisingly well in practice, allowing frameworks to meet goals such as data locality nearly perfectly. In addition, resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

Mesos's flexible fine-grained sharing model also has other advantages. First, even organizations that only use one framework can use Mesos to run multiple instances of that framework in the same cluster, or multiple versions of the framework. Our contacts at Yahoo! and Facebook indicate that this would be a compelling way to isolate production and experimental Hadoop workloads and to roll out new versions of Hadoop [14, 12].

Second, by providing a means of sharing resources across frameworks, Mesos allows framework developers to build *specialized* frameworks targeted at particular problem domains rather than one-size-fits-all abstractions. Frameworks can therefore evolve faster and provide better support for each problem domain.

We have implemented Mesos in 10,000 lines of C++. The system scales to 50,000 (emulated) nodes and uses ZooKeeper [4] for fault tolerance. To evaluate Mesos, we have ported three cluster computing systems to run over it: Hadoop, MPI, and the Torque batch scheduler. To validate our hypothesis that specialized frameworks provide value over general ones, we have also built a new framework on top of Mesos called Spark, optimized for iterative jobs where a dataset is reused in many parallel operations, and shown that Spark can outperform Hadoop by 10x in iterative machine learning workloads.

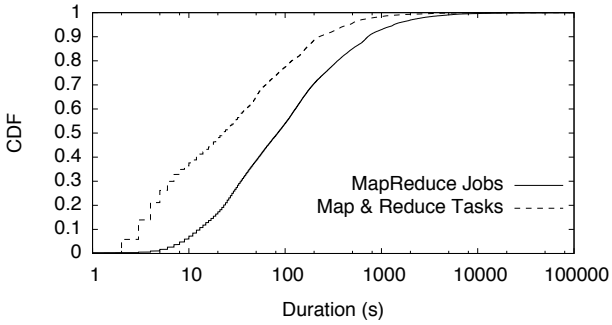


Figure 1: CDF of job and task durations in Facebook's Hadoop data warehouse (data from [44]).

This paper is organized as follows. Section 2 details the data center environment that Mesos is designed for. Section 3 presents the architecture of Mesos. Section 4 analyzes our distributed scheduling model and characterizes the environments it works well in. We present our implementation of Mesos in Section 5 and evaluate it in Section 6. Section 7 surveys related work. Finally, we conclude in Section 8.

2 Target Environment

As an example of a workload we aim to support, consider the Hadoop data warehouse at Facebook [5, 6]. Facebook loads logs from its web services into a 1200-node Hadoop cluster, where they are used for applications such as business intelligence, spam detection, and ad optimization. In addition to "production" jobs that run periodically, the cluster is used for many experimental jobs, ranging from multi-hour machine learning computations to 1-2 minute ad-hoc queries submitted interactively through an SQL interface to Hadoop called Hive [3]. Most jobs are short (the median being 84s long), and the jobs are composed of fine-grained map and reduce tasks (the median task being 23s), as shown in Figure 1.

To meet the performance requirements of these jobs, Facebook uses a fair scheduler for Hadoop that takes advantage of the fine-grained nature of the workload to make scheduling decisions at the level of map and reduce tasks and to optimize data locality [44]. Unfortunately, this means that the cluster can only run Hadoop jobs. If a user wishes to write a new ad targeting algorithm in MPI instead of MapReduce, perhaps because MPI is more efficient for this job's communication pattern, then the user must set up a separate MPI cluster and import terabytes of data into it.[2] Mesos aims to enable fine-grained sharing between *multiple* cluster computing frameworks, while giving these frameworks enough control to achieve placement goals such as data locality.

[2]This problem is not hypothetical; our contacts at Yahoo! and Facebook report that users want to run MPI and MapReduce Online [13, 12].

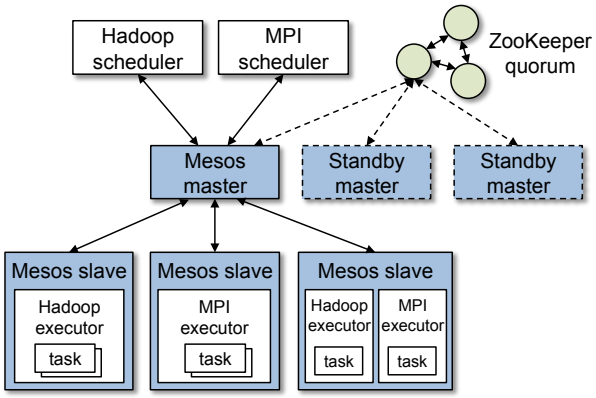


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

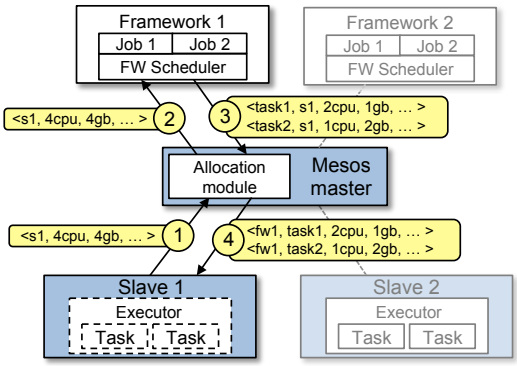


Figure 3: Resource offer example.

3 Architecture

We begin our description of Mesos by discussing our design philosophy. We then describe the components of Mesos, our resource allocation mechanisms, and how Mesos achieves isolation, scalability, and fault tolerance.

3.1 Design Philosophy

Mesos aims to provide a scalable and resilient core for enabling various frameworks to efficiently share clusters. Because cluster frameworks are both highly diverse and rapidly evolving, our overriding design philosophy has been to *define a minimal interface that enables efficient resource sharing across frameworks, and otherwise push control of task scheduling and execution to the frameworks*. Pushing control to the frameworks has two benefits. First, it allows frameworks to implement diverse approaches to various problems in the cluster (e.g., achieving data locality, dealing with faults), and to evolve these solutions independently. Second, it keeps Mesos simple and minimizes the rate of change required of the system, which makes it easier to keep Mesos scalable and robust.

Although Mesos provides a low-level interface, we expect higher-level libraries implementing common functionality (such as fault tolerance) to be built on top of it. These libraries would be analogous to library OSes in the exokernel [25]. Putting this functionality in libraries rather than in Mesos allows Mesos to remain small and flexible, and lets the libraries evolve independently.

3.2 Overview

Figure 2 shows the main components of Mesos. Mesos consists of a *master* process that manages *slave* daemons running on each cluster node, and *frameworks* that run *tasks* on these slaves.

The master implements fine-grained sharing across frameworks using *resource offers*. Each resource offer contains a list of free resources on multiple slaves. The master decides *how many* resources to offer to each framework according to a given organizational policy, such as fair sharing, or strict priority. To support a diverse set of policies, the master employs a modular architecture that makes it easy to add new allocation modules via a pluggin mechanism. To make the master fault-tolerant we use ZooKeeper [4] to implement the failover mechanism (see Section 3.6).

A framework running on top of Mesos consists of two components: a *scheduler* that registers with the master to be offered resources, and an *executor* process that is launched on slave nodes to run the framework's tasks. While the master determines how many resources are offered to each framework, the frameworks' schedulers select *which* of the offered resources to use. When a frameworks accepts offered resources, it passes to Mesos a description of the tasks it wants to run on them. In turn, Mesos launches the tasks on the corresponding slaves.

