Queueing Systems (2021) 97:3–37
https://doi.org/10.1007/s11134-020-09684-6

Open problems in queueing theory inspired by datacenter computing

Mor Harchol-Balter[1]

Received: 1 December 2020 / Revised: 1 December 2020 / Accepted: 4 December 2020 /
Published online: 27 January 2021

Abstract
Datacenter operations today provide a plethora of new queueing and scheduling problems. The notion of a "job" has become more general and multi-dimensional. The ways in which jobs and servers can interact have grown in complexity, involving parallelism, speedup functions, precedence constraints, and task graphs. The workloads are vastly more variable and more heavy-tailed. Even the performance metrics of interest are broader than in the past, with multi-dimensional service-level objectives in terms of tail probabilities. The purpose of this article is to expose queueing theorists to new models, while providing suggestions for many specific open problems of interest, as well as some insights into their potential solution.

Keywords Cloud computing · Tail probabilities · Speedup curve · Parallel scheduling · Multi-core · Heavy tails

Mathematics Subject Classification 60K25 · 60K30 · 68M20 · 90B36 · 91B32

1 Introduction

Most computing today happens within datacenters, often in the form of public clouds such as Amazon's EC2 [1], Windows Azure [4], and Google Compute Engine [3], or on private clouds. Global data center spending exceeds 150 billion dollars yearly [5]. Large datacenters typically consist of tens of thousands of servers, running jobs that process petabytes of data daily. Behind these jobs sit ever-demanding users, impatiently waiting for the result of their jobs.

This work was supported by: NSF-CMMI-1938909, NSF-CSR-1763701, NSF-XPS-1629444, and a Google 2020 Faculty Research Award.

✉ Mor Harchol-Balter
harchol@cs.cmu.edu

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

At the heart of the data center is the job scheduler, also known as the load balancer, dispatcher, or front-end router. The scheduler manages the jobs in the datacenter. It determines which jobs are given priority. It determines which jobs get assigned to which queues and servers. It determines to what degree each job is parallelized. It decides when jobs need to be restarted, preempted, or dropped. Because the work of the scheduler is so complex, companies sometimes deploy multiple schedulers for each datacenter [20,107].

The scheduler might have any number of goals that it is trying to optimize for. The scheduler might be looking at the user perspective; for example, trying to minimize the average user response time or the tail of response time. *Response time*, a.k.a., sojourn time, is the time from when a job first arrives until it completes service. On the other hand, the scheduler might be looking at the system perspective; for example, trying to minimize the number of servers in use or the total power consumption. Often there are multiple competing metrics that the scheduler is trying to trade-off between. Everywhere, the scheduler is dealing with queues: Which jobs to queue and which to serve? How to order the jobs within each queue? How to prioritize between different queues?

All of these computing questions are creating a new heyday for queueing theory. There are a multitude of new queueing problems generated from studying the operation of datacenters. In collaborating with several of the major cloud providers, we find that we are faced with many more queueing problems than we can answer. Many of these problems are similar to queueing theory models of the past, but there are changes to what jobs look like, what servers look like, what performance metrics we want to be optimizing, and even what workloads look like today. The purpose of this article is to expose queueing theorists to a few of the new problems.

Sections 2, 3, 4 and 5 deal with new job and server models. These new models originate from the fact that today's jobs are predominantly *parallel* jobs, which makes them different from the traditional one-server-per-job model. Section 6 characterizes today's computing workloads. We will see that the variability in job service requirements today is orders of magnitude higher than anything in the past. Section 7 focuses on today's performance metrics. While almost all papers in queueing theory focus on mean response time (or, equivalently, mean number in system), almost no one in industry cares for this metric. In Sect. 7, we describe today's metrics and what kinds of scheduling policies might be needed to optimize for these metrics.

This document is aimed at queueing theorists. As such, we purposely avoid discussion of lower-level computer systems details that we consider to be of secondary importance. One example of a simplification that we will make is to use the word "server" whenever we are talking about something that processes jobs. A server might refer to a whole machine, or it might refer to a single CPU core on a larger machine, or even just a single thread. If it seems immaterial to the modeling problem, we will simply use the word *server*, or write server/core, and gloss over the details.

2 Multiserver jobs

Traditional queueing theory is built upon models, such as the M/G/n model, that assume that each job runs on a single server. These models were representative of most jobs for a very long time. However, traditional one-server-per-job models are no longer representative of modern computer systems.

If we look at the large data centers at Google, Microsoft, and Facebook today, we see that most typically *a single job runs on multiple servers simultaneously*. One difference is that today's workloads include many machine learning jobs, all of which are highly parallel. For example, Google's Borg Scheduler [148] schedules many parallel machine learning jobs like TensorFlow [9,106] among servers within its data centers.

Figure 1 shows that jobs are very different in the number of servers that they request, and the difference can vary by five orders of magnitude. We refer to jobs that occupy multiple servers/cores as *multiserver jobs*. Multiserver jobs have always existed in the unique world of supercomputing centers, which were built to run very large-scale parallel simulations that cannot be run elsewhere. However now, even everyday ordinary jobs are multiserver jobs.

Figure 2 illustrates what we will refer to as *the multiserver job queueing model*. Here, there are a total of n servers. Jobs arrive with average rate λ and are served in First-Come-First-Served (FCFS) order. With probability p_i, an arrival is of class i. An arriving job of class i requests n_i servers and holds onto these servers for S_i time.

Note that the term *job size* is a little ambiguous in this model. We need to be clear whether we are referring to (i) the number of servers being requested (n_i), or (ii) the service duration (S_i), or (iii) the product of the two. When looking at the product, it is common to describe the size in units of CPU-seconds.

Importantly, we note that we are assuming that jobs are served in FCFS order. This is the default scheduling policy used throughout the cloud-computing industry when running multiserver jobs; see for example the CloudSim, iFogSim, EPSim and GridSim cloud computing simulators [108], or the Google Borg Scheduler [144]. There may be multiple priority classes, where class 1 jobs are all served before class 2 jobs, and so on, but within each class the jobs are served in FCFS order.

The multiserver job model is fundamentally different from the one-server-per-job model in that work conservation does not hold. In the one-server-per-job model, as long as there are enough jobs present, none of the n servers will be idle. By contrast,

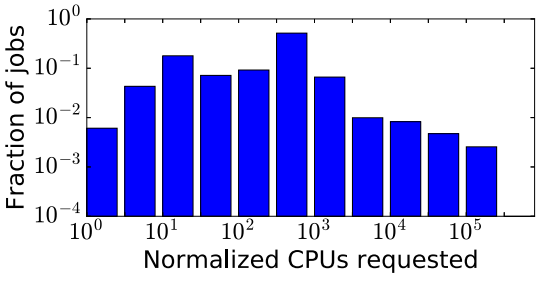

Fig. 1 Distribution of the number of CPU cores requested by individual Google jobs according to Google's recently published Borg Trace [144,158]. For privacy reasons, Google publishes only normalized numbers; however, it is still clear that the range spans 5 orders of magnitude across different jobs

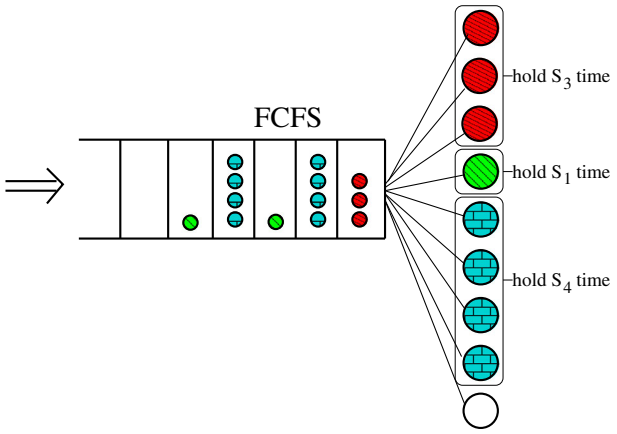

Fig. 2 Multiserver job queueing model with $n = 9$ servers. An arriving job of class i requests n_i servers and holds onto these servers for S_i time. In this particular illustration, $n_i = i$

in the multiserver job model, there may be servers idle because the job at the head of the queue does not "fit" into the available servers. (It needs more servers than are available.)[1] Consequently, server utilization and system stability both depend on the scheduling policy and are often far below ideal values. Specifically, ideally our stability region would be

$$\lambda \leq \frac{n}{\sum_i p_i n_i \mathbf{E}[S_i]}. \tag{1}$$

However, the *true* stability region can be far smaller than (1), depending on the particulars of the model (in particular the n_i's); half of all servers or more might be wasted [82].

At present almost nothing is known about the performance of the multiserver job model. A few works have attempted to derive the steady-state distribution of the number of jobs in the system in highly simplified systems, where all jobs have the same exponential service duration $S_i \sim \text{Exp}(\mu), \forall i$, and there are only $n = 2$ servers; see papers by Brill and Green [50] and Filippopoulos and Karatza [62]. However, the solutions are highly complex, typically involving roots to a quartic equation, which makes the solutions impractical. Earlier work by Kim [102], based on a matrix analytic approach, is similarly impractical since the complexity scales exponentially with the size of the system. In summary, understanding the mean response time for multiserver job systems with more than $n = 2$ servers is an entirely *open problem*, even when all jobs have the same exponentially distributed service durations.

Not only is the multiserver job model largely unstudied with respect to response time, but even the stability region for this model is only partially understood. In 2016,

[1] In the multiserver job model, we assume FCFS scheduling, which is what is used in datacenters. This is not to be confused with the virtual machine (VM) packing problem, where the literature has focused on packing jobs into VMs based on the number of resources that they request, so as to achieve throughput optimality (see [77,85,109,110,118]). However, even in the VM packing problem, waste can occur.

Rumyantsev and Morozov [123] derived the stability region for the multiserver job model where all jobs have the *same* exponential service duration $S_i \sim \text{Exp}(\mu)$, $\forall i$, although, unlike [50,62], they do allow for any number of servers n. This work was generalized in [114] and [15] to allow for more general arrival processes. However, the assumption that all jobs have the same exponential service duration has prevailed. Very recently, Grosof et al. [82] derived a simple closed-form expression for the stability region of the multiserver job model where jobs have *different* exponential service time durations. Unfortunately, the work in [82] is limited to the case of only two classes. Characterizing the stability region in a more general setting is an *open problem*.

Finally, we note that the multiserver job model opens up new scaling regimes where *both* the number of servers requested by a job and the system load scale with the total number of servers. Wang et al. [151] investigate the probability of queueing under this asymptotic scaling regime.

While very little is known about the performance of multiserver job models, there is a close cousin of the model, which we will call the *multiserver job dropping model*, or just *dropping model* for short, which is analytically tractable under very general settings. In the dropping model, jobs which cannot immediately receive service are dropped. The dropping model exhibits a beautiful product form when job durations (the S_i's) are exponentially distributed. Arthurs and Kaufman [24] were the first to observe the product form. Whitt [156] generalized the model to allow jobs to demand multiple resource types, while van Dijk [146] allowed durations to be generally distributed. Tikhonenko [143] combined aspects of [146,156]. An interesting *open problem* is whether anything can be said for the multiserver job model where there is a finite (but nonzero) queue capacity.

The multiserver job model has appeared in the literature under other names. It is closely related to *streaming models* for communication networks. In that setting, the resource being shared is bandwidth in the network. The "jobs" are audio or video flows which require a fixed bandwidth reservation to run. (This is akin to needing a fixed number of servers.) Flows requiring fixed bandwidth are often referred to as "streaming flows" (see [33]), but are also sometimes referred to as "inelastic jobs" (see [112,117]). The papers dealing with streaming flows largely operate in the dropping model, where the goal is to schedule to minimize a cost related to dropping probabilities (see [31,54,97]). Note that the setting here can be more complex than the multiserver job dropping model—sometimes because the authors are seeking an optimal dropping policy, and sometimes because the setting is an entire network.

Another variant of our multiserver job model is a model where the n_i servers of a job are held for different i.i.d. times. This is also a challenging problem; see [16]. Yet another related model, proposed by Baccelli and Foss [27], is the Poisson Hail model, where the servers are viewed as a single Euclidean space, and each job occupies a random interval of this space for some fixed period of time. Stability for Poisson Hail is studied in [27,63].

3 Speedup functions

In Sect. 2, we considered large datacenters, running parallel jobs, where the jobs explicitly request exactly the number of servers that they need. However, in the case of a smaller server farm or just a single multi-core machine, the number of servers/cores can be far more limited. Here, we do not have enough servers to allow every parallel job to specify the number of servers that it needs. Instead, it is the job of the operating system or scheduler to figure out *how to best allocate the limited number of servers/cores among the jobs* at every moment of time.

