Cluster Scheduling for Data Centers

MALTE SCHWARZKOPF

EXPERT-CURATED GUIDES TO THE BEST OF CS RESEARCH

Research for Practice combines the resources of the ACM Digital Library, the largest collection of computer science research in the world, with the expertise of the ACM membership. In every RfP column an expert shares a short curated selection of papers on a concentrated, practically oriented topic.

This installment of Research for Practice features a curated selection from Malte Schwartzkopf, who takes us on a tour of distributed cluster scheduling, from research to practice, and back again. With the rise of elastic compute resources, cluster management has become an increasingly hot topic in systems R&D, and a number of competing cluster managers including Kubernetes, Mesos, and Docker are currently jockeying for the crown in this space. Interested in the foundations behind these systems, and how to achieve fast, flexible, and fair scheduling? Malte's got you covered! *—Peter Bailis*

Increasingly, many applications and websites rely on distributed back ends running in cloud data centers. In these data centers, clusters of hundreds or thousands of machines run workloads ranging from fault-tolerant, load-balanced web servers to batch data-processing pipelines and distributed storage stacks.

A *cluster manager* is special "orchestration" software that manages the machines and applications in such a data center automatically: some widely known examples are

Kubernetes, Mesos, and Docker Swarm. Why are cluster managers needed? Most obviously because managing systems at this scale is beyond the capabilities of human administrators. Just as importantly, however, automation and smart resource management save real money. This is true both at large scale—Google estimates that its cluster-management software helped avoid building several billion-dollar data centers—and at the scale of a startup's cloud deployment, where wasting hundreds of dollars a month on underutilized virtual machines may burn precious runway.

As few academic researchers have access to real, large-scale deployments, academic papers on cluster management largely focus on *scheduling* workloads efficiently, given limited resources, rather than on more operational aspects of the problem. Scheduling is an optimization problem with many possible answers whose relative goodness depends on the workload and the operator's goals. Thinking about how to solve the scheduling problem, however, has also given rise to a vigorous debate about the right architecture for *scalable* schedulers for ever-larger clusters and increasingly demanding workloads.

Let's start by looking at a paper that nicely summarizes the many facets of a fully-fledged industry cluster manager, and then dive into the architecture debate.

Google's "Secret Sauce"

Verma, A., et al. 2015. Large-scale cluster management at Google with Borg. *In Proceedings of the 10th European Conference on Computer Systems (EuroSys)*: 18:1–18:17; http://dl.acm.org/citation.cfm?id=2741964.

This paper is mandatory reading for anyone who wants to understand what it takes to develop a fully-fledged cluster manager and to deploy it effectively. Borg handles all aspects of cluster orchestration: it monitors the health of machines, restarts failed jobs, deploys binaries and secrets, and oversubscribes resources just enough to maintain key SLOs (service-level objectives), while also leaving few resources to sit idle.

To achieve this, the Borg developers had to make many decisions: from choosing an isolation model (Google uses containers), to how packages are distributed to machines (via a torrent-like distribution tree), how tasks and jobs find each other (using a custom DNS-like naming system), why and how low-priority and high-priority workloads share the same underlying hardware (a combination of clever oversubscription, priority preemption, and a quota system), and even how to handle failures of the Borgmaster controller component (Paxos-based failover to a new leader replica). There is a treasure trove of neat tricks here (e.g., the automated estimation of a task's real resource needs in §5.5), as well as a ton of operational experience and sound distributed-system design.

A related Queue article describes how Borg and Omega, also developed at Google, have impacted Kubernetes, an open-source cluster manager developed with substantial inspiration from Borg. The features offered by Kubernetes are currently far more limited than Borg's, but Kubernetes catches up further with every release.

The actual *scheduling* of work to machines makes up only one subsection of the Borg paper (§3.2), but it is both challenging and crucially important. Plenty of standalone papers have been written solely about the scheduling problem. Unfortunately, they can at times be quite confusing: some describe schedulers that operate at a higher level than Borg, or ones designed for workloads that differ from Google's mix of long-running service jobs and finite-runtime batch-processing jobs. Fortunately, the next paper describes a neat way of separating some of these concerns.

Many Schedulers: Offers or Requests?

Hindman, B., et al. 2011. Mesos: a platform for fine-grained resource sharing in the data center. *In Proceedings of the Usenix Conference on Networked Systems Design and Implementation (NSDI)*: 295–308; http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf.

Mesos was the first academic publication on modern cluster management and scheduling. The implementation is open source and appeared at a time when Borg was still unknown outside Google. The Mesos authors had the key insight, later reaffirmed by the Borg paper, that dynamically sharing the underlying cluster among many different workloads and frameworks (e.g., Hadoop, Spark, and TensorFlow) is crucial to achieving high resource utilization.