Figure 3 shows an example of how a framework gets scheduled to run a task. In step (1), slave 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources. In step (2) the master sends a resource offer describing what is available on slave 1 to framework 1. In step (3), the framework's scheduler replies to the master with information about two tasks to run on the slave, using $\langle 2 \text{ CPUs}, 1 \text{ GB RAM} \rangle$ for the first task, and $\langle 1 \text{ CPUs}, 2 \text{ GB RAM} \rangle$ for the second task. Finally, in step (4), the master sends the tasks to the slave, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2. In addition, this resource offer process repeats when tasks finish and new resources become free.

While the thin interface provided by Mesos allows it to scale and allows the frameworks to evolve independently, one question remains: how can the constraints of a framework be satisfied without Mesos knowing about these constraints? In particular, how can a

framework achieve data locality without Mesos knowing which nodes store the data required by the framework? Mesos answers these questions by simply giving frameworks the ability to *reject* offers. A framework will reject the offers that do not satisfy its constraints and accept the ones that do. In particular, we have found that a simple policy called delay scheduling [44], in which frameworks wait for a limited time to acquire nodes storing the input data, yields nearly optimal data locality. We report these results in Section 6.3.

In the remainder of this section, we describe how Mesos performs two key functionalities: resource offer allocation (performed by allocation modules in the master), and resource isolation (performed by slaves). We then describe the main elements that make the resource offer mechanism robust and efficient.

3.3 Resource Allocation

Mesos delegates allocation decisions to a pluggable *allocation module*, so that organizations can tailor allocation to their needs. In normal operation, Mesos takes advantage of the fact that most tasks are short, and only reallocate resources when tasks finish. This usually happens frequently enough so that new frameworks acquire their share quickly. For example, if a framework's share is 10% of the cluster, it needs to wait on average 10% of the mean task length to receive its share. If resources are not freed quickly enough, the allocation module also has the ability to *revoke* (kill) tasks.

So far, we have implemented two simple allocation modules: one that performs fair sharing [27] and one that implements strict priorities. Similar policies are used by current schedulers for Hadoop and Dryad [31, 44].

3.3.1 Revocation

As described earlier, in an environment with fine-grained tasks, Mesos can reallocate resources quickly by simply waiting for tasks to finish. However, if a cluster becomes filled by long tasks, e.g., due to a buggy job or a greedy framework, Mesos can also revoke (kill) tasks. Before killing a task, Mesos gives its framework a *grace period* to clean it up. Mesos asks the respective executor to kill the task, but kills the entire executor and all its tasks if it does not respond to the request. We leave it up to the allocation module to implement the *policy* for revoking tasks, but describe two related mechanisms here.

First, while killing a task has a low impact on many frameworks (e.g., MapReduce or stateless web servers), it is harmful for frameworks with interdependent tasks (e.g., MPI). We allow these frameworks to avoid being killed by letting allocation modules expose a *guaranteed allocation* to each framework – a quantity of resources that the framework may hold without losing tasks. Frameworks read their guaranteed allocations through an API call. Allocation modules are responsible for ensuring that the guaranteed allocations they provide can all be met concurrently. For now, we have kept the semantics of guaranteed allocations simple: if a framework is below its guaranteed allocation, none of its tasks should be killed, and if it is above, any of its tasks may be killed. However, if this model is found to be too simple, it is also possible to let frameworks specify priorities for their tasks, so that the allocation module can try to kill only low-priority tasks.

Second, to decide when to trigger revocation, allocation modules must know which frameworks would use more resources if they were offered them. Frameworks indicate their interest in offers through an API call.

3.4 Isolation

Mesos provides performance isolation between framework executors running on the same slave by leveraging existing OS isolation mechanisms. Since these mechanisms are platform-dependent, we support multiple isolation mechanisms through pluggable *isolation modules*.

Our current implementation isolates resources using operating system container technologies, specifically Linux containers [10] and Solaris projects [17]. These technologies can limit the CPU, memory, network bandwidth, and (in new Linux kernels) I/O usage of a process tree. They also support dynamic reconfiguration of a container's resource limits, which is necessary for Mesos to be able to add and remove resources from an executor as it starts and finishes tasks. In the future, we also plan to investigate using virtual machines for isolation.

We note that node isolation technologies are not perfect, but that using containers is already an advantage over the state of the art in frameworks such as Hadoop, where tasks from different jobs on the same machine simply run in separate processes.

3.5 Making Resource Offers Scalable and Robust

Because task scheduling in Mesos is a distributed process in which the master and framework schedulers communicate, it needs to be efficient and robust to failures. Mesos includes three mechanisms to help with this goal.

First, because some frameworks will *always* reject certain resources, Mesos lets them short-circuit the rejection process and avoid communication by providing *filters* to the master. We support two types of filters: "only offer nodes from list L" and "only offer nodes with at least R resources free". A resource that fails a filter is treated exactly like a rejected resource. By default, any resources rejected during an offer have a temporary 5 second filter placed on them, to minimize the programming burden on developers who do not wish to manually set filters.

Second, because a framework may take time to respond to an offer, Mesos counts resources offered to a

framework towards its share of the cluster for the purpose of allocation. This is a strong incentive for frameworks to respond to offers quickly and to filter out resources that they cannot use, so that they can get offers for more suitable resources faster.

Third, if a framework has not responded to an offer for a sufficiently long time, Mesos *rescinds* the offer and re-offers the resources to other frameworks.

We also note that even without the use of filters, Mesos can make tens of thousands of resource offers per second, because the scheduling algorithm it must perform (fair sharing) is highly efficient.

3.6 Fault Tolerance

Since all the frameworks depends on the master, it is critical to make the master fault-tolerant. To achieve this we use two techniques. First, we have designed the master to be *soft state*, *i.e.,* the master can reconstruct completely its internal state from the periodic messages it gets from the slaves, and from the framework schedulers. Second, we have implemented a hot-standby design, where the master is shadowed by several backups that are ready to take over when the master fails. Upon master failure, we use ZooKeeper [4] to select a new master from the existing backups, and direct all slaves and framework schedulers to this master. Subsequently, the new master will reconstruct the internal state from the messages it receives from slaves and framework schedulers.

Aside from handling master failures, Mesos reports task, slave and executor failures to frameworks' schedulers. Frameworks can then react to failures using the policies of their choice.

Finally, to deal with scheduler failures, Mesos allows a framework to register multiple schedulers such that when one fails, another one is notified by the Mesos master to take over. Frameworks must use their own mechanisms to share state between their schedulers.

4 Mesos Behavior

In this section, we study Mesos's behavior for different workloads. Our main goal here is not to develop a detailed and exact model of the system, but to provide a coarse understanding of its behavior.

In short, we find that Mesos performs very well when frameworks can scale up and down elastically, tasks durations are homogeneous, and frameworks prefer all nodes equally (Section 4.2). When different frameworks prefer different nodes, we show that Mesos can emulate a centralized scheduler that uses fair sharing across frameworks (Section 4.2.1). In addition, we show that Mesos can handle heterogeneous task durations without impacting the performance of frameworks with short tasks (§4.3). We also discuss how frameworks are incentivized to improve their performance under Mesos, and argue that these incentives also improve overall cluster utilization (§4.4). We conclude this section with some limitations of Mesos's distributed scheduling model (§4.5).