When determining a policy for sharing a set of servers among multiple jobs, it is important to understand exactly how a job benefits by receiving more servers. Fortunately, jobs are often *malleable*, meaning that they are designed to be runnable on any number of servers [51,57,86]. Parallelizing an individual job across multiple servers reduces its response time. In practice, however, a job typically receives *diminishing marginal returns* from being allocated more servers, because there is some overhead to running the job in parallel.

It was shown in [38,95] that many of the benefits and overheads of parallelization can be encapsulated in a job's *speedup function*, $s(k)$, which specifies a job's "speedup" when running on k servers. If we imagine that $s(1) = 1$, then a job which is allocated k servers will run $s(k)$ times faster than if the job were allocated just 1 server. Put another way, if we define the *inherent size* of a job to be its service time on a single server, then if x is the inherent size of a job, then $\frac{x}{s(k)}$ is the job's service time when run on k servers.[2]

Typically, $s(k)$ is concave and sublinear, as shown in Fig. 3a. The function

$$s(k) = k^p,$$

where $0 < p < 1$, is a commonly assumed speedup function; see [34,36,95,160].[3]

An alternative family of speedup functions is shown in Fig. 3b. These threshold-based speedup functions take the form

$$\begin{aligned} s(k) &= k \quad \text{for } k < k_0, \\ s(k) &= k_0 \quad \text{for } k \geq k_0, \end{aligned}$$

where $k_0 > 1$. Threshold-based speedup functions are motivated by the fact that the code associated with a job might only be parallelizable up to some point (k_0). Thus, giving the job more than k_0, servers/cores does not provide additional benefit.

The speedup function associated with a job is often something one can easily approximate just by looking at the code. In cases where it is very important to understand the speedup function exactly, one can run benchmarks where the job is run on different numbers of servers and its speedup is evaluated in each case. It is thus reasonable for us to assume that the speedup function is known.

[2] In the above example, we are thinking of the job as being run *alone* on the k servers. If two jobs are time-sharing the same k servers, then the service time of each will double.

[3] If $k < 1$, it is common to assume that $s(k) = k$, which is consistent with the intuition that if a job is allocated half a server, then it runs at half speed.

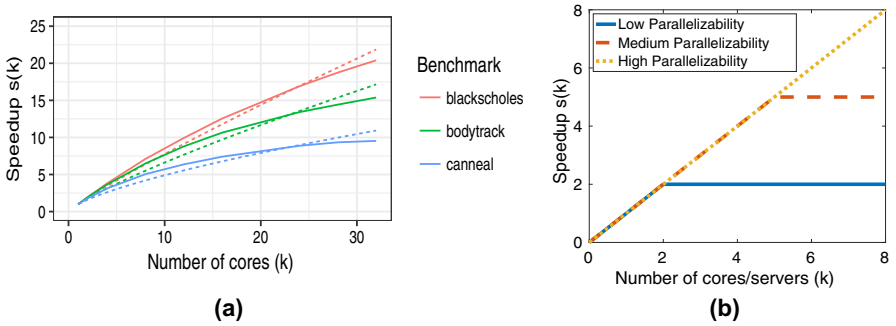

Fig. 3 Two types of speedup curves. The speedup curve, $s(k)$, depicts the speedup that a job obtains when run on k servers/cores. **a** This figure shows speedup curves of the form $s(k) = k^p$ (dotted lines) which have been fitted to speedup curves from real workloads (solid lines) measured from jobs in the PARSEC-3 parallel benchmarks [160]. The three workloads, blacksholes, bodytrack, and canneal, are best fit by the function $s(k) = k^p$, where $p = 0.89$, $p = 0.82$, and $p = 0.69$, respectively. **b** This figure shows various speedup curves of the threshold type, where the job is fully parallelizable given $k < k_0$ servers, but receives no speedup benefit beyond that threshold point k_0

3.1 The problem statement

Overall goal Typically, jobs arrive over time. There are a fixed number of servers, n, as well as a queue. Each job follows a speedup function, $s(k)$, which defines its parallelizability. The high-level goal is to determine, at every moment of time, what fraction of the n servers to allocate to each job, with the goal of minimizing mean response time, or some similar metric. Aside from the optimization problem, it is also useful to derive a formula for the mean response time, given a particular allocation algorithm.

Some optimization tradeoffs If jobs have increasing but concave speedup functions, then an individual job will benefit from being given more servers. However, the overall efficiency of the system will drop if we give too many servers to a single job. We need to balance this tradeoff. At the same time, we need to also consider the fact that some jobs might have much smaller inherent size than others, and we might want to bias in favor of these jobs to minimize mean response time.

Different settings of the problem There are several different settings one can consider, all of practical importance. Firstly, there is the question of whether different jobs have different speedup functions or whether they all have the same speedup function. It is often the case that some jobs are more parallelizable than others, thus having different speedup functions. However, if all jobs originate from the same workload, or are different instances of the same program, then they will all have the same speedup function.

Next, there is the question of whether a job's inherent size is known at the time when the job arrives. In the first case, the job's inherent size is known when the job arrives, or one has a reasonable guess as to its size, based on its name. In the second case, we have no knowledge of an individual job's inherent size. Despite not knowing the job's inherent size when it arrives, we might still be able to deduce something

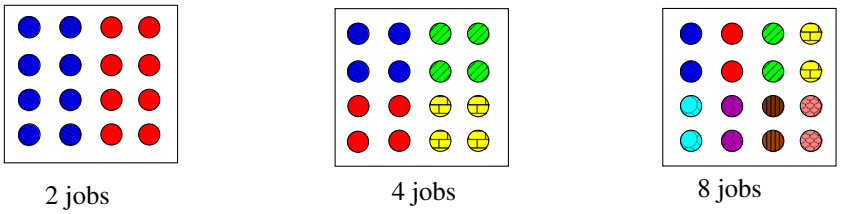

Fig. 4 EQUI policy shares servers equally among all available jobs

about its inherent size as it runs. Specifically, companies often maintain records of jobs that complete, so it is easy to formulate an approximate *distribution* of inherent job sizes. Let S be a random variable denoting the job's size. We define a job's *age*, a, to be the amount of work that the job has completed so far. When a job arrives ($a = 0$), its expected remaining inherent size is $\mathbf{E}[S]$. However, once the job achieves age a, its expected remaining inherent size is

$$\mathbf{E}[S - a \mid S > a] \ .$$

Now if the inherent job size distribution is exponential, we will not learn anything about a job's remaining size as it ages. But in other cases, we might learn a lot.

3.2 The most natural policy: EQUI

Given that speedup functions are concave, the most intuitive and simple allocation policy is a policy we call *EQUI*. Under EQUI, at all times, the n servers are divided equally among the jobs in the system. Specifically, whenever there are ℓ jobs in the system, each job is parallelized across n/ℓ servers, as shown in Fig. 4. (Jobs can be allocated fractional servers.) In the case where $\ell > n$, we can either imagine that the first n jobs each get 1 server and the others wait in a queue, or that all the jobs share the servers, with each getting a fractional piece (n/ℓ) of a server.

The intuition behind EQUI is very clear: EQUI keeps every job running at the most "profitable" part of the speedup curve. For any concave and sublinear speedup function, EQUI maximizes the *system efficiency*. When job sizes are exponentially distributed, and arrivals are Poisson, it is easy to see that for any system state (number of jobs), the rate at which EQUI completes work is at least as high as any other allocation.

Berg et al. [34] prove that EQUI minimizes mean response time under the broad setting where all jobs follow the same concave, sublinear speedup function $s(k)$, and all have sizes which are unknown but exponentially distributed, and the arrival process is a Poisson process with rate λ. If we consider the same setting as in [34], but where the inherent job size distribution is not exponential, then finding the optimal server allocation policy is an *open problem*. As we will see in Sect. 3.3, it is very likely that EQUI is *not* the optimal policy here.

3.3 Where EQUI fails: when sizes are known

In this section, we continue to assume that all jobs follow the same speedup function, but now we assume that the inherent job sizes are all known. Optimizing mean response time in this setting is still a largely *open problem*.

To understand why this setting is so complicated, let us consider a very simple example. Suppose that there are only two jobs, both present at time 0, one of which has twice the inherent size of the other, where both jobs follow the same concave, sublinear speedup function, $s(k)$. Which job should be allocated more servers? At first, one might think that the large job should be allocated twice as many servers as the small one. But the opposite is true: the small job should be allocated most of the servers. It is best to get our intuition from the Shortest-Remaining-Processing-Time (SRPT) policy, which at all times runs that job with the Shortest-Remaining-Processing-Time. SRPT is defined for the case where there is only one server, and it is known to be optimal for minimizing mean response time in that *single-server* case; see [124]. Observe that SRPT is also optimal if all jobs are fully parallelizable ($s(k) = k, \ \forall k$), because that is equivalent to the 1-server case, in that all n servers can be viewed as a single server. However, given that jobs are *not* fully parallelizable, we no longer want to follow a *strictly* SRPT policy, where we allocate *all* servers to the smaller job (because that can waste servers), but we still want to give the majority of servers to the smaller job.

To further hone our intuition, let us consider a second example. Again suppose that there are two jobs, where this time the jobs have the exact same inherent size. Again both jobs follow the same concave, sublinear speedup function, $s(k)$. Now our intuition tells us that the servers should be split evenly between the two jobs. However, this intuition turns out to be false, and, as we will see, it is once again typically better to give one of the jobs a significantly larger fraction of the servers. The intuition behind this is subtle. Although the two jobs start out with the same inherent size, by allocating more servers to one of the jobs, say job j, we can quickly reduce the remaining inherent size of job j, making job j now more attractive to serve than the other job. More intuition can be gleaned by again looking at a single server, where it is known that in the case of equal-sized jobs Processor-Sharing (PS) scheduling (which is akin to EQUI in the multi-server case) is sub-optimal compared to FCFS[4] [90, p.483].

In Sect. 3.2, we saw that when the inherent job sizes are not known and are exponentially distributed, then EQUI is optimal. The intuition was that given a concave, sublinear speedup function, EQUI keeps every job operating at the most profitable part of its speedup curve. By contrast, the above intuitions tell us that if the inherent job sizes are known (even if they're all equal), then one wants more of an SRPT-like policy. These intuitions are formalized in a 2020 paper, [36], which produces an exact and simple formula for the optimal allocation[5] in the case where all jobs are present at time 0 and all jobs share the same speedup function, $s(k) = k^p$. As explained in [36], because the jobs are *partially* parallelizable, what's needed is a *mixture* of SRPT and EQUI, which the authors call *"high-efficiency SRPT" (heSRPT)*. Under heSRPT,

[4] Note that SRPT and FCFS are equivalent in the case where all jobs have the same size.

[5] The optimal allocation is derived both for the case where the goal is to minimize mean response time and the case where the goal is to minimize mean slowdown. The slowdown metric is discussed in Sect. 7.1.3.

every single job at all times receives some share of the n servers, but the jobs with a smaller inherent size receive a higher share.

The exact specification of heSRPT provided in [36] is given in terms of a formula which specifies, at every moment of time t, a vector $\theta^*(t) = (\theta_1^*(t), \theta_2^*(t), \ldots, \theta_M^*(t))$ that says what fraction of the total service capacity (the n servers) should be allocated to each job, where M denotes the original number of jobs present at time 0. As an example, consider the case where the speedup function is $s(k) = k^{0.5}$. If there are only two jobs of equal inherent size, then the optimal policy for minimizing mean response time specifies that $\theta^*(t) = \left(\frac{3}{4}, \frac{1}{4}\right)$ until the time t where one of the jobs completes, at which time the remaining job is allocated all the servers. As another example, under the same speedup function, suppose that there are three jobs, where one has inherent size x and the other two both have size $2x$. Now, $\theta^*(t) = \left(\frac{5}{9}, \frac{3}{9}, \frac{1}{9}\right)$, where the job of size x is the one given the largest share ($\frac{5}{9}$), until that time t when the job of size x completes. At that time there are two jobs remaining, and these are allocated shares according to $\theta^*(t) = \left(\frac{3}{4}, \frac{1}{4}\right)$. Interestingly, under heSRPT, the specific inherent sizes of the jobs do not matter; it is only the ordering of the sizes of the jobs that affects the allocation.

We have seen that heSRPT optimizes mean response time in the case where all jobs are present at time 0 and the speedup function has the form $s(k) = k^p$. However if the speedup function is any non-trivial function other than $s(k) = k^p$ or the simple threshold-based function, then it is an entirely *open problem* to determine how to optimally allocate servers to jobs. Hopefully, the above intuition still hold though.

An even bigger *open problem* is the question of how to allocate servers to jobs when jobs arrive over time, as in a Poisson process. One could of course try running an online version of heSRPT. While the online version of heSRPT is not bad [36], it is not optimal either. Allowing arrivals over time adds a great deal of complexity to the optimization problem, since we now have to also consider how many jobs we want to complete before the next arrival. Even for the simplest speedup functions (for example, threshold-based speedup), once we add in arrivals, the optimal allocation is an *open problem*.