Different frameworks often have different ideas of how independent work units (tasks) should be scheduled; indeed, the frameworks that Mesos targeted initially already had their own schedulers. To arbitrate resources among these frameworks without forcing them into the straitjacket of a single scheduling policy, Mesos neatly separates two concerns: a lower-level *resource manager* allocates resources to frameworks (e.g., subject to fairness constraints), and the higher-level *framework schedulers* choose which specific tasks to run where (e.g., respecting data locality preferences). An important consequence of this design is that Mesos can support long-running service tasks just as well as short, high-throughput batch-processing tasks—they are simply handled by different frameworks.

Mesos avoids having a complex API for frameworks to specify their resource needs. It does this by inverting the interaction between frameworks and the resource manager: instead of frameworks *requesting* resources, the Mesos resource manager *offers* resources to frameworks in turn. Each framework takes its pick, and the remaining resources are subsequently offered to the next one. Using appropriately sized offers, Mesos can also enforce fairness policies across frameworks, although this aspect has in practice turned out to be less important than perhaps originally anticipated.

The Mesos offer mechanism is somewhat controversial: the Omega paper observed that Mesos must offer all cluster resources to each framework to expose the full knowledge of existing state (including, for example, preemptible tasks), but that a slow framework scheduler can adversely affect other frameworks and overall cluster utilization in the absence of optimistic parallel offers and a conflict-resolution mechanism. In response, the Mesos developers have devised concepts for such extensions to Mesos.

Yet, Mesos's multischeduler design has been quite impactful: many other cluster managers have since adopted similar architectures, though they use either request-driven allocation (e.g., Hadoop YARN) or an Omega-style shared-state architecture (e.g., Microsoft's Apollo and HashiCorp's Nomad).

Another deliberate consequence of the offer-driven design is that the Mesos resource manager is fairly simple, which benefits its scalability. The next two papers look deeper into this concern: the first one proposes an even simpler design for even greater scalability, and the second suggests that scaling a complex scheduler is more feasible than widely thought.

Breaking the scheduler further apart

Ousterhout, K., et al. 2013. Sparrow: distributed, low-latency scheduling. *In Proceedings of the Symposium on Operating Systems Principles (SOSP)*: 69–84; http://dl.acm.org/citation.cfm?id=2522716.

Even within what Mesos considers framework-level tasks, there may be another level of scheduling. Specifically, some data-analytics systems break their processing into many short work units: Spark, for example, often generates application-level "tasks" that run for only a few hundred milliseconds (note that these are different from the *cluster-level* tasks that Borg or Mesos frameworks place!). Using such short tasks has many benefits: it implicitly balances load across the workers that process them; failures lose only a small amount of state; and straggler tasks that run much longer than others have smaller absolute impacts.

Shorter tasks, however, impose a higher load on the scheduler that places them. In practice, this is usually a framework-level scheduler, or a scheduler within the job itself (as in standalone Spark). With tens of thousands of tasks, this scheduler might get overwhelmed: it might simply be unable to support the decision throughput required. Indeed, the paper shows that the centralized application-level task scheduler within each Spark job scales to only about 1,500 tasks per second. Queueing tasks for assignment at a single scheduler hence increases their overall "makespan," as well as leaving resources idle while they await new tasks to be assigned by the overwhelmed scheduler.

Sparrow addresses this problem in a radical way: it builds task queues at each worker and breaks the scheduler into several distributed schedulers that populate these worker-side queues independently. Using the "power of two random choices" property, which says that (under certain assumptions) it suffices to poll two random queues to achieve a good load balance, Sparrow then randomly places tasks at workers. This requires neither state at the scheduler, nor communication between schedulers—and, hence, scales well by simply adding more schedulers.

This paper includes several important details that make the random-placement approach practical and bring it close to the choices that an omniscient scheduler would make to balance queue lengths perfectly. As an example, Sparrow speculatively enqueues a given task in several queues, spreading its bets across multiple workers and smoothing head-of-line blocking from other straggler tasks. It can also deal with placement constraints and offer weighted fair sharing across multiple jobs that share the same workers.

Sparrow could in principle be used as a cluster-level scheduler, but in practice works best load-balancing application-level tasks over long-running workers of a single framework (which, for example, serves queries or runs analytics jobs). At the cluster-scheduler level, the task startup overhead normally makes tasks below tens of seconds in duration impractical because of package distribution, container launch, etc. Consequently, the open-source Sparrow implementation supplies a Spark application-level scheduler plug-in.

Finally, while Sparrow's randomized, distributed decisions are scalable, they assume that tasks run within fixed-size resource slots and that queues of equal length amount to equally good choices. Several follow-up papers address some of these issues while maintaining the same distributed architecture for scalability and fault tolerance. One, however, looks at whether Sparrow-style wide distribution is really required for scalability.