4.1 Definitions, Metrics and Assumptions

In our discussion, we consider three metrics:

- *Framework ramp-up time:* time it takes a new framework to achieve its allocation (*e.g.,* fair share);
- *Job completion time:* time it takes a job to complete, assuming one job per framework;
- *System utilization:* total cluster utilization.

We characterize workloads along four attributes:

- *Scale up:* Frameworks can elastically increase their allocation to take advantage of free resources.
- *Scale down:* Frameworks can relinquish resources without significantly impacting their performance.
- *Minimum allocation:* Frameworks require a certain minimum number of slots before they can start using their slots.
- *Task distribution:* The distribution of the task durations. We consider both homogeneous and heterogeneous distributions.

We differentiate between two types of resources: *mandatory* and *preferred*. A resource is *mandatory* if a framework must acquire it in order to run. For example, a graphical processor unit (GPU) is mandatory if a framework cannot run without access to GPU. In contrast, a resource is *preferred* if a framework performs "better" using it, but can also run using another equivalent resource. For example, a framework may prefer using a node that locally stores its data, but it can remotely access the data from other nodes if it must.

We assume the amount of *mandatory* resources requested by a framework never exceeds its guaranteed share. This ensures that frameworks will not deadlock waiting for the mandatory resources to become available. For simplicity, we also assume that all tasks run on identical slices of machines, called *slots*, and that each framework runs a single job.

We consider two types of frameworks: *elastic* and *rigid*. An elastic framework (*e.g.,* Hadoop, Dryad) can scale its resources up and down, *i.e.,* it can start using slots as soon as it acquires them and release slots as soon its task finish. In contrast, a rigid framework (*e.g.,* MPI) can start running its jobs only after it has allocated all its slots, *i.e.,* the minimum allocation of a rigid framework is equal to its full allocation.

4.2 Homogeneous Tasks

We consider a cluster with n slots and a framework, f, that is entitled to k slots. For the purpose of this analysis, we consider two distributions of the task durations:

	Elastic Framework		Rigid Framework	
	Constant dist.	Exponential dist.	Constant dist.	Exponential dist.
Ramp-up time	T_s	$T_s \ln k$	T_s	$T_s \ln k$
Completion time	$(1/2+\beta)T_s$	$(1+\beta)T_s$	$(1+\beta)T_s$	$(\ln k+\beta)T_s$
Utilization	1	1	$\beta/(1/2+\beta)$	$\beta/(\ln k-1+\beta)$

Table 1: Ramp-up time, job completion time and utilization for both elastic and rigid frameworks, and for both constant and exponential task duration distributions. The framework starts with no slots. k is the number of slots the framework is entitled under the scheduling policy, and βT_s represents the time it takes a job to complete assuming the framework gets all k slots at once.

constant and exponential. The mean task duration is T_s, and assume that framework f runs a job which requires $\beta k T_s$ computation time. Thus, assuming the framework has k slots, it takes the job βT_s time to finish.

Table 1 summarizes the job completion times and the utilization for the two types of frameworks and for the two types of task length distributions. As expected, elastic frameworks with constant task durations perform the best, while rigid frameworks with exponential task duration perform the worst. Due to lack of space we present here only the results and include derivations in [29].

Framework ramp-up time If task durations are constant, it will take framework f at most T_s time to acquire k slots. This is simply because during a T_s interval, every slot will become available, which will enable Mesos to offer the framework all its k preferred slots. If the duration distribution is exponential, the expected ramp-up time can be as high as $T_s \ln k$ [29].

Job completion time The expected completion time[3] of an elastic job is at most $(1+\beta)T_s$, which is within T_s (*i.e.,* the mean duration of a task) of the completion time of the job when it gets all its slots instantaneously. Rigid jobs achieve similar completion times for constant job durations, but exhibit much higher completion times for exponential job durations, *i.e.,* $(\ln k+\beta)T_s$. This is simply because it takes a framework $T_s \ln k$ time on average to acquire all its slots and launch the job.

System utilization Elastic jobs fully utilize their allocated slots, since they can use every slot as soon as they get it. As a result, assuming infinite demand, a system running elastic jobs is fully utilized. Rigid frameworks provide slightly worse utilizations, as their jobs cannot start before they get their full allocations, and thus they waste the slots acquired early.

4.2.1 Placement Preferences

So far, we have assumed that frameworks have no slot preferences. In practice, different frameworks prefer different nodes and their preferences may change over time. In this section, we consider the case where frameworks have different preferred slots.

The natural question is how well Mesos will work in this case when compared to a centralized scheduler that has full information about framework preferences. We consider two cases: (a) there exists a system configuration in which each framework gets all its preferred slots and achieves its full allocation, and (b) there is no such configuration, *i.e.,* the demand for preferred slots exceeds the supply.

In the first case, it is easy to see that, irrespective of the initial configuration, the system will converge to the state where each framework allocates its preferred slots after at most one T_s interval. This is simple because during a T_s interval all slots become available, and as a result each framework will be offered its preferred slots.

In the second case, there is no configuration in which all frameworks can satisfy their preferences. The key question in this case is how should one allocate the preferred slots across the frameworks demanding them. In particular, assume there are x slots preferred by m frameworks, where framework i requests r_i such slots, and $\sum_{i=1}^{m} r_i > x$. While many allocation policies are possible, here we consider the weighted fair allocation policy where the weight associated with a framework is its intended allocation, s_i. In other words, assuming that each framework has enough demand, framework i will get $x \times s_i/(\sum_{i=1}^{m} s_i)$.

The challenge with Mesos is that the scheduler does not know the preferences of each framework. Fortunately, it turns out that there is an easy way to achieve the fair allocation of the preferred slots described above: simply offer slots to frameworks proportionally to their intended allocations. In particular, when a slot becomes available, Mesos can offer that slot to framework i with probability $s_i/(\sum_{i=1}^{n} s_i)$, where n is the total number of frameworks in the system. Note that this scheme is similar to lottery scheduling [41]. Furthermore, note that since each framework i receives roughly s_i slots during a time interval T_s, the analysis of the ramp-up and completion times in Section 4.2 still holds.

4.3 Heterogeneous Tasks

So far we have assumed that frameworks have homogeneous task duration distributions. In this section, we discuss heterogeneous tasks, in particular, tasks that are either short and long, where the mean duration of the long tasks is significantly longer than the mean of the short tasks. Such heterogeneous workload can hurt frameworks with short jobs. Indeed, in the worst case, all

[3] When computing job completion time we assume that the last tasks of the job running on the framework's k slots finish at the *same* time.

nodes required by a short job might be filled with long tasks, so the job may need to wait a long time (relative to its running time) to acquire resources on these nodes.

The master can alleviate this by implementing an allocation policy that limits the number of slots on each node that can be used by long tasks, *e.g.,* no more than 50% of the slots on each node can run long tasks. This ensures there are enough short tasks on each node whose slots become available with high frequency, giving frameworks better opportunities to quickly acquire a slot on one of their preferred nodes. Note that the master does not need to know whether a task is short or long. By simply using different timeouts to revoke short and long slots, the master incentivizes the frameworks to run long tasks on long slots only. Otherwise, if a framework runs a long task on a short slot, its performance may suffer, as the slot will be most likely revoked before the task finishes.