3.4 When jobs have different speedup functions

In this section, we examine the important case where jobs have different speedup functions. Here, almost everything is open, but there is some intuition that can be learned from the little work that exists.

The simplest version involves going back to our setup in Sect. 3.2, where we assume that jobs' inherent sizes are unknown and are exponentially distributed. Recall that in the case of a *single* speedup function, the optimal policy is EQUI, whose optimality stems from the fact that it maximizes the rate of departures in every state. We now imagine we have *two* classes of jobs, where class i has speedup function s_i, and where class 1 jobs are *less parallelizable* than class 2 jobs, i.e.,

$$s_1(k) < s_2(k), \qquad \forall k.$$

With two speedup functions, EQUI clearly no longer maximizes the rate of departures in every state. At first, we might be tempted to give all the servers to the class 2 jobs (the more parallelizable jobs), but what if class 2 jobs are already saturated? To maximize the rate of departures, it seems we really want to allocate each server to that job that will get the most benefit from that additional server. So, for example, if all jobs of class 2 are already at their saturated point, where they do not benefit much from an incremental server, we would instead allocate the server to a job of class 1, whose *differential* performance can still stand to improve a lot.

In [34], the authors define the GREEDY class to be those policies that achieve the maximum total rate of departures in every state. Importantly, GREEDY is truly a *class* of policies since there may be multiple allocations that all achieve the maximum departure rate, given that jobs are at different points on their speedup curves. At first, it might seem that any GREEDY policy should minimize mean response time. However, consider this thought experiment. Imagine that we are in state (n_1, n_2), where n_i denotes the number of jobs of class i. Now consider two policies, P_1 and P_2, where *both* are GREEDY policies. Suppose that in state (n_1, n_2), P_1 runs more class 1 jobs (and fewer class 2 jobs) than P_2. Then, in some sense P_1 seems preferable to P_2 because it next moves into a state that has more parallelizable work left (more class 2 jobs). The fact that P_1 *defers parallelizable work* is desirable because the class 2 jobs are more exploitable, particularly if we later enter a state with very few jobs, since class 2 jobs can obtain more benefit from using all available servers.

It turns out that *both* maximizing the rate of departures *and* deferring parallelizable work are important characteristics in an optimal policy. The authors in [34] propose a policy GREEDY*, which is the GREEDY policy where in all states we pick the option that maximally defers parallelizable work (when possible). Every intuition would tell us that GREEDY* should be optimal. But this is still not true! It turns out that sometimes the optimal policy is not a GREEDY one. Specifically, deferring parallelizable work is *so* advantageous that it is worth deferring extra parallelizable work at the expense of a slightly suboptimal rate of departures. Experiments indicate that GREEDY*'s mean response time is within 1–2% of optimal [34].

The above shows why the case of two speedup functions is so complex. Even in the simplest case described above, where there are only two strictly ranked speedup functions, and where all jobs have inherent sizes drawn from the same exponential distribution, finding the optimal allocation policy is still an *open problem*.

One direction where progress might be possible is to consider more specific, simpler forms of speedup functions that are still realistic, for example, threshold-based speedup functions (see Fig. 3b). We have observed such threshold-based speedup classes in database workloads (specifically TPC-H workloads). In [35], the authors consider two classes of jobs: class 1 with a threshold-based speedup, and class 2 which is fully *elastic*, i.e. $s_2(k) = k, \ \forall k$. A further assumption is made that class 1 jobs have smaller inherent size than class 2 jobs. In this setting, GREEDY* is the policy that gives strict priority to class 1 jobs, and [35] proves that GREEDY* is optimal here.

A similar situation, where there is an elastic class of jobs as well as an inelastic class, also comes up in bandwidth sharing (see [32,33,45]). Here, the *elastic jobs* are data flows, for example, file transfers. A file can be sent at low bandwidth, taking a long time to complete the transfer, or it can be sent at high bandwidth, taking a short time

to complete the transfer. The file transfer speed scales linearly with the bandwidth allocated [111]. The *inelastic jobs* here are typically referred to as streaming jobs. They are voice or video calls that require a specific fixed amount of bandwidth. (This is different from a threshold speedup function.)

In general, it is common that one has many types of jobs, each with a different speedup function and each with a different distribution of inherent size. It is a huge *open problem* to figure out how to allocate servers to jobs in this setting. Industry is currently lagging behind research. Industry uses ad hoc solutions, where the upper bound on the number of servers allocated to each job is a tuned ("voodoo") parameter. Having discussed this problem with several major companies, it is clear that companies have not yet figured out how to make use of the individual speedup functions of jobs, nor their inherent sizes, in determining the optimal allocation, both factors which we have found to be critical. Even machine learning approaches aimed at "learning" the optimal allocation do not yet take these critical factors into account [61].

4 Parallel DAG jobs and serverless computing

For many jobs, the level of parallelization will change over time. For example, a database query might consist of a first phase that can be fully parallelized across all n servers (with linear speedup), followed by a second phase where all the results from the first phase need to be joined, which has to be done serially (no parallelization), followed by a third phase where only partial parallelization is possible.

In traditional cloud computing, as described in Sect. 2, the user needs to specify a number of servers (typically virtual machines, VMs) on which her job will run. For jobs with changing levels of parallelization, the user will choose a number of servers which is the *maximum* of what is needed in any phase of the job. This clearly results in a lot of waste as VMs are left idle. Furthermore, the user is charged for VMs that she is often not using.

In an effort to reduce this waste, in recent years, cloud providers have introduced *serverless computing*, for example, AWS Lambda, Azure Functions, Google Cloud Functions. Recent studies report that 74% of enterprises that use the cloud are already using or experimenting with serverless [6], and projections forecast that most of the applications currently in the cloud will transition to using serverless computing in the near future [25,100,152]. These predictions have resulted in a flurry of computer systems research on enabling a broader range of serverless applications [66,99,131, 137,161].

In serverless computing, a user no longer requests and pays for a fixed number of VMs. Instead serverless frameworks like AWS Step [26] allow a user to express her job as a Directed Acyclic Graph (DAG) of functions (tasks). The responsibility of allocating servers to these tasks is relegated to the computing system which is being shared by many users running many jobs. The user is only charged for the resources actually used by her job, where the job's resource needs are allowed to increase and decrease as the job runs.

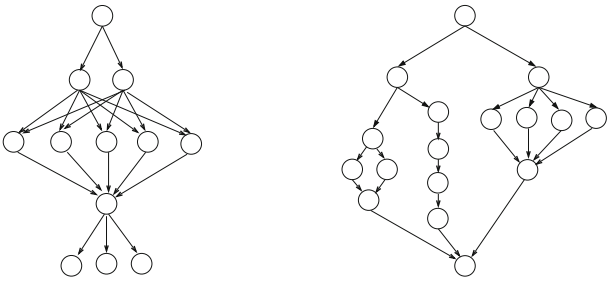

Fig. 5 Examples of DAG jobs

The DAG job model Before, we can describe the server allocation problem for serverless computing, we need to be clear on the job model. A job is described by a DAG. Examples of DAGs are shown in Fig. 5.

There are a few things to note about the DAG: First, a job refers to the entire DAG. The job is broken up into many independent *tasks*, where every node in the DAG is a task. (In serverless computing, the tasks are referred to as functions.) The DAG is a directed graph, specifying precedence relations (an ordering) for running the tasks:

- A *layered DAG* has explicit levels. For example, the DAG in Fig. 5a has five levels. All the tasks within a level can be run in parallel. (However, we can choose to run these serially, or run only a subset in parallel.) All tasks within a level need to be completed before even a single task at the next level can be started.
- A more general DAG does not have explicit levels. There is a lot more flexibility on the ordering of performing the tasks. Figure 5b shows a DAG with two branches that do not depend on each other at all, so we have a lot of leeway on scheduling these.

Importantly, a job is not considered to be complete until *all* the tasks in its DAG are complete.

The DAG scheduling problem One can now imagine that DAG jobs arrive over time, and, as usual, we need to share our n servers among all the jobs. Our goal again is to minimize mean response time across jobs. Now, at every moment of time, we need to decide not only which jobs to run (we can imagine that the others wait in a queue) but also which tasks within each job to run. Here, we assume that one task fully occupies one server.

The DAG model differs from the speedup model that we saw in Sect. 3, because the extent to which a job can be parallelized is specified by its DAG. Nonetheless, we still have a lot of flexibility in choosing which jobs to run, and which tasks within those jobs.

The DAG scheduling problem is as general as they come: different jobs can have differently shaped DAGs; the tasks within a DAG can have different sizes (service times); the sizes of the tasks can be known or unknown; even the DAG structure itself (number of levels, etc.) can be known or unknown.

In practice, however, it is far more typical that all tasks within the same level have the same size (or similar sizes). It is also common in practice that one has a good estimate for what this size is. So, for example, if all k tasks within a level have size 1, and they are run in parallel (on different servers), then after time 1 all k will complete. It is also common that one either knows the DAG associated with each job in advance, or that the DAG is "revealed" as the job runs, one level at a time.

Even with these simplifications, finding the optimal allocation policy is a very hard *open problem*, and analyzing the mean job response time of different allocation policies is also a hard *open problem*. Intuitively, since one does not get credit for completing a job until all its tasks are complete, it does not make sense to start up too many jobs at once. Also, following the usual principle of "shortest-job-first," and noting that the number of levels of a job dictates its minimum runtime, it may make sense to prioritize jobs with a small number of levels or a small amount of total work.

DAG scheduling has received a good deal of attention from the theoretical computer science community; see, for example, [14,17–19,23,39–44,52,67,80,115,134,147]. However the theoretical computer science community tends to think in terms of a *worst-case model*, where arrivals and the job structures are adversarially chosen, whereas in computing centers, it is more realistic to view the arrival times, job structures, and sizes as being drawn from some distributions.

5 Limited fork–join model for parallel jobs

In all the prior sections, we considered parallel job models where there was a single centralized queue to hold jobs. All the scheduling was also heavily centralized: a single controller had full ownership of n servers and could make decisions on how to allocate the n servers across the parallel jobs.

In this section, we consider a different parallel setting. While we are still dealing with parallel jobs, composed of tasks, the servers are now distributed, and each server has its own queue. The only control we have is on how we route each job's tasks to the different queues.

Our model in this section is motivated by the popular MapReduce framework [56]. MapReduce is an important model in many big data processing applications such as search indexing, distribution sorts, log analysis, and machine learning. In MapReduce, every job is divided into independent tasks that can be run in parallel in any order (the "map" phase). Once all the tasks of a job complete, their results need to be joined together (the "reduce" phase). Figure 6 provides an illustration of the map phase. Jobs arrive over time. When a job arrives, each of its tasks is dispatched to a different FCFS queue. The different tasks will thus likely incur different queueing times; so even if their sizes are similar, their individual response times can be different. The *response time of a job* is the maximum of the *response times* of its tasks.

It is worth taking a minute to talk about the *service time*, a.k.a. *runtime* of a task, i.e., the time from when the task gets the server until it completes. A task's runtime has two components. First, there is the *inherent size* of the task (in seconds), denoted by random variable X. Next, there is a *server slowdown factor* associated with the particular server on which the task is running, at the time when that task runs, denoted

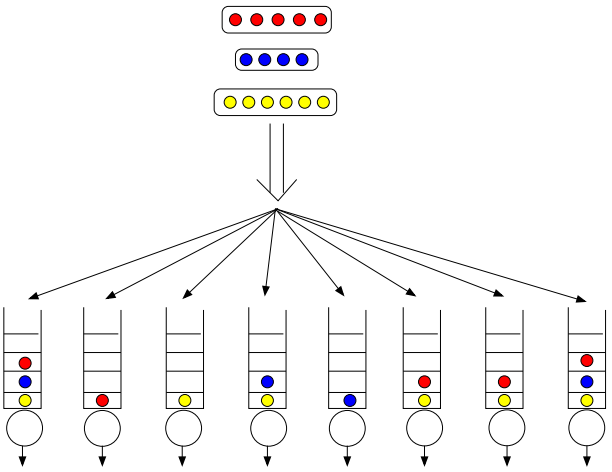

Fig. 6 Limited fork–join model with $n = 8$ queues

by $S > 1$. Even if all servers have the same speeds, the fact that the servers are distributed means that the servers may be experiencing different conditions. A task might find that its server is currently going through garbage collection, or is being slowed down by some other process running on it in the background. The runtime of the task at a particular server is best modeled by the product $S \cdot X$; see [72] for details.

Given a job with k tasks, the dispatcher might choose to send the job's tasks to the k shortest of the n queues, or it might choose to send the tasks to k random queues. The common practice is to send the tasks to those servers that have the data that they need. If a task is sent to a server that does not have the data that it needs, then there is an additional transfer cost needed to bring over the needed data, which takes time.

