heSRPT: Optimal Scheduling of Parallel Jobs with Known Sizes

Benjamin Berg
Carnegie Mellon University

Rein Vesilo
Macquarie University

Mor Harchol-Balter
Carnegie Mellon University

ABSTRACT

Nearly all modern data centers serve workloads which are capable of exploiting parallelism. When a job parallelizes across multiple servers it will complete more quickly, but jobs receive diminishing returns from being allocated additional servers. Because allocating multiple servers to a single job is inefficient, it is unclear how best to share a fixed number of servers between many parallelizable jobs. In this paper, we provide the first closed form expression for the optimal allocation of servers to jobs. Specifically, we specify the number of servers that should be allocated to each job at every moment in time. Our solution is a combination of favoring small jobs (as in SRPT scheduling) while still ensuring high system efficiency. We call our scheduling policy high-efficiency SRPT (heSRPT).

1. INTRODUCTION

In this paper we consider a typical scenario where a data center composed of N servers is tasked with completing a set of M parallelizable jobs, where typically M is much smaller than N. In our scenario, each job has a different inherent size (service requirement) which is known up front to the system. In addition, each job can be run on any number of servers at any moment in time. These assumptions are reasonable for many parallelizable workloads such as training neural networks using TensorFlow [1, 8]. Our objective is to allocate servers to jobs so as to minimize the *mean flow time* across all jobs, where the flow time of a job is the time until the job leaves the system. What makes this problem difficult is that jobs receive a concave, sublinear speedup from parallelization – jobs have a decreasing marginal benefit from being allocated additional servers (see Figure 1). Hence, in choosing a job to receive each additional server, one must keep the overall efficiency of the system in mind. In this paper, we will derive the optimal policy for allocating servers to jobs when all jobs follow a realistic sublinear speedup function.

The optimal allocation policy will depend heavily on the how parallelizable the jobs are. To see this, first consider the case where jobs are *embarrassingly parallel*. In this case, we observe that the entire data center can be viewed as a *single server* that can be perfectly utilized by or shared between jobs. Hence, from the single server scheduling literature, it is known that the Shortest Remaining Processing Time policy (SRPT) will minimize the mean flow time across jobs. By contrast, consider the case where jobs are hardly parallelizable and a single job receives very little benefit from additional servers. In this case, the optimal policy is to divide the system equally between jobs, a policy called EQUI. When jobs are neither embarrassingly parallel nor completely sequential, the optimal policy with respect to mean flow time must split the difference between EQUI and SRPT, favoring short jobs while still respecting the overall efficiency of the system.

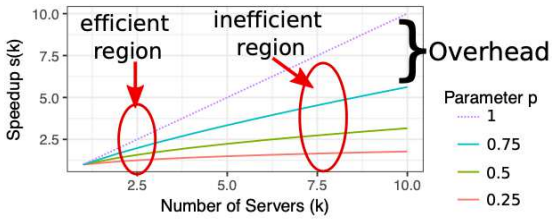

Figure 1: A variety of speedup functions of the form $s(k) = k^p$, shown with varying values of p. When $p = 1$ we say that jobs are *embarrassingly parallel*, and hence we consider cases where $0 < p < 1$. Note that all functions in this family are concave and lie below the embarrassingly parallel speedup function ($p = 1$).

Our Model

Our model assumes that all M jobs are present at time $t = 0$. Job i is assumed to have some inherent size x_i where, WLOG,

$$x_1 \geq x_2 \geq \ldots \geq x_M.$$

In general we will assume that all jobs follow the *same* speedup function, $s : \mathbb{R}^+ \to \mathbb{R}^+$, which is of the form

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

for some $0 < p < 1$. If a job i of size x_i is allocated k servers for its entire lifetime, it will complete at time

$$\frac{x_i}{s(k)}.$$

In general, the number of servers allocated to a job can change over the course of the job's lifetime. It therefore helps to think of $s(k)$ as a *rate*[1] of service where the remaining size of job i after running on k servers for a length of time t is

$$x_i - t \cdot s(k).$$

[1]WLOG we assume the service rate of a single server to be 1. More generally, we could assume the rate of each server to be μ, which would simply replace $s(k)$ by $s(k)\mu$ in every formula

MAMA 2019 Phoenix, AZ, USA

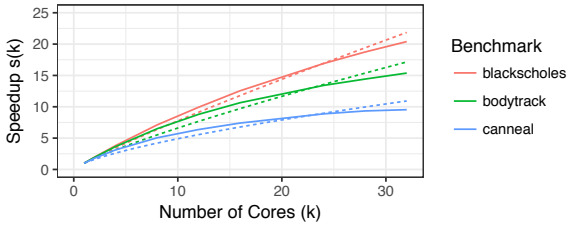

Figure 2: Various speedup functions of the form $s(k) = k^p$ (dotted lines) which have been fit to real speedup curves (solid lines) measured from jobs in the PARSEC-3 parallel benchmarks [9]. The three jobs, blackscholes, bodytrack, and canneal, are best fit by the functions where $p = .89$, $p = .82$, and $p = .69$ respectively.