4.4 Framework Incentives

Mesos implements a decentralized scheduling approach, where each framework decides which offers to accept or reject. As with any decentralized system, it is important to understand the incentives of various entities in the system. In this section, we discuss the incentives of a framework to improve the response times of its jobs.

Short tasks: A framework is incentivized to use short tasks for two reasons. First, it will be able to allocate any slots; in contrast frameworks with long tasks are restricted to a subset of slots. Second, using small tasks minimizes the wasted work if the framework loses a task, either due to revocation or simply due to failures.

No minimum allocation: The ability of a framework to use resources as soon as it allocates them–instead of waiting to reach a given minimum allocation–would allow the framework to start (and complete) its jobs earlier.

Scale down: The ability to scale down allows a framework to grab opportunistically the available resources, as it can later release them with little negative impact.

Do not accept unknown resources: Frameworks are incentivized not to accept resources that they cannot use because most allocation policies will account for all the resources that a framework owns when deciding which framework to offer resources to next.

We note that these incentives are all well aligned with our goal of improving utilization. When frameworks use short tasks, Mesos can reallocate resources quickly between them, reducing the need for wasted work due to revocation. If frameworks have no minimum allocation and can scale up and down, they will opportunistically utilize all the resources they can obtain. Finally, if frameworks do not accept resources that they do not understand, they will leave them for frameworks that do.

4.5 Limitations of Distributed Scheduling

Although we have shown that distributed scheduling works well in a range of workloads relevant to current cluster environments, like any decentralized approach, it can perform worse than a centralized scheduler. We have identified three limitations of the distributed model:

Fragmentation: When tasks have heterogeneous resource demands, a distributed collection of frameworks may not be able to optimize bin packing as well as a centralized scheduler.

There is another possible bad outcome if allocation modules reallocate resources in a naive manner: when a cluster is filled by tasks with small resource requirements, a framework f with large resource requirements may starve, because whenever a small task finishes, f cannot accept the resources freed up by it, but other frameworks can. To accommodate frameworks with large per-task resource requirements, allocation modules can support a *minimum offer size* on each slave, and abstain from offering resources on that slave until this minimum amount is free.

Note that the wasted space due to both suboptimal bin packing and fragmentation is bounded by the ratio between the largest task size and the node size. Therefore, clusters running "larger" nodes (e.g., multicore nodes) and "smaller" tasks within those nodes (e.g., having a cap on task resources) will be able to achieve high utilization even with a distributed scheduling model.

Interdependent framework constraints: It's possible to construct scenarios where, because of esoteric interdependencies between frameworks' performance, only a single global allocation of the cluster resources performs well. We argue such scenarios are rare in practice. In the model discussed in this section, where frameworks only have preferences over placement, we showed that allocations approximate those of optimal schedulers.

Framework complexity: Using resources offers may make framework scheduling more complex. We argue, however, that this difficulty is not in fact onerous. First, whether using Mesos or a centralized scheduler, frameworks need to know their preferences; in a centralized scheduler, the framework would need to express them to the scheduler, whereas in Mesos, it needs to use them to decide which offers to accept. Second, many scheduling policies for existing frameworks are online algorithms, because frameworks cannot predict task times and must be able to handle failures and stragglers [46, 44]. These policies are easily implemented over resource offers.

5 Implementation

We have implemented Mesos in about 10,000 lines of C++. The system runs on Linux, Solaris and Mac OS X.

Mesos applications can be programmed in C, C++, Java, and Python. We use SWIG [16] to generate interface bindings for the latter two languages.

To reduce the complexity of our implementation, we use a C++ library called `libprocess` [8] that provides an actor-based programming model using efficient asynchronous I/O mechanisms (`epoll`, `kqueue`, etc). We also leverage Apache ZooKeeper [4] to perform leader election, as described in Section 3.6. Finally, our current frameworks use HDFS [2] to share data.

Our implementation can use Linux containers [10] or Solaris projects [17] to isolate applications. We currently isolate CPU cores and memory.[4]

We have implemented five frameworks on top of Mesos. First, we have ported three existing cluster systems to Mesos: Hadoop [2], the Torque resource scheduler [37], and the MPICH2 implementation of MPI [21]. None of these ports required changing these frameworks' APIs, so all of them can run unmodified user programs. In addition, we built a specialized framework for iterative jobs called Spark, which we discuss in Section 5.3.

5.1 Hadoop Port

Porting Hadoop to run on Mesos required relatively few modifications, because Hadoop concepts such as map and reduce tasks correspond cleanly to Mesos abstractions. In addition, the Hadoop "master", known as the JobTracker, and Hadoop "slaves", known as TaskTrackers, naturally fit into the Mesos model as a framework scheduler and executor.

To add support for running Hadoop on Mesos, we took advantage of the fact that Hadoop already has a pluggable API for writing job schedulers. We wrote a Hadoop scheduler that connects to Mesos, launches TaskTrackers as its executors, and maps each Hadoop task to a Mesos task. When there are unlaunched tasks in Hadoop, our scheduler first launches Mesos tasks on the nodes of the cluster that it wants to use, and then sends the Hadoop tasks to them using Hadoop's existing task launching interface. Finally, our executor notifies Mesos when tasks finish by listening for task finish events using an interface provided by the TaskTracker.

We use delay scheduling [44] to achieve data locality by waiting for slots on the nodes that contain task input data. In addition, our approach allowed us to reuse Hadoop's existing algorithms for re-scheduling of failed tasks and speculative execution (straggler mitigation).

We also needed to change how map output data is served to reduce tasks. Hadoop normally writes map output files to the local filesystem, then serves these to reduce tasks using an HTTP server included in the TaskTracker. However, the TaskTracker within Mesos runs as an executor, which may be terminated if it is not running tasks, which would make map output files unavailable to reduce tasks. We solved this problem by providing a shared file server on each node in the cluster to serve local files. Such a service is useful beyond Hadoop, to other frameworks that write data locally on each node.

In total, our Hadoop port is 1500 lines of code.

5.2 Torque and MPI Ports

We have ported the Torque cluster resource manager to run as a framework on Mesos. The framework consists of a Mesos *framework scheduler* and *framework executor*, written in 360 lines of Python code, that launch and manage different components of Torque. In addition, we modified 3 lines of Torque source code to allow it to elastically scale up and down on Mesos depending on the jobs in its queue.

After registering with the Mesos master, the framework scheduler configures and launches a Torque server and then periodically monitors the server's job queue. While the queue is empty, the framework scheduler releases all tasks (down to an optional minimum, which we set to 0) and refuses all resource offers it receives from Mesos. Once a job gets added to Torque's queue (using the standard `qsub` command), the framework scheduler begins accepting new resource offers. As long as there are jobs in Torque's queue, the framework scheduler accepts offers as necessary to satisfy the constraints of as many jobs in the queue as possible. On each node where offers are accepted Mesos launches the framework executor, which in turn starts a Torque backend daemon and registers it with the Torque server. When enough Torque backend daemons have registered, the torque server will launch the next job in its queue.

Because jobs that run on Torque (e.g. MPI) may not be fault tolerant, Torque avoids having its tasks revoked by not accepting resources beyond its guaranteed allocation.