Goals Our goal in this setting is twofold. Firstly, from an optimization perspective, we want to dispatch a job's k tasks to queues so as to minimize the response time across *jobs*. Secondly, given a particular dispatching policy, like, "Send tasks to random queues," we would like to derive the mean *job* response time. Studying the performance of parallel jobs in a distributed server setting is extremely complex. It is therefore common to create *theoretical abstractions*, where we ignore certain issues like server slowdowns or data location.

The classic *fork-join model* is one theoretical abstraction which deals with parallel jobs in a distributed server setting. In the fork–join model, jobs arrive over time and each job consists of $k = n$ tasks which each join one of the n queues. The job is considered to be complete only when all of its tasks complete. In the fork–join model, if all the tasks had exactly the same size, then all the queues would be synchronized and the analysis would be trivial. But when tasks have different sizes, even exponentially-distributed sizes, the queues can quickly start to look different from each other. The classic fork–join model has been widely studied. Unfortunately, tight characterizations of job response time are unknown except when $n = 2$. See [141] for a survey. It has been proven that the mean delay of a job scales as $\Theta(\ln(n))$ as $n \to \infty$ under proper assumptions [28,29,116], but a tight characterization of the constant in the $\Theta(\ln(n))$ is not known.

The MapReduce model is different from the classic fork–join model because k, the number of tasks associated with a job, is typically far smaller than the number of servers, n. Large data centers can easily have $n = 10{,}000$ servers or even $n = 100{,}000$ servers. By contrast, the MapReduce jobs might have $k = 100$ tasks or $k = 1000$ tasks. A more realistic theoretical abstraction is therefore the *limited fork–join model* where jobs have $k \ll n$ tasks.

Fortunately, the limited fork–join model can be much more analytically tractable than the class fork–join model. Rizk et al. [122], Lee et al. [103], and Wang et al. [150] all give bounds on mean response time assuming that the tasks of a job are randomly dispatched, as well as some other conditions. The Wang et al. [150] result shows that when the jobs do not have too many tasks, specifically jobs have $k = o\left(n^{\frac{1}{4}}\right)$ tasks, then the queues are asymptotically independent (that is, they become independent for $n \to \infty$). Wang et al. also prove that independence leads to an upper bound on overall job response time; this upper bound turns out to be good when n is not very large.

While good progress has been made on analyzing the limited fork–join model, *open problems* remain. It is unclear how much larger k can be made while still achieving asymptotic independence. If the queues are not asymptotically independent, then a different technique is needed to analyze mean job response time. Analyzing the case where the k tasks are dispatched to the k shortest queues, or k queues with least work is also an interesting *open problem*, with excellent recent progress in [132,154].

Of course there are far more *open problems* once one takes practical conditions into account. As stated earlier, the routing of tasks to servers should factor in the fact that the servers might be operating at different speeds and the fact that they have different data stored. If a task is routed to a server that does not have the data that it needs, a time cost must be specified for migrating that data. Finally, there is of course the whole question of how to store data in the first place, given that servers typically have limited space.

Finally, there is an interesting related problem which comes up in practice when implementing MapReduce and other similar models. It is common that one of the k tasks takes far longer than the other tasks to complete. This could be because the task was simply larger. (It is not always possible to subdivide a job into equal-sized tasks.) However, this can also happen for many other reasons: (i) the task might be dispatched to a longer queue; (ii) the other tasks in its queue might turn out to be exceptionally large; (iii) the server to which the task is sent might be temporarily slowed down for some reason. The fact that even just one task is taking an exceptionally long time will cause the job's response time to be high and can also impede many other jobs. In an effort to reduce the tail due to one exceptionally lengthy task response time, it is common to replicate just that task. Specifically, that task is restarted from scratch at another queue, in the hope that it will experience a better task response time. As soon as either the original task completes or the replica completes, the task is considered to be done. Replication (a.k.a. "redundancy") is both very powerful in reducing response times, but also can be very dangerous because it adds work to the system, which can in turn hurt response time [72]. While many theoretical papers have been written on simple redundancy models where the "job" is a single task (for example, [21,22,72–75,101,119–121,138]), mixing redundancy with limited fork–join makes the analysis

a lot harder, and there is room for a lot more research in this space. Some preliminary work was done recently by Wang, Joshi, and Wornell [149] in the highly simplified model where the servers have no queues.

6 Computing workloads: extreme job size variability and heavy tails

An important component of both performance and systems modeling is the workload. Fortunately, we do not have to guess what modern workloads look like, since many companies publish traces which allow us to understand their jobs; see, for example, Google [157], Microsoft [2], and Alibaba [142]. In this section, we will examine jobs from a new 2019 trace [158] that shows all jobs run during May 2019 from eight different Google compute clusters. Much of our description will follow a paper by Tirmazi et al. [144] that examines this trace.

The first thing to note about Google jobs is that they are multi-dimensional. A job holds onto a certain number of processors (CPUs) and a certain amount of memory for a certain amount of time. The *resource consumption* of a job is described in CPU-hours (number of CPUs times hours held) and in Memory-Unit-hours (number of memory units times hours held). Because Google does not like to reveal exact numbers, it uses normalized units in expressing compute and memory usage. Thus, per-job compute usage is expressed in units of NCU-hours (normalized CPU times hours) and per-job memory usage is expressed in units of NMU-hours (normalized memory units times hours). Note that a 100 NCU-hour job might have consumed 100 machines for 1 h, or 5 machines for 20 h, or various other combinations.

6.1 Compute usage in data centers

Figure 7a shows the distribution of compute usage. The fact that we see a reasonably straight line on a log–log scale tells us that compute usage follows a Pareto(α)

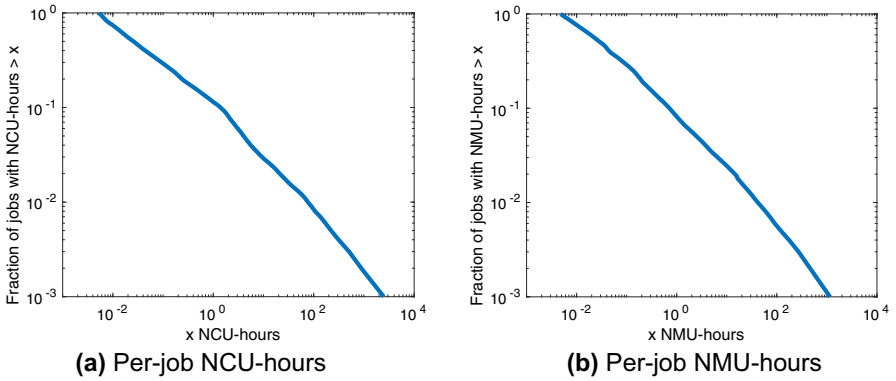

Fig. 7 CCDF of resource usage based on the Google 2019 trace of millions of jobs run at Google in May 2019 [144,158]. NCU-hours denotes Normalized CPU-hours used. NMU-hours denotes Normalized Memory-Unit-hours used

distribution, where α is the negative slope of this line. That is

$$\mathbf{P}\{\text{job uses } > x \text{ NCU-hours}\} = x^{-\alpha},$$

where $-\alpha$ is the slope of the line. For the Google jobs, $\alpha = 0.69$, which is quite small. This indicates that the distribution is quite variable and extremely heavy-tailed. For background on heavy-tailed distributions, see [65,133].

To be specific, we find that, while the mean NCU-hours used per job is about 1.2, the variance is 33,300, which means that the squared coefficient of variation is

$$C^2 = \frac{\text{variance}}{\text{mean}^2} = 23{,}000,$$

which is huge! To put this in perspective, most queueing papers are based on exponential job size distributions, which have $C^2 = 1$. In 1996, measurements of compute consumption in UNIX jobs at U.C. Berkeley found $C^2 = 50$ [92,93]. A few studies of job sizes at supercomputing centers in 2004 and 2005 found C^2 in the range from 28 to 256 [105,126]. The compute variability across jobs seen at Google today is several orders of magnitude higher than these numbers.

It is well-known that Pareto(α) distributions, particularly those with $\alpha < 1$ exhibit a strong *heavy-tailed property*, whereby a small fraction of the very largest jobs comprise most of the load. In prior empirical studies of compute consumption and file sizes [53,87,88,92,94], the authors found that the top 1% of jobs comprise 50% of the load. This is much more extreme than the oft quoted "80–20 rule," where the largest 20% of the jobs comprise 80% of the load. The heavy-tailed property exhibited in Google's data centers today is even more extreme than what was seen in [53,87,88,92,94]. For Google jobs today, the largest (most compute-intensive) 1% of jobs comprise about 99% of the compute load (see [144])!

6.2 Memory usage in data centers

Memory usage at Google's data centers follows much the same patterns as compute usage. Figure 7b shows the distribution of memory usage. Again, we see a Pareto(α) distribution, where this time $\alpha = 0.72$. Again, we see astronomical variability in the memory usage: $C^2 \approx 43{,}000$. Again, we see an extremely strong "heavy-tailed property", whereby the top 1% of jobs comprise about 99% of the total memory usage. In [144], it is further shown that there is a positive correlation between memory usage and compute usage.

6.3 Some implications for scheduling

Scheduling of jobs at most companies basically follows First-Come-First-Serve (FCFS). In particular, a job's dimensions (number of servers needed, duration, memory

needs) are typically not taken into account in scheduling. As an example, the Google Borg datacenter scheduler runs one large central FCFS queue. The jobs in the queue are tiered in that "production jobs" have higher priority than "batch jobs." However, within a tier, the jobs are largely served FCFS, very much following our description in Sect. 2. Each job has a CPU requirement (number of CPUs) and a memory requirement (number of Memory Units) and a duration (service time requirement). The duration is typically not known, although it certainly might be guessable based on the job name, or it might be learnable as a job runs (as we saw in Sect. 3.1). However, the CPU and memory requirements are both known a priori.

The extremely high variability in compute usage (and memory usage) that we saw in the previous sections holds both across tiers, but also within a single tier. This is particularly problematic for the batch jobs, which already have lower priority. Given the extremely high variability across batch jobs, it seems quite likely that small batch jobs (the "mice") are getting stuck waiting in the queue behind large batch jobs (the resource "hogs"). Thus in terms of response time, the mice are essentially inheriting the large size of the hogs (that size is adding to their response time). From the perspective of minimizing mean response time, it makes much more sense to isolate the mice from the hogs.

One solution is to give the mice priority over the hogs (as in scheduling policies like Shortest-Job-First). However, it might be the case that the hogs are also the most important jobs (hogs tend to be large machine learning jobs), and thus we do not want to bias against these.

A better solution is to physically separate the mice from the hogs in accordance with the Size-Interval-Task-Assignment (SITA) scheme proposed in [53,91]. In the context of the heavy-tailed property, this might mean creating one region of the data center for the 99% smallest jobs (the mice), and a separate region of the datacenter for the 1% largest jobs (the hogs). The mice would then be scheduled FCFS into the mice region of the datacenter, and the hogs would be scheduled FCFS into the hogs region. Given that the mice comprise only about 1% of the total compute usage, the mice region of the datacenter would be tiny (on the order of 1% of the servers), while the hogs region would comprise about 99% of the datacenter. In this way, the hogs would not be penalized, but the mice could receive some isolation.

There are of course other practical considerations that one would need to worry about. First of all, one would need to make sure that the mice could fit into the mice region. Specifically, if a mouse needs a large number of servers, but for a very short time, the mouse region needs to have access to at least that number of servers. Another practical consideration is that one might not know which jobs are mice versus hogs because one does not know the duration of jobs a priori. Such an issue was addressed in [89]. The basic idea is to assume that all jobs are mice until their compute usage exceeds some threshold. At that point, the job is deemed to be a hog (and could be moved to the hog section).

7 Performance metrics

Mean response time (or, equivalently, the mean number of jobs in the system) is the performance metric that receives the most attention in queueing theory papers. As always, a job's *response time* is the time from when it arrives to the system until it completes service (a.k.a. sojourn time, or time in system). However, in the computing industry, the performance metrics of interest are often quite different from mean response time. This section describes performance metrics that we see in industry.

When we talk about a performance metric, we have in mind two viewpoints:

- *Analysis of the metric* Here, we are interested in being able to derive a certain performance metric. If, for example, our metric is mean response time, we would be looking for a way to derive mean response time for the system.
- *Optimizing the metric* Here, we are interested in the optimal scheduling policy for achieving a performance goal. If, for example, our metric is mean response time, we would be looking for the scheduling policy that minimizes mean response time.

For each performance metric, we describe what is known and what open problems exist. For completeness, we start with the simplest metric, mean response time, where the most is known (but open problems still exist), and then branch out to other more popular industry metrics, where much less is known. This section is by no means a complete list of all metrics; we have tried to focus on what we see most in the computing industry.