We choose the family of functions $s(k) = k^p$ because they sublinear and concave and can be fit to a variety of empirically measured speedup functions (see Figure 2). Similarly, [6] assumes $s(k) = k^p$ where $p = 0.5$.

In general, we assume that there is some policy, P, which allocates servers to jobs at every time, t. When describing the state of the system, we will use $m^P(t)$ to denote the number of remaining jobs in the system at time t, and $x_i^P(t)$ to denote the remaining size of job i at time t. We also denote the completion time of job i under policy P as T_i^P. When the policy P is implied, we will drop the superscript.

We will assume that the number of servers allocated to a job need not be discrete. In general, we will think of the N servers as a *single, continuously divisible resource*. Hence, the policy P can be defined by an *allocation function* $\boldsymbol{\theta}^P(t)$ where

$$\boldsymbol{\theta}^P(t) = (\theta_1^P(t), \theta_2^P(t), \ldots, \theta_M^P(t)).$$

Here, $0 \leq \theta_i^P(t) \leq 1$ for each job i, and $\sum_{i=1}^M \theta_i^P(t) \leq 1$. An allocation of $\theta_i^P(t)$ denotes that under policy P, at time t, job i receives a speedup of $s(\theta_i^P(t) \cdot N)$. We will always assume that completed jobs receive an allocation of 0 servers.

We denote the optimal allocation function which minimizes mean flow time as $\boldsymbol{\theta}^*(t)$. Similarly, $m^*(t)$, $x_i^*(t)$, and T_i^* denote the corresponding quantities under the optimal policy.

Why Server Allocation is Counter-intuitive

Consider a simple system with $N = 10$ servers and $M = 2$ identical jobs of size 1, where $s(k) = k^{.5}$, and where we wish to minimize mean flow time. One intuitive argument would be that, since everything in this system is symmetric, the optimal policy would allocate half the servers to job one and half the servers to job two. Interestingly, while this does minimize the *makespan* of the jobs, it does *not* minimize their flow time. Alternately, a queueing theorist might look at the same problem and say that to minimize flow time, we should use the SRPT policy, allocating all servers to job one and then all servers to job two. However, this causes the system to be very inefficient. We will show that the optimal policy in this case is to allocate 75% of the servers to job one and 25% of the servers to job two. In our simple, symmetric system, the optimal allocation is very asymmetric! Note that this asymmetry is *not* an artifact of the form of the speedup function used. If we had instead assumed that s was Amdahl's Law [6] with a parallelizable fraction of $f = .9$, the optimal split is to allocate 63.5% of the system to one of the jobs. If we imagine a set of M arbitrarily sized jobs, one suspects that the optimal policy favors shorter jobs, but calculating the exact allocations for this policy is not trivial.

Why Finding the Optimal Policy is Hard

At first glance, solving for the optimal policy seems amenable to classical optimization techniques. However, naive application of these techniques would require solving $M!$ optimization problems (corresponding to each of the possible completion orders of the jobs), each consisting of $O(M^2)$ variables (corresponding to each job's allocation between consecutive departures) and $O(M)$ constraints (which enforce the completion order being considered). Furthermore, although these techniques could produce the optimal policy for a single problem instance, it is unlikely that they would yield a closed form solution. We instead advocate for finding a closed form solution for the optimal policy, which allows us to build intuition about the underlying dynamics of the system.

2. PRIOR WORK

Despite the prevalence of parallelizable data center workloads, it is not known, in general, how to optimally allocate servers across a set of parallelizable jobs. There has been extensive work from the theoretical computer science community [7, 4, 5, 2] regarding how to schedule parallelizable jobs with speedup functions in order to minimize mean flow time. However, this work has concentrated on competitive analysis rather than direct optimization. Recently, the performance modeling community has considered this problem [3], but this work only considers jobs with unknown, exponentially distributed sizes. We therefore present the first closed form analysis of the optimal policy when parallelizable jobs have known sizes.

3. MINIMIZING MEAN FLOW TIME

The purpose of this section is to determine the optimal allocation function $\boldsymbol{\theta}^*(t)$ which defines the allocation for each job at any moment in time, t, and minimizes mean flow time. As previously noted, the space of potential allocation policies is too large to be amenable to direct optimization. Our approach is therefore to reduce this search space by proving a series of properties of the optimal allocation function, $\boldsymbol{\theta}^*(t)$ (Theorems 1,2,3,4). These properties will allow us to derive the optimal allocation function in Theorem 5.

Overview of Our Results

We begin by showing that the optimal allocation does not change between job departures, and hence it will suffice to consider the value of the allocation function only at times just after a job departure occurs. This is stated formally as Theorem 1.

Theorem 1. Consider any two times t_1 and t_2 where, WLOG, $t_1 < t_2$. Let $m^*(t)$ denote the number of jobs in the system at time t under the optimal policy. If $m^*(t_1) = m^*(t_2)$ then

$$\boldsymbol{\theta}^*(t_1) = \boldsymbol{\theta}^*(t_2).$$

Using just this very mild characterization of the optimal policy, we can show that the optimal policy completes jobs in Shortest-Job-First (SJF) order. This is stated in Theorem 2.