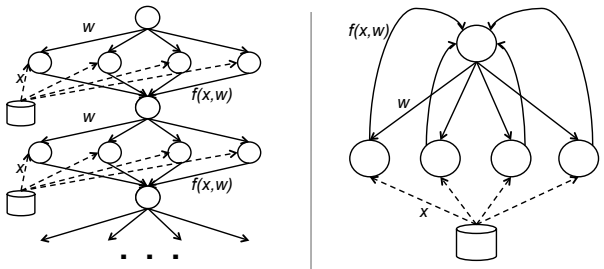
In addition to the Torque framework, we also created a Mesos MPI "wrapper" framework, written in 200 lines of Python code, for running MPI jobs directly on Mesos.

5.3 Spark Framework

To show the value of simple but specialized frameworks, we built Spark, a new framework for *iterative jobs* that was motivated from discussions with machine learning researchers at our institution.

One iterative algorithm used frequently in machine learning is logistic regression [11]. An implementation of logistic regression in Hadoop must run each iteration as a separate MapReduce job, because each iteration depends on values computed in the previous round. In this case, every iteration must re-read the input file from disk into memory. In Dryad, the whole job can be expressed as a data flow DAG as shown in Figure 4a, but the data

[4] Support for network and IO isolation was recently added to the Linux kernel [9] and we plan to add support for these resources too.



a) Dryad b) Spark

Figure 4: Data flow of a logistic regression job in Dryad vs. Spark. Solid lines show data flow within the framework. Dashed lines show reads from a distributed file system. Spark reuses processes across iterations, only loading data once.

must still must be reloaded from disk into memory at each iteration. Reusing the data in memory between iterations in Dryad would require *cyclic* data flow.

Spark's execution is shown in Figure 4b. Spark uses the long-lived nature of Mesos executors to cache a slice of the data set in memory at each executor, and then run multiple iterations on this cached data. This caching is achieved in a fault-tolerant manner: if a node is lost, Spark remembers how to recompute its slice of the data.

Spark leverages Scala [**?**] to provide a language-integrated syntax similar to DryadLINQ [43]: users invoke parallel operations by applying a function on a special "distributed dataset" object, and the body of the function is captured as a closure to run as a set of tasks in Mesos. Spark then schedules these tasks to run on executors that already have the appropriate data cached, using delay scheduling. By building on-top-of Mesos, Spark's implementation only required 1300 lines of code.

Due to lack of space, we have limited our discussion of Spark in this paper and refer the reader to [45] for details.

6 Evaluation

We evaluated Mesos through a series of experiments on the Amazon Elastic Compute Cloud (EC2). We begin with a macrobenchmark that evaluates how the system shares resources between four workloads, and go on to present a series of smaller experiments designed to evaluate overhead, decentralized scheduling, our specialized framework (Spark), scalability, and failure recovery.

6.1 Macrobenchmark

To evaluate the primary goal of Mesos, which is enabling diverse frameworks to efficiently share a cluster, we ran a macrobenchmark consisting of a mix of four workloads:

- A Hadoop instance running a mix of small and large jobs based on the workload at Facebook.
- A Hadoop instance running a set of large batch jobs.
- Spark running a series of machine learning jobs.
- Torque running a series of MPI jobs.

Bin	Job Type	Map Tasks	Reduce Tasks	# Jobs Run
1	selection	1	NA	38
2	text search	2	NA	18
3	aggregation	10	2	14
4	selection	50	NA	12
5	aggregation	100	10	6
6	selection	200	NA	6
7	text search	400	NA	4
8	join	400	30	2

Table 2: Job types for each bin in our Facebook Hadoop mix.

We compared a scenario where the workloads ran as four frameworks on a 96-node Mesos cluster using fair sharing to a scenario where they were each given a static partition of the cluster (24 nodes), and measured job response times and resource utilization in both cases. We used EC2 nodes with 4 CPU cores and 15 GB of RAM.

We begin by describing the four workloads in more detail, and then present our results.

6.1.1 Macrobenchmark Workloads

Facebook Hadoop Mix Our Hadoop job mix was based on the distribution of job sizes and inter-arrival times at Facebook, reported in [44]. The workload consists of 100 jobs submitted at fixed times over a 25-minute period, with a mean inter-arrival time of 14s. Most of the jobs are small (1-12 tasks), but there are also large jobs of up to 400 tasks.[5] The jobs themselves were from the Hive benchmark [7], which contains four types of queries: text search, a simple selection, an aggregation, and a join that gets translated into multiple MapReduce steps. We grouped the jobs into eight bins of job type and size (listed in Table 2) so that we could compare performance in each bin. We also set the framework scheduler to perform fair sharing between its jobs, as this policy is used at Facebook.

Large Hadoop Mix To emulate batch workloads that need to run continuously, such as web crawling, we had a second instance of Hadoop run a series of IO-intensive 2400-task text search jobs. A script launched ten of these jobs, submitting each one after the previous one finished.

Spark We ran five instances of an iterative machine learning job on Spark. These were launched by a script that waited 2 minutes after each job finished to submit the next. The job we used was alternating least squares, a collaborative filtering algorithm. This job is fairly CPU-intensive but also benefits from caching its input data on each node, and needs to broadcast updated parameters to all nodes running its tasks on each iteration.

Torque / MPI Our Torque framework ran eight instances of the `tachyon` raytracing job [39] that is part of the SPEC MPI2007 benchmark. Six of the jobs ran small

[5] We scaled down the largest jobs in [44] to have the workload fit a quarter of our cluster size.

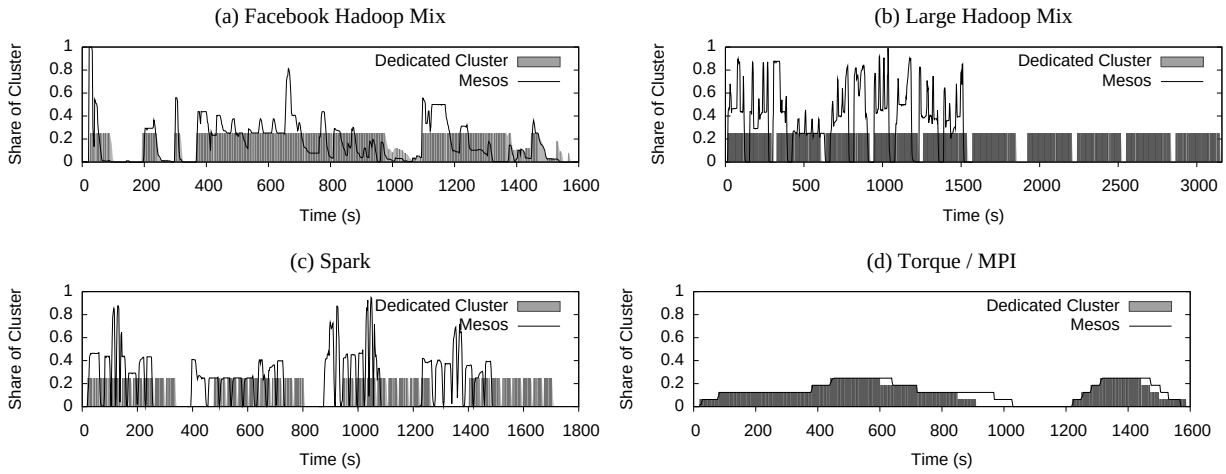


Figure 5: Comparison of cluster shares (fraction of CPUs) over time for each of the frameworks in the Mesos and static partitioning macrobenchmark scenarios. On Mesos, frameworks can scale up when their demand is high and that of other frameworks is low, and thus finish jobs faster. Note that the plots' time axes are different (*e.g.,* the large Hadoop mix takes 3200s with static partitioning).