Can We Have Quality *and* Speed?

Gog, I., et al. 2016. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of the Usenix Conference on Operating Systems Design and Implementation (OSDI)*: 99–115; https://www.usenix.org/system/files/conference/osdi16/osdi16-gog.pdf.

Distributed decisions have benefits in scalability and fault tolerance but must be made in the presence of reduced (and only statistically sampled) information about cluster state. Centralized schedulers, which make all decisions in the same place, have the information to apply more sophisticated algorithms—for example, to avoid overloaded machines. Of course, this applies both at the cluster level and to application-level schedulers.

This paper sets out to investigate whether—fault-tolerance benefits notwithstanding—distribution is indeed necessary for scalability. It notes that it is crucial to amortize the cost of decisions over many tasks, especially if the scheduler supports features that require reconsidering the existing placements, such as task preemption. Think about it this way: if the scheduler picked a task off a queue, looked at a whole bunch of machines, and did some complex calculations just to decide where to put a single task, it probably wouldn't scale well.

Firmament generalizes the Quincy scheduler, a cool—and sometimes-overlooked—system that uses a min-cost, max-flow constraint solver to schedule batch workloads. The constraint solver always schedules the *entire* cluster workload, not just waiting tasks. Because min-cost, max-flow solvers are highly optimized, their algorithms amortize the work well over many tasks.

Applied naively, however, the Quincy approach cannot scale to large workloads over thousands of machines—the constraint-solver runtime, which dominates scheduling latency, becomes unacceptably high. To fix this, Firmament concurrently runs several min-cost, max-flow algorithms with different properties and solves the optimization problem *incrementally* if possible, refining a previous solution rather than starting over.

With some additional tricks, Firmament achieves subsecond decision times even when scheduling a Google-scale cluster running hundreds of thousands of tasks. This allows Sparrow-style application-level tasks to be placed within hundreds of milliseconds in a centralized way even on thousands of machines. The paper also shows that there is no scalability-driven need for distributed cluster-level schedulers, as Firmament runs a 250-times-accelerated Google cluster trace with median task runtimes of only four seconds, while still making subsecond decisions in the common case. The simulator and Firmament itself are open-source, and there is a plug-in that allows Kubernetes to use Firmament as a scheduler.

The Firmament paper suggests that decision quality need not be compromised to solve a perceived scalability problem in cluster scheduling. Nevertheless, Sparrow-style distributed schedulers are useful: for example, a fault-tolerant application-level load balancer that serves a fixed set of equally powerful workers might well use Sparrow's architecture.

Research to practice

What does all this mean for you, the reader? For one, you are almost certainly using applications that run on Borg and other cluster managers every day. Moreover, if you are running a business that computes on large data or runs web applications (or, especially, both!), you will probably want the automation of a cluster manager. Many companies already do so and run their own installations of Mesos or Kubernetes on clusters of VMs provisioned on the cloud or on machines on their own premises.

The problem of scaling cluster managers and their schedulers to very large clusters, however, is one that most readers won't have to face: only a few dozen companies run such large clusters, and buying resources on AWS (Amazon Web Services) or Google's or Microsoft's clouds is the easiest way to scale. Yet, scheduler scalability can also be an issue in smaller clusters: if you are running interactive analytics workloads with short tasks, a scalable scheduler may give you better resource utilization.

Another important aspect is that cluster workloads are quite diverse, and scheduling policies in practice often require substantial hand-tuning using placement constraints. Indeed, this is what makes cluster scheduling different from the multiprocessor scheduling that your kernel does: while most applications are fine with the general-purpose policies the kernel applies to assign processes to cores, current cluster-level placement policies do not work well for all workload mixes without manual operator help.

Future directions

It is challenging for humans to develop scheduling policies and good placement heuristics that suit all (or even most) workloads, and research on policies that help in specific settings is guaranteed to continue.

Another approach, however, may also be viable. Recent, early results suggest that the abundant metrics data and the feedback loops of cluster-scheduling decisions are a good fit for modern machine-learning techniques: they allow training neural networks to automatically learn custom heuristics tailored to the workload. For example, reinforcement learning can effectively learn packing algorithms that match or outperform existing, human-specified heuristics, and a neural network outperforms human planners in TensorFlow operator scheduling.

Therefore, future research may raise the level of automation in cluster management even further: perhaps the cluster scheduler will someday learn its own algorithm.

Malte Schwarzkopf *is a post-doctoral associate in the PDOS (Parallel and Distributed Operating Systems) Group at MIT. His research focuses on distributed systems, with past work in cluster scheduling, data-center networking, and parallel data processing. He received both his B.A. and Ph.D. from the University of Cambridge, and his research has won awards from EuroSys (2013) and the USENIX Symposium on Networked Systems Design and Implementation (2015).*