All the metrics in this section apply both to parallel jobs (Sects. 2, 3, 4, 5) and to serial jobs. However, to better align with prior work, which is mostly in the serial one-server-per-job model, we will describe the metrics in terms of serial jobs. In particular, when we refer to a *job's size*, we will be talking about its service requirement in terms of just time. Throughout, we will assume that jobs arrive to our system over time, for example, according to a Poisson process.

7.1 The simplest metric: mean response time

Because mean response time is so strongly related to job sizes, it helps to differentiate between the case where a job's size is known when it arrives to the system, versus the case where job sizes are not known a priori.

7.1.1 If sizes known

There are cases in computing where the job sizes are known at the time when a job arrives. For example, if the job consists of downloading a file, then the size of the job is typically proportional to the size of the file. When job sizes are known, and one is dealing with a single-server system, the optimal scheduling policy is well-known to be Shortest-Remaining-Processing-Time (SRPT) [124]. This is true not just for an M/G/1 system, but even when the arrival process (arrival times and job sizes) is adversarially chosen. SRPT at all times preemptively runs that job with the shortest

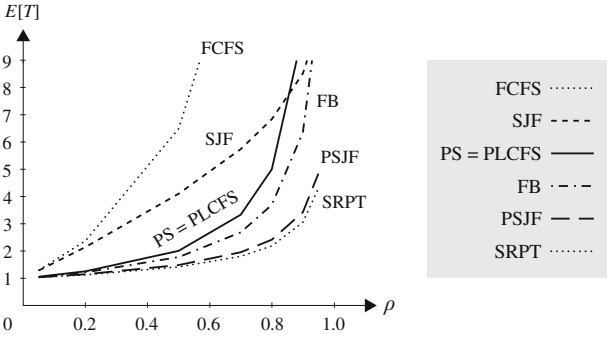

Fig. 8 Mean response time, $\mathbf{E}[T]$, as a function of load for the M/G/1 with various scheduling policies. The job size distribution is a Weibull with mean 1 and $C^2 = 10$. This figure taken from [90, p. 524]. Here, SJF refers to (non-preemptive) Shortest Job First, while PSJF refers to Preemptive Shortest Job First. FB is the Foreground-Background policy. PS is Processor-Sharing and PLCFS is Preemptive Last-Come-First-Served

remaining service requirement. Note that if all job sizes are the same, then SRPT is equivalent to First-Come-First-Served (FCFS), assuming FCFS tie-breaking.

SRPT has the biggest impact when job size variability is high, because it ensures that short jobs (or jobs with very little remaining work) do not get stuck waiting behind long jobs. However, even when the job size variability is not that high, the benefits of SRPT over policies that do not make use of size, like FCFS, are very clear, as shown in Fig. 8 where the job size distribution has squared coefficient of variation $C^2 = 10$. Recall from Sect. 6 that job size variability is often much higher, with C^2 in the thousands. The analysis of mean response time for the M/G/1/SRPT queue is well-known [125], as is mean response time for all the other policies shown in Fig. 8 [90].

One would imagine that SRPT would also be optimal in the multi-server setting. For the M/G/k, we define SRPT-k as the algorithm that at all times runs those k jobs with the smallest remaining processing time, preempting jobs in service as needed. For a multi-server system with no arrivals (all jobs present at the start), SRPT-k is optimal under a worst-case adversarial setting [81]. Once we add arrivals, however, surprisingly SRPT-k is no longer optimal in the worst case adversarial setting [104], and in fact SRPT-k can be arbitrarily far from optimal [104], [90, p.426]. For the (stochastic) M/G/k setting, in 2018 Grosof et al. proved that SRPT-k is optimal under heavy traffic [83]. A bound on its mean response time was also presented in [83], although that bound is quite loose unless load is very high. Unfortunately, datacenters typically operate under lighter traffic (see Sect. 7.4). Outside of heavy traffic, understanding when SRPT-k is optimal and analyzing its performance are both big *open problems.*

Sometimes the multiple servers are distributed, each server with its own queue, where there is a front-end dispatcher that routes each incoming job to a queue. As shown in Fig. 9, now there are *two* questions related to optimality: (i) How should we schedule jobs within the individual queues? (ii) What dispatching policy should be used? When the goal is to minimize mean response time, unsurprisingly the answer to question (i) is to use SRPT scheduling at each server. Regarding question (ii), Grosof

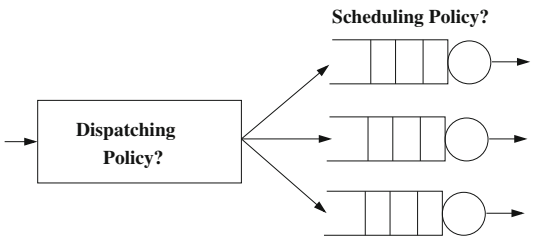

Fig. 9 Two decision points within a distributed load balancing system: (i) Pick the scheduling policy for the servers. (ii) Pick the dispatching policy

et al. [84] proves that a dispatching policy called Guardrails yields optimal mean response time in the heavy-traffic limit. Outside of heavy traffic, optimal dispatching is an *open problem*.

7.1.2 Sizes not known

More commonly, job sizes are *not* known a priori. What *is* known is the job size distribution, since one can observe jobs as they complete. One also knows the current *age* of a job, which is how much service the job has received so far. Let S be a random variable denoting a job's size. Then, one can imagine using the known age of the job, a, to estimate the *remaining size* of the job, given its age. One could then assign every job of age a a rank, $r(a)$, where the job's rank is its expected remaining size,

$$r(a) = \mathbf{E}\left[S - a \mid S > a\right].$$

It then makes sense to choose to always (preemptively) run the job with smallest rank, i.e., the job with Smallest Expected Remaining Processing Time (SERPT). While the SERPT policy sounds optimal for mean response time, because of its similarity to SRPT, it is missing a subtlety. Imagine that a job's remaining time has a Bimodal distribution, where, with probability half it is very small, but with probability half it is very large. While the expected remaining time of the job is large, it might nonetheless pay to run the job for a very short time, just in case it completes. This is the principle behind the Gittins Index policy which assigns to every job a rank, $r(a)$, based on its age a and its job size S, where

$$r(a) = \inf_{\Delta > 0} \frac{\mathbf{E}\left[\min\{S - a, \Delta\} \mid S > a\right]}{\mathbf{P}\left\{S - a \leq \Delta \mid S > a\right\}} \tag{2}$$

and then (preemptively) runs that job with lowest rank. The rank in (2) takes into account the expected remaining size of a job, given its age, but it also takes into account the probability that the job will complete in the next Δ time.

The Gittins Index policy is known to be optimal for minimizing mean response time in the M/G/1 queue, when job sizes are unknown, known, or partially known [7,8,78].[6] However, Gittins is a complex policy. In [130], Scully et al. proved that a much simpler policy, related to SERPT, is within a factor 5 of optimal. It is an *open problem* whether

[6] Gittins becomes SRPT when job sizes are known.

that factor can be improved, or whether other simple policies with near-optimal mean response time exist. The response time analysis of both the Gittins policy and the SERPT policy was only recently derived (2018), via the introduction of the SOAP framework, see [129], which produces a closed-form expression for response time for any policy which can be expressed via a rank function.[7]

For the M/G/k queue, one would imagine that the Gittins-k policy might be optimal, where Gittins-k at all times runs the k jobs with the lowest Gittins ranks. Very recently (2020), Scully et al. proved the first upper bound on the response time of Gittins-k and also proved that Gittins-k is heavy-traffic optimal, under very general job models [127,128]. Proving the optimality of Gittins-k outside of heavy traffic remains an *open problem*.

7.1.3 A related metric with more practical value: mean slowdown

The mean slowdown metric is related to mean response time, but is somewhat more practical. A job's slowdown is defined as its response time divided by its inherent size:

$$\text{Slowdown of job } j = \frac{\text{Response time of } j}{\text{Size of } j}. \tag{3}$$

When the goal is minimizing mean slowdown, it becomes important to give short response times to the short jobs, as their denominator in (3) is small. We save longer response times for the longer jobs, which are better able to absorb the longer response time over their larger denominators. Mean slowdown makes practical sense in that a person who is downloading a web page has much less tolerance for high response time than a person who is downloading a 2-h movie and does not mind a few minutes wait (while they go make popcorn).

The optimal algorithm for minimizing mean slowdown in the M/G/1 system is the RS algorithm [98]. Under RS, every job is assigned a rank which is equal to its current remaining size (R) multiplied by its original size (S). At all times, the RS algorithm preemptively runs that job with the lowest rank. The first analysis of the RS algorithm (both its mean response time and mean slowdown) is given in [129].

7.2 Response time tail: jobs with deadlines

We now describe one of the most popular performance metrics for the computing industry. Consider Facebook's customers, who are busy downloading Facebook pages. If the time to download a Facebook page is under 400 ms, then the time is not noticeable to a user; in particular, the user cannot differentiate between a 100 ms response time and a 200 ms response time. If the time exceeds 400 ms, the user will notice it, be irritated, and might eventually stop using Facebook. Thus it is in Facebook's interest to minimize the fraction of downloads that take more than 400 ms. If we think of a

[7] A job's "rank" is its priority, where lower rank is better, and where ties are broken in FCFS order. Rank is a function of age, but can also depend on a job's size or class [129].

stream of jobs arriving over time, we are trying to minimize

$$\mathbf{P}\{\text{Response time} > 400\,\text{ms}\}\ .$$

How should we do that?

This problem is equivalent to imagining that every job that arrives has a deadline of 400 ms. We would like to schedule the jobs so as to minimize the fraction that miss their deadlines. We can imagine, to start, that the job sizes are known a priori.

How to schedule jobs to minimize $\mathbf{P}\{\text{Response time} > 400\,\text{ms}\}$ is an *open problem*. One idea is to run Least Laxity First (LLF) [113], i.e., at all times, we would (preemptively) run that job which is closest to missing its deadline. However, one can imagine how this can become problematic, because the server starts working on one job, and then switches to working on a new arrival that might be closer to meeting the 400 ms deadline, and then switches again when another job arrives, potentially causing all the jobs (but the last one) to miss their deadlines because of all the sharing.

An alternative idea is an algorithm we will call Drop-If-Hopeless (DIH). Under DIH, we schedule all the jobs in FCFS order. When a job arrives, it looks at the queue ahead of it. If the job's size plus its waiting time in the queue exceeds 400 ms, then the job is dropped; otherwise the job enters the queue. Thus, the jobs in the queue are all guaranteed to have a response time within 400 ms. We believe that DIH was originally considered by [76]. While DIH makes a lot of sense, one can see that it too is not optimal. Imagine that a large job arrives and enters the queue, causing the work in the queue to now be close to 400 ms. This induces drops of many future arrivals because the work in the queue is too high. Was it really worth allowing that large job to enter?

We can imagine an improvement on DIH, which we will call Drop-The-Larger-Hopeless (DTLH). Under DTLH, we again schedule all jobs in FCFS order. When a job j arrives, if we see that the sum of j's size and its waiting time exceeds 400 ms, then rather than immediately dropping j, we first check if there's a job k in the queue which has larger size than j. If there is such a job k, then we drop k, rather than j. In this way, we are still keeping only jobs in the queue that will make the 400 ms goal, but we are biasing toward keeping smaller jobs, which should minimize the fraction of future drops.

While DTLH sounds good, it too is not optimal, and it is an *open problem* to determine how close to optimal it is. Understanding the dropping probability under DTLH is also an *open problem*, although the dropping probability for DIH is understood, under certain conditions; see [76].

When it comes to evaluating the tail of response time, $\mathbf{P}\{\text{Response Time} > t\}$, the standard technique is to numerically invert the Laplace transform of the tail function. Abate, Choudhury, and Whitt [12] present an overview of the classical numerical inversion techniques; see also [13,60,139]. More recently den Iseger and others [58,59] have invented different techniques to expand the class of functions that can be inverted as well as to improve the precision of the inversion. Very recently, Horváth et al. [96] found a way to invert the Laplace transform using concentrated matrix exponential (CME) distributions, further improving accuracy and stability in arithmetic calculations.

While there is plenty of numerical work, there are very few *analytical* (non-numerical) results on tails of response time. We do not have a closed-form expression for $\mathbf{P}\{\text{Response Time} > t\}$ even for the simplest queueing systems. Almost all the analytical research on tails of response time looks at the tail in the asymptotic limit, i.e., finding a function $f(t)$ such that

$$\lim_{t\to\infty} \frac{\mathbf{P}\{T > t\}}{f(t)} = 1.$$

The earliest work on asymptotic tail analysis dates back to Smith, [135], who investigates the tail of queueing times in a GI/G/1 FCFS system. Later work has looked at tail asymptotics in the context of different scheduling policies: Glynn and Whitt [79], Abate, Choudhury, and Whitt [10,11], and a beautiful series of papers including Borst, Boxma, Deng, Núñez-Queija and Zwart; see [46–49]. The work has also been extended to multi-server queues; see Foss and Korshunov [64]. Unfortunately, while comparing the asymptotic tail under different scheduling policies is interesting from a theoretical standpoint, it does not solve the practical question of scheduling to minimize $\mathbf{P}\{\text{Response Time} > t\}$ for a particular t; this remains an *open problem*.