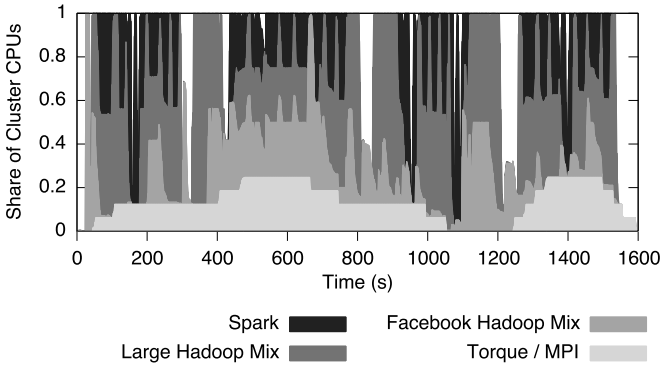


Figure 6: Framework shares on Mesos during the macrobenchmark. By pooling resources, Mesos lets each workload scale up to fill gaps in the demand of others. In addition, fine-grained sharing allows resources to be reallocated in tens of seconds.

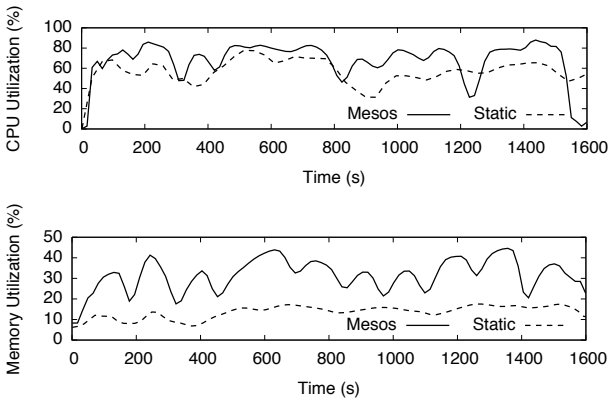


Figure 7: Average CPU and memory utilization over time across all nodes in the Mesos cluster vs. static partitioning.

problem sizes and two ran larg ones. Both types used 24 parallel tasks. We submitted these jobs at fixed times to both clusters. The `tachyon` job is CPU-intensive.

6.1.2 Macrobenchmark Results

A successful result for Mesos would show two things: that Mesos achieves higher utilization than static partitioning, and that jobs finish at least as fast in the shared cluster as they do in dedicated ones, and possibly faster due to gaps in the demand of other frameworks. Our results show both effects, as detailed below.

We show the fraction of CPU cores allocated to each framework by Mesos over time in Figure 6. We see that Mesos enables each framework to scale up during periods when other frameworks have low demands, and thus keeps cluster nodes busier. For example, at time 350, when both Spark and the Facebook Hadoop framework have no running jobs and Torque is using 1/8 of the cluster, the large-job Hadoop framework scales up to 7/8 of the cluster. In addition, we see that resources are reallocated rapidly (e.g. when a Facebook Hadoop job starts around time 360) due to the fine-grained nature of tasks. Finally, higher allocation of nodes also translates into higher CPU and memory utilization as show in Figure 7. The shared Mesos cluster has on average 10% higher CPU utilization and 18% higher memory utilization than the statically partitioned cluster.

A second question is how much better jobs perform under Mesos than on dedicated clusters. We present this data in two ways. First, Figure 5 compares the resource allocation over time of each framework in the shared and dedicated clusters. Shaded areas show the allocation in the dedicated cluster, while solid lines show the share on Mesos. We see that the fine-grained frameworks (Hadoop and Spark) take advantage of Mesos to scale up beyond 1/4 of the cluster when global demand allows this, and consequently finish bursts of submitted jobs faster in Mesos. At the same time, Torque

Framework	Sum of Exec Times on Dedicated Cluster (s)	Sum of Exec Times on Mesos (s)	Speedup
Facebook Hadoop Mix	7235	6319	**1.14**
Large Hadoop Mix	3143	1494	**2.10**
Spark	1684	1338	**1.26**
Torque / MPI	3210	3352	**0.96**

Table 3: Aggregate performance of each framework in the macrobenchmark (sum of running times of all the jobs in the framework). The speedup column shows the relative gain on Mesos.

Framework	Job Type	Time on Dedicated Cluster (s)	Avg. Speedup on Mesos
Facebook Hadoop Mix	selection (1)	24	**0.84**
	text search (2)	31	**0.90**
	aggregation (3)	82	**0.94**
	selection (4)	65	**1.40**
	aggregation (5)	192	**1.26**
	selection (6)	136	**1.71**
	text search (7)	137	**2.14**
	join (8)	662	**1.35**
Large Hadoop Mix	text search	314	**2.21**
Spark	ALS	337	**1.36**
Torque / MPI	small tachyon	261	**0.91**
	large tachyon	822	**0.88**

Table 4: Performance of each job type in the macrobenchmark. Bins for the Facebook Hadoop mix are in parentheses.

achieves roughly similar allocations and job durations under Mesos (with some differences explained later).

Second, Tables 3 and 4 show a breakdown of job performance for each framework. In Table 3, we compare the aggregate performance of each framework, defined as the sum of job running times, in the static partitioning and Mesos scenarios. We see the Hadoop and Spark jobs as a whole are finishing faster on Mesos, while Torque is slightly slower. The framework that gains the most is the large-job Hadoop mix, which almost always has tasks to run and fills in the gaps in demand of the other frameworks; this framework performs 2x better on Mesos.

Table 4 breaks down the results further by job type. We observe two notable trends. First, in the Facebook Hadoop mix, the smaller jobs perform worse on Mesos. This is due to an interaction between the fair sharing performed by the Hadoop framework (among its jobs) and the fair sharing performed by Mesos (among frameworks): During periods of time when Hadoop has more than 1/4 of the cluster, if any jobs are submitted to the other frameworks, there is a delay before Hadoop gets a new resource offer (because any freed up resources go to the framework farthest below its share), so any small job submitted during this time is delayed for a long time relative to its length. In contrast, when running alone, Hadoop can assign resources to the new job as soon as any of its tasks finishes. This problem with hierarchical fair sharing is also seen in networks [38], and could be mitigated either by running the small jobs on a separate framework or changing the allocation policy in Mesos (e.g. to use lottery scheduling instead of always offering resources to the framework farthest below its share).