Theorem 2 (Optimal Completion Order)**.** The optimal policy completes jobs in the order

$$M, M-1, M-2, \ldots, 1$$

and hence jobs are completed in the Shortest-Job-First (SJF) order.

Since jobs are completed in SJF order, we can conclude that, at time t, the jobs left in the system are specifically jobs $1, 2, \ldots, m(t)$.

Besides the completion order, the other key property of the optimal allocation that we will exploit is the *scale-free property*. The scale-free property states that for any job, i, job i's allocation relative to jobs completed after job i (i.e. the jobs larger than job i) remains constant throughout job i's lifetime. The scale-free property is stated formally in Theorem 3.

Theorem 3 (Scale-free Property)**.** Let t be a time when there are exactly i jobs in the system and hence $m(t) = i$. Consider any t' such that $t' < t$. Then,

$$\frac{\theta_i^*(t')}{\sum_{j=1}^{i} \theta_j^*(t')} = \theta_i^*(t).$$

The scale-free property reveals the optimal substructure of the optimal policy – knowing the correct allocation to each job when there are $M-1$ jobs in the system reduces the problem of finding the correct allocation given M jobs to a single variable optimization problem.

The final property needed to reduce our search space is the size-invariant property, given in Theorem 4.

Theorem 4 (Size-invariant Property)**.** Consider any two sets of jobs, A and B, each containing $m(t)$ jobs at time t. If $\boldsymbol{\theta}_A^*(t)$ and $\boldsymbol{\theta}_B^*(t)$ are the optimal allocations to the jobs in sets A and B at time t, respectively, we have that

$$\boldsymbol{\theta}_A^*(t) = \boldsymbol{\theta}_B^*(t).$$

That is, the allocation function depends only on the number of unfinished jobs in the system, not their remaining sizes.

This is a counter-intuitive result because one might imagine that the optimal allocation should be different if two very differently sized jobs are in the system instead of two equally sized jobs.

Finally Theorem 5 provides the optimal allocation function which minimizes mean flow time.

Theorem 5 (Optimal Allocation Function)**.** At time t, when $m(t)$ jobs remain in the system,

$$\theta_i^*(t) = \begin{cases} \left(\frac{i}{m(t)}\right)^{\frac{1}{1-p}} - \left(\frac{i-1}{m(t)}\right)^{\frac{1}{1-p}} & 1 \le i \le m(t) \\ 0 & i > m(t) \end{cases}$$

Note that the optimal allocation to job i is independent of its remaining size, in accordance with Theorem 4. Given the optimal allocation function $\boldsymbol{\theta}^*(t)$, we can also explicitly compute the optimal mean flow time. This is stated in Theorem 6.

Theorem 6 (Optimal Mean Flow Time)**.** Given a set of M jobs of size $x_1 > x_2 > \ldots > x_M$, the mean flow time, T^*, under the optimal allocation policy $\boldsymbol{\theta}^*(t)$ is given by

$$T^* = \frac{1}{s(N)M} \sum_{k=1}^{M} x_k \cdot \left[ks(1+\omega_k^*) - (k-1)s(\omega_k^*)\right]$$

where

$$\omega_k^* = \frac{1}{\left(\frac{k}{k-1}\right)^{\frac{1}{1-p}} - 1} \qquad \forall 1 < k \le M$$

and

$$\omega_1^* = 0$$

4. CONCLUSION

The optimal allocation policy derived in this paper biases towards short jobs, but does not give strict priority to these jobs in order to maintain the overall efficiency of the system. That is, as stated in Theorem 5,

$$0 < \boldsymbol{\theta}_1^*(t) < \boldsymbol{\theta}_2^*(t) < \ldots < \boldsymbol{\theta}_{m(t)}^*(t).$$

We refer to the optimal policy as *high efficiency Shortest-Remaining-Processing-Time* or *heSRPT*.

Acknowledgments

The first and third authors were supported by: NSF-CSR-1763701, NSF-XPS-1629444, and a Microsoft Faculty Award 2018.

5. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] K. Agrawal, J. Li, K. Lu, and B. Moseley. Scheduling parallelizable jobs online to minimize the maximum flow time. SPAA '16, pages 195–205. ACM, 2016.

[3] B. Berg, J.P. Dorsman, and M. Harchol-Balter. Towards optimality in parallel scheduling. *ACM POMACS (SIGMETRICS)*, 1(2):40:1 – 40:30, 2018.

[4] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235:109–141, 1999.

[5] J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. SODA '09, pages 685–692. ACM, 2009.

[6] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, 2008.

[7] Sungjin Im, Benjamin Moseley, Kirk Pruhs, and Eric Torng. Competitively scheduling tasks with intermediate parallelizability. *ACM Transactions on Parallel Computing (TOPC)*, 3(1):4, 2016.

[8] Sung-Han Lin, Marco Paolieri, Cheng-Fu Chou, and Leana Golubchik. A model-based approach to streamlining distributed training for asynchronous sgd. In *MASCOTS 2018*, pages 306–318. IEEE, 2018.

[9] X. Zhan, Y. Bao, C. Bienia, and K. Li. PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News*, 44:1–16, 2017.