7.3 The 99%-tile of response time

In datacenter work, both in industry and computer systems research, people like to talk about the 99th%-tile of response time; see, for example, Dean and Barroso [55], Berger et al. [37], Zhu et al. [162,163], Xu et al. [159]. The 99th%-tile might actually be the most commonly discussed metric in computer systems. When people talk about the 99th%-tile, they are intuitively imagining that they are studying the "almost worst-case" situation, or as high as the worst will get in practice, while at the same time not really thinking about an adversarial situation.

There are several ways that the 99th%-tile metric comes up in practice.

In the first way, there is a Service Level Objective (SLO), which is a response time goal that we don't want to exceed. This is akin to the 400 ms number in Sect. 7.2. Now we would like to find a way of scheduling jobs so as to ensure that 99% of the jobs have response time under this 400 ms SLO. That is, we want to schedule to ensure that:

$$\mathbf{P}\{\text{Response time} > 400\,\text{ms}\} < .01\ .$$

This is similar to what we saw in Sect. 7.2, except that instead of simply trying to minimize the fraction of jobs with response time > 400 ms, now we are requiring that no more than 1% have this behavior.

Sometimes the 99th%-tile SLO is combined with priority classes. In queueing when we hear "priority" we think that class 1 has absolute priority over class 2. In the compute industry, priority often comes in a different form, where class 1 jobs have a much tighter SLO than class 2. For example, the goal might be:

How should we schedule jobs to guarantee class 1 jobs a 99%-tile of response time of 400 ms, while guaranteeing class 2 jobs a 99%-tile of 4000 ms?

Sometimes the 99th%-tile SLO is phrased in terms of a capacity provisioning goal, for example

How many servers do we need in our datacenter, to ensure that at least 99% *of jobs complete within the* 400 *ms SLO?*

Note that this sounds a lot like a square-root staffing kind of rule [155], except that there's a 400 ms number which needs to factor into our answer somewhere.

Oftentimes, the 99th%-tile metric is phrased *without reference to an SLO at all*. Here, the goal is stated simply as

We want to schedule jobs so as to minimize the 99*th%-tile of response time.*

This latter phrasing is now quite different from Sect. 7.2.

While all of these phrasings involving the 99th%-tile are commonly spouted in industry, in our opinion, they are all a bit odd. In fact, after you think about them a little while, it becomes unclear whether the people spouting these metrics really want what they say they want! Consider for example this last metric of simply minimizing the 99th%-tile of response time. Observe that 1% of the jobs are never counted in this metric—they simply do not matter. So there is no point on even working on 1% of the jobs. Why not pick in advance which 1% we are not going to work on and just get rid of those? In fact, why not make that 1% be the very largest jobs, since they are the ones that contain most of the work. So here is an idea: we look at the job size distribution. We cut off the biggest 1% of the jobs. For the remaining 99% of jobs, we schedule these in FCFS order, to minimize the maximum response time of these remaining jobs.

When we suggest to companies that algorithms like the one above can achieve their objectives, they complain that such algorithms are not OK. "You can't simply drop all the big jobs!" So then we suggest simply moving that 1% set of big jobs permanently to the end of the line (always giving them lowest priority). This does not make companies happy either, because then they complain, "But some of those jobs might be important. You're being unfair to those jobs."

So, what do companies really want? They *do* care about having a low 99%-tile of response time, but they don't want to achieve a low 99%-tile at the expense of hurting "important" jobs. If the important jobs are correlated with the largest jobs, then a policy of the DIH type (which uses size only indirectly) would be preferable to a policy of the DTLH type (which directly penalizes larger jobs), which is still preferable to just dropping the top 1% of jobs. This is an issue that must be considered both for the tail formulation in this section and that in Sect. 7.2.

7.4 Power consumption

Power consumption is a huge problem in datacenters. Datacenters consume over 3% of the global electricity supply and account for over 2% of the total global greenhouse gas emissions [145].

Datacenter utilization is typically under 30% [30,136,140]. The reason for this is overprovisioning. One way this happens is that a user's job's resource needs might fluctuate over time: say the job needs 2 servers in parallel for its first phase and then 100 servers for its second phase and then back to two servers. In that case, the user will request 100 servers, which is over-provisioning much of the time. Another situation is that the user's job is providing a service, and the resource needs of the job depend on the arrival rate into that service, where the arrival rate fluctuates over time. The user will provision for the peak arrival rate into the service, which again leads to over-provisioning much of the time.

Having servers occasionally idling would not be such a problem except that datacenter servers use nearly as much power (65%) when they are sitting idle as they do when they are processing jobs, [71]. Thus, having servers sitting idle is very expensive.

Thus, the only way to reduce power is to dynamically turn servers off at times when they are not needed. Unfortunately, once we shut down a server, it becomes very hard to get it back up if it is suddenly needed again. Specifically, there is a huge *setup time* needed to turn servers on. A setup time can easily be 200 s, while desired response times are in the 400−500 ms range; thus, setup time can have a big effect on mean response time. To make things worse, the server is operating at full (100%) power during this setup time when it's unavailable.

This leads to a great many questions: When should one turn servers off? When should one turn them back on? How should one schedule jobs on the servers to minimize the need to turn servers off?

Power management raises a lot of *open problems*. The first thing one needs to understand is the effect of setup times in multiserver systems. The M/G/1 queue with setup times was first analyzed in 1964 by Welch [153]; however, the M/M/k system with setup time was not analyzed until 2013. In an M/M/k queue with setup, it is assumed that any server that is not in use will immediately shut down (to save power). Every arriving job picks an off server, if one exists, and puts it into setup mode; the job then joins the queue. In 2014, Gandhi et al. [68] derived the Laplace transform of response time for an M/M/k queue with setup, assuming exponentially distributed setup times. Unfortunately, the formula for an M/M/k/setup system is complicated and can only be cleanly expressed for small k. It would be really nice to have an *approximate* formula akin to the beautiful decomposition result that exists for the M/G/1/setup [153] and the M/M/∞/setup [70]. In particular, it turns out that the setup time matters less and less for larger systems, i.e., the M/M/k/setup system eventually starts to look just like an M/M/k queue without setup, as k gets high. It would be very nice to have a simple approximation for the M/M/k/setup as a function of k. Finally, the M/G/k system with setup time is also a wide *open problem*.

Ideally, what we really want is to be able to analyze a k server system with time-varying load where idle servers are shut off during low load periods, but need to go through setup when load goes back up. We could then better understand when it pays to turn servers off and on. In Gandhi et al. [71], the authors describe a datacenter operation with time-varying load and a front-end dispatcher which both handles the routing of jobs to servers and also controls which servers are on and off. (They provide a realistic prototype of a Facebook datacenter.) They propose a two-pronged policy for dealing with time-varying load. First, their policy delays turning servers off; when

the server idles, it needs to stay idle for some designated amount of time before it is shut off. This is called a "delayed-off." This alone is insufficient, though. The problem is that typical front-end dispatchers aim to balance load among servers, which means that a server which becomes idle will not stay idle for long. To ensure that servers that become idle will stay idle long enough to turn off, the policy defines an ordering on the servers, wherein the dispatcher always tries to pack jobs into the lowest-numbered servers first. This allows the high-numbered servers to become idle and then not receive further arrivals, allowing them to turn off. The packing policy is referred to as "load UNbalancing," because it does the opposite of load balancing, which spreads out jobs among all servers. In [69], the authors provide a very preliminary attempt at analyzing policies of this nature. There is room for much more work in this space on the analytical front.

8 Conclusion

The goal of this paper was to examine new queueing models, workloads, and metrics, all inspired by datacenter computing today.

On the modeling front, we saw that the typical job in datacenters is a parallel "multiserver" job which occupies multiple servers for some period of time. We also saw that jobs are often flexible in the number of servers on which they can run, and being allocated more servers does not typically translate into a proportional speed increase. This sublinear speedup behavior makes allocating servers across jobs complex. We also saw other common forms of parallel jobs, including the DAG job, made up of independent tasks with precedence constraints, inspired by serverless computing, as well as the limited fork–join job, inspired by the popular MapReduce framework.

On the computing workloads front, we studied jobs at Google's datacenters. We saw that per-job compute usage (expressed as a product of number of servers and time) can be extremely variable, with squared coefficients of variation in the tens of thousands. The distribution of per-job compute usage is Pareto distributed with a surprisingly strong heavy-tailed property: the top 1% of jobs are true resource hogs, comprising 99% of the total load.

On the metrics front, we examined performance metrics of interest in cloud computing today. We saw that even relatively simple metrics, like mean response time, are much harder to optimize when job sizes are unknown or only partially known and multiple servers are involved. We studied the slowdown metric, response time tail metrics, and the counter-intuitive but popular 99%-tile SLO. We also studied energy metrics, which often involve combining response time tails, capacity needs, and power usage.

Each new model, workload, and metric has suggested new open problems for queueing theorists, of practical importance to cloud computing today.

Acknowledgements We would like to thank Sem Borst, Onno Boxma, and Isaac Grosof for their helpful suggestions and careful proof-reading.

Funding Funding was provided by National Science Foundation (Grant numbers CMMI-1938909, CSR-1763701, XPS-1629444) and Google (Grant number 2020 Faculty Research Award).

References

1. Amazon EC2. http://aws.amazon.com/ec2/. Accessed 15 Nov 2020
2. Azure Public Dataset (2019). https://github.com/Azure/AzurePublicDataset. Accessed 15 Nov 2020
3. Google Compute Engine. http://cloud.google.com/products/compute-engine.html. Accessed 15 Nov 2020
4. Windows Azure. http://www.windowsazure.com/. Accessed 15 Nov 2020
5. Datacenter Spending (2020). https://www.cbronline.com/news/data-centre-spending. Accessed 15 Nov 2020
6. Flexera.: State of the Cloud Report (2020). https://www.flexera.com/blog/industry-trends/trend-of-cloud-computing-2020/. Accessed 15 Nov 2020
7. Aalto, S., Ayesta, U., Righter, R.: On the Gittins index in the M/G/1 queue. Queueing Syst. **63**(1), 437–458 (2009)
8. Aalto, S., Ayesta, U., Righter, R.: Properties of the Gittins index with application to optimal scheduling. Probab. Eng. Inf. Sci. **25**(3), 269–288 (2011)
9. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X..: Tensorflow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), pp. 265–283 (2016)
10. Abate, J., Choudhury, G.L., Whitt, W.: Asymptotics for steady-state tail probabilities in structured Markov queueing models. Stoch. Mod. **10**(1), 99–143 (1994)
11. Abate, J., Choudhury, G.L., Whitt, W.: Waiting-time tail probabilities in queues with long-tail service-time distributions. Queueing Syst. **16**, 311–338 (1994)
12. Abate, J., Choudhury, G.L., Whitt, W.: An introduction to numerical transform inversion and its application to probability models. In: Grassmann, W.K. (ed.) Computational Probability, pp. 257–323. Springer, Boston (2000)
13. Abate, J., Whitt, W.: A unified framework for numerically inverting Laplace transforms. INFORMS J. Comput. **18**(4), 408–421 (2006)
14. Acar, U., Blelloch, G.E., Blumofe, R.: The data locality of work stealing. Theory Comput. Syst. **35**(3), 321–347 (2002)
15. Afanaseva, L., Bashtova, E., Grishunina, S.: Stability analysis of a multi-server model with simultaneous service and a regenerative input flow. Methodol. Comput. Appl. Probab. **22**, 1439–1455 (2020)
16. Afanaseva, L., Grishunina, S.: Stability conditions for a multiserver queueing system with a regenerative input flow and simultaneous service of a customer by a random number of servers. Queueing Syst. **94**, 213–241 (2020)
17. Agrawal, K., Li, J., Lu, K., Moseley, B.: Scheduling parallel DAG jobs online to minimize average flow time. In: Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '16), pp. 176–189 (2016)
18. Agrawal, K., Li, J., Lu, K., Moseley, B.: Scheduling parallelizable jobs online to minimize the maximum flow time. In: Symposium on Parallel Algorithms and Architectures (SPAA'16), pp. 195–205 (2016)
19. Agrawal, K., Li, J., Lu, K., Moseley, B.: Scheduling parallelizable jobs online to maximize throughput. In: LATIN 2018: Theoretical Informatics—13th Latin American Symposium, Buenos Aires, Argentina, pp. 755–776 (2018)
20. Ahmad, N., Greenberg, A.G., Lahiri, P., Maltz, D., Patel, P.K., Sengupta, S., Vaid, K.V.: Distributed load balancer. Google Patents. U.S. Patent App. 12/189,438 (2008)
21. Anton, E., Ayesta, U., Jonckheere, M., Verloop, I.M..: On the stability of redundancy models (2019). arXiv:1903.04414
22. Anton, E., Ayesta, U., Jonckheere, M., Verloop, I.M..: Improving the performance of heterogeneous data centers through redundancy (2020). arXiv:2003.01394
23. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: 10th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 119–129 (1998)
24. Arthurs, E., Kaufman, J.: Sizing a message store subject to blocking criteria. In: IFIP Performance Conference, pp. 547–564 (1979)