Lastly, Torque is the only framework that performed worse, on average, on Mesos. The large `tachyon` jobs took on average 2 minutes longer, while the small ones took 20s longer. Some of this delay is due to Torque having to wait to launch 24 tasks on Mesos before starting each job, but the average time this takes is 12s. We believe that the rest of the delay may be due to stragglers (slow nodes). In our standalone Torque run, we noticed two jobs each took about 60s longer to run than the others (Figure 5d). We discovered that both of these jobs were using a node that performed slower on single-

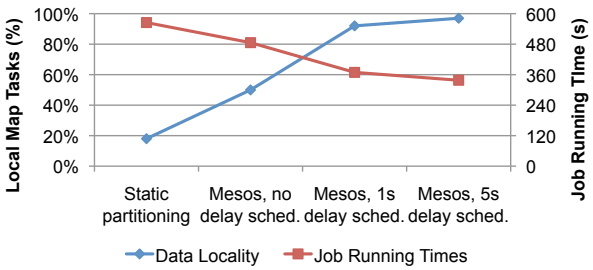


Figure 8: Data locality and average job durations for 16 Hadoop instances running on a 93-node cluster using static partitioning, Mesos, or Mesos with delay scheduling.

node benchmarks than the others (in fact, Linux reported a 40% lower bogomips value on this node). Because `tachyon` hands out equal amounts of work to each node, it runs as slowly as the slowest node. Unfortunately, when we ran the shared cluster scenario, we did not collect the data necessary to check for stragglers.

6.2 Overhead

To measure the overhead Mesos imposes when a single framework uses the cluster, we ran two benchmarks using MPI and Hadoop on an EC2 cluster with 50 nodes, each with 2 CPU cores and 6.5 GB RAM. We used the High-Performance LINPACK [19] benchmark for MPI and a WordCount job for Hadoop, and ran each job three times. The MPI job took on average 50.9s without Mesos and 51.8s with Mesos, while the Hadoop job took 160s without Mesos and 166s with Mesos. In both cases, the overhead of using Mesos was less than 4%.

6.3 Data Locality through Fine-Grained Sharing and Resource Offers

In this experiment, we evaluated how Mesos' resource offer mechanism enables frameworks to control their tasks' placement, and in particular, data locality. We ran 16 instances of Hadoop using 93 EC2 nodes, each with 4 CPU cores and 15 GB RAM. Each node ran a map-only scan job that searched a 100 GB file spread throughout the cluster on a shared HDFS file system and

outputted 1% of the records. We tested four scenarios: giving each Hadoop instance its own 5-6 node static partition of the cluster (to emulate organizations that use coarse-grained cluster sharing systems), and running all instances on Mesos using either no delay scheduling, 1s delay scheduling or 5s delay scheduling.

Figure 8 shows averaged measurements from the 16 Hadoop instances across three runs of each scenario. Using static partitioning yields very low data locality (18%) because the Hadoop instances are forced to fetch data from nodes outside their partition. In contrast, running the Hadoop instances on Mesos improves data locality, even without delay scheduling, because each Hadoop instance has tasks on more nodes of the cluster (there are 4 tasks per node), and can therefore access more blocks locally. Adding a 1-second delay brings locality above 90%, and a 5-second delay achieves 95% locality, which is competitive with running one Hadoop instance alone on the whole cluster. As expected, job performance improves with data locality: jobs run 1.7x faster in the 5s delay scenario than with static partitioning.

6.4 Spark Framework

We evaluated the benefit of running iterative jobs using the specialized Spark framework we developed on top of Mesos (Section 5.3) over the general-purpose Hadoop framework. We used a logistic regression job implemented in Hadoop by machine learning researchers in our lab, and wrote a second version of the job using Spark. We ran each version separately on 20 EC2 nodes, each with 4 CPU cores and 15 GB RAM. Each experiment used a 29 GB data file and varied the number of logistic regression iterations from 1 to 30 (see Figure 9).

With Hadoop, each iteration takes 127s on average, because it runs as a separate MapReduce job. In contrast, with Spark, the first iteration takes 174s, but subsequent iterations only take about 6 seconds, leading to a speedup of up to 10x for 30 iterations. This happens because the cost of reading the data from disk and parsing it is much higher than the cost of evaluating the gradient function computed by the job on each iteration. Hadoop incurs the read/parsing cost on each iteration, while Spark reuses cached blocks of parsed data and only incurs this cost once. The longer time for the first iteration in Spark is due to the use of slower text parsing routines.

6.5 Mesos Scalability

To evaluate Mesos' scalability, we emulated large clusters by running up to 50,000 slave daemons on 99 Amazon EC2 nodes, each with 8 CPU cores and 6 GB RAM. We used one EC2 node for the master and the rest of the nodes to run slaves. During the experiment, each of 200 frameworks running throughout the cluster continuously launches tasks, starting one task on each slave that it receives a resource offer for. Each task sleeps for a period of time based on a normal distribution with a mean of 30 seconds and standard deviation of 10s, and then ends. Each slave runs up to two tasks at a time.

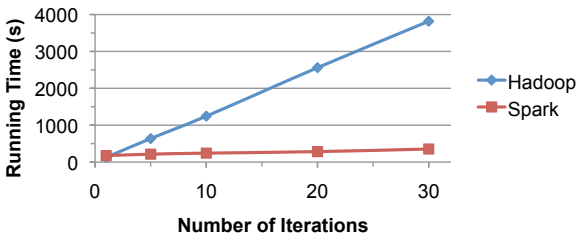


Figure 9: Hadoop and Spark logistic regression running times.

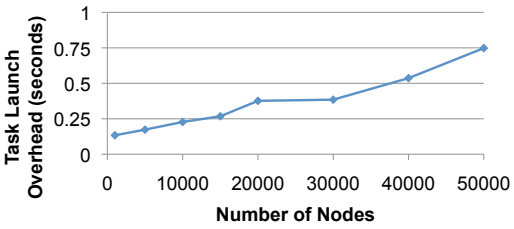


Figure 10: Mesos master's scalability versus number of slaves.

Once the cluster reached steady-state (i.e., the 200 frameworks achieve their fair shares and all resources were allocated), we launched a test framework that runs a single 10 second task and measured how long this framework took to finish. This allowed us to calculate the extra delay incurred over 10s due to having to register with the master, wait for a resource offer, accept it, wait for the master to process the response and launch the task on a slave, and wait for Mesos to report the task as finished.

We plot this extra delay in Figure 10, showing averages of 5 runs. We observe that the overhead remains small (less than one second) even at 50,000 nodes. In particular, this overhead is much smaller than the average task and job lengths in data center workloads (see Section 2). Because Mesos was also keeping the cluster fully allocated, this indicates that the master kept up with the load placed on it. Unfortunately, the EC2 virtualized environment limited scalability beyond 50,000 slaves, because at 50,000 slaves the master was processing 100,000 packets per second (in+out), which has been shown to be the current achievable limit on EC2 [15].

6.6 Failure Recovery

To evaluate recovery from master failures, we conducted an experiment similar to the scalability experiment, running 200 to 4000 slave daemons on 62 EC2 nodes with 4 cores and 15 GB RAM. We ran 200 frameworks that each launched 20-second tasks, and two Mesos masters connected to a 5-node ZooKeeper quorum using a default tick interval of 2 seconds. We synchronized the two masters' clocks using NTP and measured the mean time

to recovery (MTTR) after killing the active master. The MTTR is the time for all of the slaves and frameworks to connect to the second master. In all cases, the MTTR was between 4 and 8 seconds, with 95% confidence intervals of up to 3s on either side. Due to lack of space, we refer the reader to [29] for a graph of the results.