25. AWS. Netflix & AWS Lambda Case Study. https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/. Accessed 15 Nov 2020
26. AWS. Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html. Accessed 15 Nov 2020
27. Baccelli, F., Foss, S.: Poisson hail on a hot ground. J. Appl. Probab. **48**(A), 343–366 (2011)
28. Baccelli, F., Makowski, A.M.: Simple computable bounds for the fork–join queue. Technical Report RR-0394, INRIA (1985)
29. Baccelli, F., Makowski, A.M., Shwartz, A.: The fork–join queue and related systems with synchronization constraints: stochastic ordering and computable bounds. Adv. Appl. Probab. **21**, 629–660 (1989)
30. Barroso, L.A., Holzle, U.: The case for energy-proportional computing. Computer **40**(12), 33–37 (2007)
31. Bean, N.G., Gibbens, R.J., Zachary, S.: Asymptotic analysis of single resource loss systems in heavy traffic, with applications to integrated networks. Adv. Appl. Probab. **27**(1), 273–292 (1995)
32. Bekker, R., Borst, S., Núñez-Queija, R.: Performance of TCP-friendly streaming sessions in the presence of heavy-tailed elastic flows. Perform. Eval. **61**(2), 143–162 (2005)
33. Benameur, N., Fredj, S. Ben, Delcoigne, F., Oueslati-Boulahia, S., Roberts, J.W.: Integrated admission control for streaming and elastic traffic. In: International Workshop on Quality of Future Internet Services, pp. 69–81 (2001)
34. Berg, B., Dorsman, J.-P., Harchol-Balter, M.: Towards optimality in parallel job scheduling. Proc. ACM Meas. Anal. Comput. Syst. (POMACS/SIGMETRICS) **1**(2), 1–30 (2017). Article 40
35. Berg, B., Harchol-Balter, M., Moseley, B., Wang, W., Whitehouse, J.: Optimal resource allocation for elastic and inelastic jobs. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'20), pp. 75–87, Philadelphia, PA (2020)
36. Berg, B., Vesilo, R., Harchol-Balter, M.: heSRPT: Parallel scheduling to minimize mean slowdown. In: 38th International Symposium on Computer Performance, Modeling, Measurement, and Evaluation (IFIP PERFORMANCE 2020), Milan, Italy (2020)
37. Berger, D., Berg, B., Zhu, T., Sen, S., Harchol-Balter, M.: Robinhood: Tail latency aware caching—dynamic reallocation from cache-rich to cache-poor. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 195–212, Carlsbad, CA (2018)
38. Bienia, C., Kumar, S., Singh, J. P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08), pp. 72–81, New York, NY (2008)
39. Blelloch, G., Gibbons, P., Matias, Y.: Provably efficient scheduling for languages with fine-grained parallelism. J. ACM **46**(2), 281–321 (1999)
40. Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Simhadri, H.V.: Scheduling irregular parallel computations on hierarchical caches. In: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11), pp. 355–366, San Jose, California (2011)
41. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
42. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: IEEE Symposium on Foundations of Computer Science, pp. 356–368 (1994)
43. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
44. Blumofe, R.D., Papadopoulos, D.: Hood: a user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin (1999)
45. Bonald, T., Proutière, A.: On performance bounds for the integration of elastic and adaptive streaming flows. In: Joint International ACM SIGMETRICS/Performance Conference on Measurement and Modeling of Computer Systems, pp. 235–245 (2004)
46. Borst, S., Núñez-Queija, R., Zwart, B.: Sojourn time asymptotics in processor-sharing queues. Queueing Syst. **53**(1–2), 31–51 (2006)
47. Borst, S.C., Boxma, O.J., Núñez-Queija, R., Zwart, B.: The impact of the service discipline on delay asymptotics. Perform. Eval. **54**(2), 175–206 (2003)
48. Boxma, O.J., Deng, Q., Zwart, B.: Waiting-time asymptotics for the M/G/2 queue with heterogeneous servers. Queueing Syst. **40**(1), 5–31 (2002)
49. Boxma, O.J., Zwart, B.: Tails in scheduling. SIGMETRICS Perform. Eval. Rev. **34**(4), 13–20 (2007)

50. Brill, P.H., Green, L.: Queues in which customers receive simultaneous service from a random number of servers: a system point approach. Manag. Sci. **30**(1), 51–68 (1984)
51. Cera, M.C., Georgiou, Y., Richard, O., Maillard, N., Navaux, P.O.A.: Supporting malleability in parallel architectures with dynamic CPUSETsMapping and dynamic MPI. In: Kant, K., Pemmaraju, S.V., Sivalingam, K.M., Wu, J. (eds.) International Conference on Distributed Computing and Networking (ICDCN'20), pp. 242–257 (2010)
52. Chowdhury, R.A., Ramachandran, V., Silvestri, F., Blakeley, B.: Oblivious algorithms for multicores and networks of processors. J. Parallel Distrib. Comput. **73**(7), 911–925 (2018)
53. Crovella, M., Harchol-Balter, M., Murta, C.: Task assignment in a distributed system: Improving performance by unbalancing load. In: Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp. 268–269. Poster Session (1998)
54. Dasylva, A., Srikant, R.: Bounds on the performance of admission control and routing policies for general topology networks with multiple call centers. In: Eighteenth Annual IEEE INFOCOM'99 International Conference on Computer Communications, pp. 505–512 (1999)
55. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
56. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
57. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and QoS-aware cluster management. In: ASPLOS'14, pp. 127–144, Salt Lake City, Utah (2014)
58. den Iseger, P.: Numerical transform inversion using Gaussian quadrature. Probab. Eng. Inf. Sci. **20**, 1–44 (2006)
59. den Iseger, P., Gruntjes, P., Mandjes, M.: A Wiener–Hopf based approach to numerical computations in fluctuation theory for Lévy processes. Math. Methods Oper. Res. **78**(1), 101–118 (2013)
60. Dubner, H., Abate, J.: Numerical inversion of Laplace transforms by relating them to the finite Fourier cosine transform. J. ACM **15**(1), 115–123 (1968)
61. Fan, Z., Sen, R., Koutris, P., Albarghouthi, A.: Automated tuning of query degree of parallelism via machine learning. In: Proceedings of the 3rd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (2020)
62. Filippopoulos, D., Karatza, H.: An M/M/2 parallel system model with pure space sharing among rigid jobs. Math. Comput. Model. **45**(5), 491–530 (2007)
63. Foss, S., Konstantopoulos, T., Mountford, T.: Power law condition for stability of Poisson hail. J. Theor. Probab. **31**, 684–704 (2018)
64. Foss, S., Korshunov, D.: Heavy tails in multi-server queue. Queueing Syst. Theory Pract. **52**, 31–48 (2006)
65. Foss, S., Korshunov, D., Zachary, S.: An Introduction to Heavy-Tailed and Subexponential Distributions, 2nd edn. Springer, New York (2013)
66. Fouladi, S., Wahby, R.S., Shacklett, B., Balasubramaniam, K.V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., Winstein, K.: Encoding, fast and slow: low-latency video processing using thousands of tiny threads. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 363–376, Boston, MA (2017)
67. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: ACM PLDI, pp. 212–223 (1998)
68. Gandhi, A., Doroudi, S., Harchol-Balter, M., Scheller-Wolf, A.: Exact analysis of the M/M/k/setup class of Markov chains via Recursive Renewal Reward. Queueing Syst. Theory Appl. **77**(2), 177–209 (2014)
69. Gandhi, A., Gupta, V., Harchol-Balter, M., Kozuch, M.: Optimality analysis of energy-peformance trade-off for server farm management. Perform. Eval. **67**(11), 1155–1171 (2010)
70. Gandhi, A., Harchol-Balter, M., Adan, I.: Server farms with setup costs. Perform. Eval. **67**(11), 1123–1138 (2010)
71. Gandhi, A., Harchol-Balter, M., Raghunathan, R., Kozuch, M.: AutoScale: dynamic, robust capacity management for multi-tier data centers. ACM Trans. Comput. Syst. **30**(4), 1–26 (2012)
72. Gardner, K., Harchol-Balter, M., Scheller-Wolf, A., Van Houdt, B.: A better model for job redundancy: decoupling server slowdown and job size. ACM/IEEE Trans. Netw. **25**(6), 3353–3367 (2017)
73. Gardner, K., Harchol-Balter, M., Scheller-Wolf, A., Velednitsky, M., Zbarsky, S.: Redundancy-d: the power of d choices for redundancy. Oper. Res. **65**(4), 1078–1094 (2017)
74. Gardner, K., Zbarsky, S., Doroudi, S., Harchol-Balter, M., Hyytia, E., Scheller-Wolf, A.: Queueing with redundant requests: exact analysis. Queueing Syst. Theory Appl. **83**(3), 227–259 (2016)

75. Gardner, K., Zbarsky, S., Doroudi, S., Harchol-Balter, M., Hyytiä, E., Scheller-Wolf, A.: Reducing latency via redundant requests: exact analysis. In: ACM Sigmetrics 2015 Conference on Measurement and Modeling of Computer Systems, pp. 347–360 (2015)
76. Gavish, B., Schweitzer, P.J.: The Markovian queue with bounded waiting time. Manag. Sci. **23**(12), 1349–1357 (1977)
77. Ghaderi, J.: Randomized algorithms for scheduling VMs in the cloud. In: 35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10–14, 2016, pp. 1–9 (2016)
78. Gittins, J.C., Glazebrook, K.D., Weber, R.: Multi-armed Bandit Allocation Indices. Wiley, New York (2011)
79. Glynn, P.W., Whitt, W.: Logarithmic asymptotics for steady-state tail probabilities in a single-server queue. J. Appl. Probab. **31**(A), 131–156 (1994)
80. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. **37**(1), 5–20 (1996)
81. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Optimization and approximation in deterministic squencing and scheduling: a survey. Ann. Discrete Math. **5**, 287–326 (1979)
82. Grosof, I, Harchol-Balter, M, Scheller-Wolf, A.: Stability for two-class multiserver-job systems (2020). arXiv:2010.00631
83. Grosof, I., Scully, Z., Harchol-Balter, M.: SRPT for multiserver systems. Perform. Eval. **127–128**, 154–175 (2018)
84. Grosof, I., Scully, Z., Harchol-Balter, M.: Load balancing guardrails: keeping your heavy traffic on the road to low response times. Proc. ACM Meas. Anal. Comput. Syst. (POMACS/SIGMETRICS) **3**(2), 1–31 (2019). Article 42
85. Guo, M., Guan, Q., Ke, W.: Optimal scheduling of VMs in queueing cloud computing systems with a heterogeneous workload. IEEE Access **6**, 15178–15191 (2018)
86. Gupta, A., Acun, B., Sarood, O., Kale, L.: Towards realizing the potential of malleable jobs. In: IEEE International Conference on High Performance Computing (HiPC'14) (2014)
87. Harchol-Balter, M.: Network analysis without exponentiality assumptions. Ph.D. thesis, University of California at Berkeley (1996)
88. Harchol-Balter, M.: The effect of heavy-tailed job size distributions on computer system design. In: Proceedings of ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics, Engineering and Statistics, Washington, DC (1999)
89. Harchol-Balter, M.: Task assignment with unknown duration. J. ACM **49**(2), 260–288 (2002)
90. Harchol-Balter, M.: Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, Cambridge (2013)
91. Harchol-Balter, M., Crovella, M., Murta, C.: On choosing a task assignment policy for a distributed server system. In: Lecture Notes in Computer Science, No. 1469: 10th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, pp. 231–242 (1998)
92. Harchol-Balter, M., Downey, A.: Exploiting process lifetime distributions for dynamic load balancing. In: Proceedings of ACM SIGMETRICS, pp. 13–24, Philadelphia, PA (1996)
93. Harchol-Balter, M., Downey, A.: Exploiting process lifetime distributions for dynamic load balancing. ACM Trans. Comput. Syst. **15**(3), 253–285 (1997)
94. Harchol-Balter, M., Schroeder, B., Bansal, N., Agrawal, M.: Size-based scheduling to improve web performance. ACM Trans. Comput. Syst. **21**(2), 207–233 (2003)
95. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**, 33–38 (2008)
96. Horvath, G., Horvath, I., Almousa, S.A.-D., Telek, M.: Numerical inverse Laplace transformation using concentrated matrix exponential distributions. Perform. Eval. **137**, 1–22 (2019)
97. Hunt, P.J., Kurtz, T.G.: Large loss networks. Stoch. Process. Appl. **53**(2), 363–378 (1994)
98. Hyytiä, E., Aalto, S., Penttinen, A.: Minimizing slowdown in heterogeneous size-aware dispatching systems. In: Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (2012)
99. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 445–451, New York, NY (2017)
100. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.J., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud programming simplified: a Berkeley view on serverless computing (2019). CoRR, arXiv:1902.03383

101. Joshi, G., Soljanin, E., Wornell, G.: Efficient replication of queued tasks for latency reduction in cloud systems. In: Allerton Conference on Communication, Control, and Computing, University of Illinois, Urbana-Champaign (2015)
102. Kim, S.S.L M/M/s queueing system where customers demand multiple server use. Ph.D. thesis, Southern Methodist University (1979)
103. Lee, K., Shah, N.B., Huang, L., Ramchandran, K.: The MDS queue: analysing the latency performance of erasure codes. IEEE Trans. Inf. Theory **63**(5), 2822–2842 (2017)
104. Leonardi, S., Raz, D.: Approximating total flow time on parallel machines. In: Proceedings of the Annual ACM Symposium on Theory of Computing (STOC), pp. 110–119 (1997)
105. Li, H., Groep, D., Wolters, L.: Workload characteristics of a multicluster supercomputer. In: 10th International Conference on Job Scheduling Strategies for Parallel Processing (IPPS'04), pp. 176–193. Springer (2004)
106. Lin, S.-H., Paolieri, M., Chou, C.F., Golubchik, L.: A model-based approach to streamlining distributed training for asynchronous SGD. In: MASCOTS 2018, pp. 306–318 (2018)
107. Lu, Y., Xie, Q., Kliot, G., Geller, A., Larus, J.R., Greenberg, A.: Join-idle-queue: a novel load balancing algorithm for dynamically scalable web services. Perform. Eval. **68**(11), 1056–1071 (2011)
108. Madni, S.H.H., Latiff, M.S.A., Abdullahi, M., Abdulhamid, S.M., Usman, M.J.: Performance comparison of heuristic algorithms for task scheduling in IaaS cloud computing environment. PLoS ONE **12**(5), 1–26 (2017)
109. Maguluri, S.T., Srikant, R.: Scheduling jobs with unknown duration in clouds. IEEE/ACM Trans. Netw. **22**(6), 1938–1951 (2014)
110. Maguluri, S.T., Srikant, R., Ying, L.: Stochastic models of load balancing and scheduling in cloud computing clusters. In: Proceedings of IEEE INFOCOM, pp. 702–710 (2012)
111. Massoulie, L., Roberts, J.W.: Bandwidth sharing and admission control for elastic traffic. Telecommun. Syst. **15**, 185–201 (2000)
112. Melikov, A.: Computation and optimization methods for multiresource queues. Cybern. Syst. Anal. **32**(6), 821–836 (1996)
113. Mok, A.: Fundamental design problems of distributed systems for the hard real-time environment. Ph.D. thesis, MIT, Department of EE and CS (1983)
114. Morozov, E., Rumyantsev, A.S.: Stability analysis of a MAP/M/s cluster model by matrix-analytic method. In: Fiems, D., Paolieri, M., Platis, A.N. (eds.) Computer Performance Engineering—13th European Workshop, EPEW 2016, Chios, Greece, October 5–7, 2016, Proceedings, volume 9951 of Lecture Notes in Computer Science, pp. 63–76. Springer (2016)
115. Narlikar, G.J.: Scheduling threads for low space requirement and good locality. Theory Comput. Syst. **35**(2), 151–187 (2002)
116. Nelson, R.D., Tantawi, A.N.: Approximate analysis of fork/join synchronization in parallel queues. IEEE Trans. Comput. **37**(6), 739–743 (1988)
117. Ponomarenko, L., Kim, C.S., Melikov, A.: Performance Analysis and Optimization of Multi-traffic on Communication Networks. Springer, Berlin (2010)
118. Psychas, K., Ghaderi, J.: On non-preemptive VM scheduling in the cloud. Proc. ACM Meas. Anal. Comput. Syst. **1**(2), 1–29 (2017). Article 35
119. Raaijmakers, Y., Borst, S., Boxma, O.: Delta probing policies for redundancy. Perform. Eval. **127**(128), 21–35 (2018)
120. Raaijmakers, Y., Borst, S., Boxma, O.: Redundancy scheduling with scaled Bernoulli service requirements. Queueing Syst. **93**(1–2), 67–82 (2019)
121. Raaijmakers, Y., Borst, S., Boxma, O.: Stability of redundancy systems with processor sharing. In: Proceedings of the 13th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'20), pp. 120–127 (2020)
122. Rizk, A., Poloczek, F., Ciucu, F.: Stochastic bounds in fork–join queueing systems under full and partial mapping. Queueing Syst. **83**(3), 261–291 (2016)
123. Rumyantsev, A., Morozov, E.: Stability criterion of a multiserver model with simultaneous service. Ann. Oper. Res. **252**(1), 29–39 (2017)
124. Schrage, L.E.: A proof of the optimality of the shortest remaining processing time discipline. Oper. Res. **16**, 678–690 (1968)
125. Schrage, L.E., Miller, L.W.: The queue M/G/1 with the shortest remaining processing time discipline. Oper. Res. **14**, 670–684 (1966)

126. Schroeder, B., Harchol-Balter, M.: Evaluation of task assignment policies for supercomputing servers: the case for load unbalancing and fairness. Clust. Comput. J. Netw. Softw. Tools Appl. **7**(2), 151–161 (2004)
127. Scully, Z., Grosof, I., Harchol-Balter, M.: The Gittins policy is nearly optimal in the M/G/k under extremely general conditions. Proc. ACM Meas. Anal. Comput. Syst. (POMACS/SIGMETRICS) **3**(4), 1–29 (2020). Article 43
128. Scully, Z., Grosof, I., Harchol-Balter, M.: Optimal multiserver scheduling with unknown job sizes in heavy traffic. In: 38th International Symposium on Computer Performance, Modeling, Measurement, and Evaluation (IFIP PERFORMANCE 2020), Milan, Italy (2020)
129. Scully, Z., Harchol-Balter, M., Scheller-Wolf, A.: SOAP: one clean analysis of all age-based scheduling policies. Proc. ACM Meas. Anal. Comput. Syst. (POMACS/SIGMETRICS) **2**(1), 1–30 (2018). Article 16
130. Scully, Z., Harchol-Balter, M., Scheller-Wolf, A.: Simple near-optimal scheduling for the M/G/1. Proc. ACM Meas. Anal. Comput. Syst. (POMACS/SIGMETRICS) **4**(1), 1–29 (2020). Article 11
131. Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., Ragan-Kelley, J.: Numpywren: serverless linear algebra (2018). CoRR, arXiv:1810.09679
132. Shneer, S., Stolyar, A..: Large-scale parallel server system with multi-component jobs (2020). arXiv:2006.11256
133. Sigman, K.: Appendix: a primer on heavy-tailed distributions. Queueing Syst. **33**(1/3), 261–275 (1999)
134. Simhadri, H.V., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A.: Experimental analysis of space-bounded schedulers. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14), pp. 30–41, Prague, Czech Republic (2014)
135. Smith, W.L.: On the distribution of queueing times. Math. Proc. Camb. Philos. Soc. **49**(3), 449–461 (1953)
136. Snyder, B.: Server virtualization has stalled, despite the hype (2010). InfoWorld. https://www.infoworld.com/article/2624771/server-virtualization-has-stalled--despite-the-hype.html. Accessed 15 Nov 2020
137. Sreekanti, V., Chenggang, W., Lin, X.C., Schleier-Smith, J., Gonzalez, J., Hellerstein, J.M., Tumanov, A.: Cloudburst: stateful functions-as-a-service. Proc. VLDB Endow. **13**(11), 2438–2452 (2020)
138. Sun, Y., Zheng, Z., Koksal, C.E., Kim, K.-H., Shroff, N.B.: Provably delay efficient data retrieving in storage clouds. In: Proceedings of IEEE INFOCOM (2015)
139. Talbot, A.: The accurate numerical inversion of Laplace transforms. IMA J. Appl. Math. **23**(1), 97–120 (1979)
140. Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., Zhang, P.: Twine: a unified cluster management system for shared infrastructure. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20) (2020)
141. Thomasian, A.: Analysis of fork/join and related queueing systems. ACM Comput. Surv. **47**(2), 1–71 (2014)
142. Tian, H., Zheng, Y., Wang, W.: Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud. In: 10th ACM Symposium on Cloud Computing (SoCC'19), Santa Cruz, CA (2019)
143. Tikhonenko, O.M.: Generalized Erlang problem for service systems with finite total capacity. Probl. Inf. Transm. **41**(3), 243–253 (2005)
144. Tirmazi, M., Barker, A., Deng, N., Haque, M.E., Qin, Z.G., Hand, S., Harchol-Balter, M., Wilkes, J.: Borg: the next generation. In: Proceedings of the 15th European Conference on Computer Systems (EuroSys'20), pp. 1–14, Greece (2020)
145. Trueman, C.: Why data centres are the new frontier in the fight against climate change. Computerworld (2019)
146. Van Dijk, N.M.: Blocking of finite source inputs which require simultaneous servers with general think and holding times. Oper. Res. Lett. **8**(1), 45–52 (1989)
147. Vandevoorde, M.T., Roberts, E.S.: WorkCrews: an abstraction for controlling parallelism. Int. J. Parallel Program. **17**(4), 347–366 (1988)

148. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Proceedings of the 10th European Conference on Computer Systems, p. 18 (2015)
149. Wang, D., Joshi, G., Wornell, G.W.: Efficient straggler replication in large-scale parallel computing. Proc. ACM Meas. Model. Comput. Syst. (ACM SIGMETRICS 2019) **4**(2), 1–23 (2019). Article 7
150. Wang, W., Harchol-Balter, M., Jiang, H., Scheller-Wolf, A., Srikant, R.: Delay asymptotics and bounds for multi-task parallel jobs. Queueing Syst. Theory Appl. **91**(3), 207–239 (2019)
151. Wang, W., Xie, Q., Harchol-Balter, M.: Zero queueing for multi-server jobs (2020). arXiv:2011.10521
152. Wardley, S.: Why the fuss about serverless? (2016). https://blog.gardeviance.org/2016/11/why-fuss-about-serverless.html. Accessed 15 Nov 2020
153. Welch, P.D.: On a generalized M/G/1 queueing process in which the first customer of each busy period receives exceptional service. Oper. Res. **12**, 736–752 (1964)
154. Weng, W., Wang, W.: Dispatching parallel jobs to achieve zero queueing delay (2020). arXiv:2004.02081
155. Whitt, W.: Understanding the efficiency of multi-server service systems. Manag. Sci. **38**(5), 708–723 (1992)
156. Whitt, W.: Blocking when service is required from several facilities simultaneously. AT&T Bell Lab. Tech. J. **64**, 1807–1856 (1985)
157. Wilkes, J.: More Google cluster data. Google research blog (2011). http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html. Accessed 15 Nov 2020
158. Wilkes, J.: Google cluster-usage traces v3 (2019). http://github.com/google/cluster-data. Accessed 15 Nov 2020
159. Xu, Y., Musgrave, Z., Noble, B., Bailey, M.: Bobtail: avoiding long tails in the cloud. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13), pp. 329–342, USA (2013)
160. Zhan, X., Bao, Y., Bienia, C., Li, K.: PARSEC3.0: a multicore benchmark suite with network stacks and SPLASH-2X. ACM SIGARCH Comput. Arch. News **44**, 1–16 (2017)
161. Zhang, W., Fang, V., Panda, A., Shenker, S.: Kappa: A programming framework for serverless computing. In: ACM Symposium on Cloud Computing (SoCC'20), pp. 328–343 (2020)
162. Zhu, T., Berger, D., Harchol-Balter, M.: SNC-Meister: admitting more tenants with tail latency SLOs. In: ACM Symposium on Cloud Computing (SoCC'16), pp. 374–387, Santa Clara, CA (2016)
163. Zhu, T., Tumanov, A., Kozuch, M.A.. Harchol-Balter, M., Ganger, G.R.: PriorityMeister: tail latency QoS for shared networked storage. In: ACM Symposium on Cloud Computing 2014 (SoCC'14), pp. 1–14, Seattle, WA (2014)