7 Related Work

HPC and Grid schedulers. The high performance computing (HPC) community has long been managing clusters [37, 47, 26, 20]. Their target environment typically consists of specialized hardware, such as Infiniband, SANs, and parallel filesystems. Thus jobs do not need to be scheduled local to their data. Furthermore, each job is tightly coupled, often using barriers or message passing. Thus, each job is monolithic, rather than composed of smaller fine-grained tasks. Consequently, a job does not dynamically grow or shrink its resource demands across machines during its lifetime. Moreover, fault-tolerance is achieved through checkpointing, rather than recomputing fine-grained tasks. For these reasons, HPC schedulers use centralized scheduling, and require jobs to declare the required resources at job submission time. Jobs are then allocated course-grained allocations of the cluster. Unlike the Mesos approach, this does not allow frameworks to locally access data distributed over the cluster. Furthermore, jobs cannot grow and shrink dynamically as their allocations change. In addition to supporting fine-grained sharing, Mesos can run HPC schedulers, such as Torque, as frameworks, which can then schedule HPC workloads appropriately.

Grid computing has mostly focused on the problem of making diverse virtual organizations share geographically distributed and separately administered resources in a secure and interoperable way. Mesos could well be used within a virtual organization, which is part of a larger grid that, for example, runs Globus Toolkit.

Public and Private Clouds. Virtual machine clouds, such as Amazon EC2 [1] and Eucalyptus [35] share common goals with Mesos, such as isolating frameworks while providing a low-level abstraction (VMs). However, they differ from Mesos in several important ways. First, their relatively course grained VM allocation model leads to less efficient resource utilization and data sharing than in Mesos. Second, these systems generally do not let applications specify placement needs beyond the size of virtual machine they require. In contrast, Mesos allows frameworks to be highly selective about which resources they acquire through resource offers.

Quincy. Quincy [31] is a fair scheduler for Dryad. It uses a centralized scheduling algorithm and provides a directed acyclic graph based programming model. In contrast, Mesos provides the low level, more flexible abstraction of resource offers and aims to support *multiple* cluster computing frameworks which in turn offer higher level programming abstractions.

Condor. Condor, a centralized cluster manager, uses the ClassAds language [36] to match node properties to job needs. Using such a resource specification language is not as flexible for frameworks as resource offers (i.e. not all framework requirements may be expressible). Also, porting existing frameworks, which have their own sophisticated schedulers, to Condor would be more difficult than porting them to Mesos, where existing schedulers fit naturally into the two level scheduling model.

8 Conclusion

We have presented Mesos, a thin management layer that allows diverse cluster computing frameworks to efficiently share resources. Mesos is built around two design elements: a fine-grained resource sharing model at the level of tasks within a job, and a decentralized scheduling mechanism called resource offers that lets applications choose which resources to use. Together, these elements let Mesos achieve high utilization, respond rapidly to workload changes, and cater to frameworks with diverse needs, while remaining simple and scalable. We have shown that existing frameworks can effectively share resources with Mesos, that new specialized frameworks, such as Spark, can provide major performance gains, and that Mesos's simple architecture allows the system to be fault tolerant and to scale to 50,000 (emulated) nodes.

We have recently open-sourced Mesos and started working with two companies (Twitter and Facebook) to test Mesos in their clusters.[6] We have also begun using Mesos to manage a 40-node cluster in our lab. Future work will report on lessons from these deployments.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Hive. http://hadoop.apache.org/hive/.

[4] Apache ZooKeeper. http://hadoop.apache.org/zookeeper/.

[5] Hadoop and hive at facebook. http://www.slideshare.net/dzhou/facebook-hadoop-data-applications.

[6] Hive – A Petabyte Scale Data Warehouse using Hadoop. http://www.facebook.com/note.php?note_id=89508453919#/note.php?note_id=89508453919.

[7] Hive performance benchmarks. http://issues.apache.org/jira/browse/HIVE-396.

[8] LibProcess Homepage. http://www.eecs.berkeley.edu/~benh/libprocess.

[9] Linux 2.6.33 release notes. http://kernelnewbies.org/Linux_2_6_33.

[10] Linux containers (LXC) overview document. http://lxc.sourceforge.net/lxc.html.

[11] Logistic Regression Wikipedia Page. http://en.wikipedia.org/wiki/Logistic_regression.

[6] Mesos is available at http://www.github.com/mesos.

[12] Personal communication with Dhruba Borthakur from the Facebook data infrastructure team.

[13] Personal communication with Khaled Elmeleegy from Yahoo! Research.

[14] Personal communication with Owen O'Malley from the Yahoo! Hadoop team.

[15] RightScale webpage. http://blog.rightscale.com/2010/04/01/benchmarking-load-balancers-in-the-cloud.

[16] Simplified wrapper interface generator. http://www.swig.org.

[17] Solaris Resource Management. http://docs.sun.com/app/docs/doc/817-1592.

[18] Twister. http://www.iterativemapreduce.org.

[19] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[20] D. Angulo, M. Bone, C. Grubbs, and G. von Laszewski. Workflow management through cobalt. In *International Workshop on Grid Computing Environments–Open Access (2006)*, Tallahassee, FL, November 2006.

[21] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Proceedings of Supercomputing '03*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[22] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. Mapreduce online. In *Proceedings of NSDI '10*. USENIX, May 2010.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[24] D. J. DeWitt, E. Paulson, E. Robinson, J. F. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. *PVLDB*, 1(1):28–41, 2008.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995.

[26] W. Gentzsch. Sun grid engine: towards creating a compute power grid. pages 35–36, 2001.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of heterogeneous resources in datacenters. Technical Report UCB/EECS-2010-55, EECS Dept, UC Berkeley, May 2010.

[28] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, , and L. Zhou. Comet: Batched stream processing in data intensive distributed computing. Technical Report MSR-TR-2009-180, Microsoft Research, December 2009.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the cluster. http://nexus.berkeley.edu/mesos_tech_report.pdf, May 2010.

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, pages 59–72, 2007.

[31] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP 2009*, November 2009.

[32] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *Proceedings of HOTOS '09*. USENIX, May 2009.

[33] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Programming bulk-incremental dataflows. Technical report, UC San Diego, http://cseweb.ucsd.edu/~dlogothetis/docs/bips_techreport.pdf, 2009.

[34] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODC '09*, pages 6–6, New York, NY, USA, 2009. ACM.

[35] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCA '08*, Chicago, Illinois, 10 2008.

[36] R. Raman, M. Solomon, M. Livny, and A. Roy. The classads language. pages 255–270, 2004.

[37] G. Staples. Torque - torque resource manager. In *SC*, page 8, 2006.

[38] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM '97*, pages 249–262, New York, NY, USA, 1997.

[39] J. Stone. Tachyon ray tracing system. http://jedi.ks.uiuc.edu/~johns/raytracer.

[40] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation - Practice and Experience*, 17(2-4):323–356, 2005.

[41] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94*, pages 1+, Berkeley, CA, USA, 1994. USENIX Association.

[42] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings SOSP '09*, pages 247–260, NY, 2009. ACM.

[43] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*, pages 1–14, 2008.

[44] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*, April 2010.

[45] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Dept, UC Berkeley, May 2010.

[46] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proc. OSDI 2008*, Dec. 2008.

[47] S. Zhou. Lsf: Load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing*, Tallahassee, FL, December 1992.