Optimal Scheduling of Parallel Jobs with Unknown Service Requirements

Benjamin Berg
Carnegie Mellon University, USA

Mor Harchol-Balter
Carnegie Mellon University, USA

ABSTRACT

Large data centers composed of many servers provide the opportunity to improve performance by parallelizing jobs. However, effectively exploiting parallelism is non-trivial. Specifically, for each arriving job one must decide its level of parallelization - the number of servers on which the job is run. Our goal is to determine the optimal allocation of servers to jobs so as to minimize the mean response time across all arriving jobs, where the response time of a job is the time from when the job arrives until it completes. In choosing the optimal level of parallelization, we must consider the following tradeoff. On the one hand, parallelizing an individual job across multiple servers reduces the response time of that individual job. On the other hand, jobs receive diminishing returns from being allocated additional servers, so allocating too many servers to a single job can lead to low system efficiency. Hence, while a higher level of parallelization may decrease an individual job's response time, it may have a deleterious effect on overall mean response time. We consider the case where the remaining sizes of jobs are unknown to the system at every moment in time. We prove that, if all jobs follow the same speedup function, the optimal policy is EQUI, which divides servers equally among jobs, maximizing system efficiency. When jobs may follow different speedup functions, EQUI is no longer optimal and we provide an alternate policy, GREEDY*, which performs within 1% of optimal in simulation. This chapter is based on content first presented in [3].

Keywords: Word One, Word Two, Word Three

INTRODUCTION

The Parallelization Tradeoff

Modern data centers are composed of a large number of servers, affording programmers the opportunity to run jobs faster by parallelizing across many servers. To exploit this opportunity, jobs are often designed to run on any number of servers [5]. Running on additional servers may reduce a job's response time, the time from when the job arrives to the system until it is completed. However, effectively exploiting parallelism is non-trivial. Specifically, one must decide how many servers to allocate to each job in the system at every moment in time. We consider the setting where jobs arrive over time, and the system must choose each job's level of parallelization in order to minimize the mean response time across jobs. In choosing each job's server allocation, one must consider the following tradeoff. Parallelizing an individual job across multiple servers reduces the response time of that individual job. In practice, however, each job receives a diminishing marginal benefit from being allocated additional servers. Hence, allocating too many servers to a single job may decrease overall system efficiency. While a larger server allocation may decrease an individual job's response time, the net effect may be an increase in the overall mean response time across jobs. We therefore aim to design a system which balances this tradeoff, choosing each job's server allocation in order to minimize the mean response time across all jobs. It was shown in [4] 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 service rate on k servers. If we normalize $s(1)$ to be 1, we see that a job will complete s(k) times faster on k servers than on a single server. In general, it is conceivable that every job will have a different speedup function. However, there are also many workloads [4] where all jobs have the same speedup function, such as when one runs many instances of the same program. It turns out that even allocating servers to jobs which follow a single speedup function is non-trivial. Hence, we will first focus on jobs following a single speedup function before turning our attention to multiple speedup functions. In addition to differing in their speedup functions, different jobs may represent different amounts of computation that must be performed. For example, if a data center is processing search queries, a simple query might require much less processing than a complex query. We refer to the amount of inherent work associated with a job as the job's size. It is common in data centers that job sizes are unknown to the system { users are not required to tell the system anything about the internals of their jobs when they submit them. Hence, we consider the case where the system does not know, at any moment in time, the remaining sizes of the jobs currently in the system.

The question of how to best allocate servers to jobs is commonly referred to as the choice between fine grained parallelism, where every job is parallelized across a large number of servers, and coarse grained parallelism, where the server allocation of each job is kept small. This same tradeoff arises in many parallel systems beyond data centers. For example, [3] considers the case of jobs running on a multicore chip. In this case, running on additional cores allows an individual job to complete more quickly, but leads to an inefficient use of resources which could increase the response times of subsequent jobs. Additionally, an operating system might need to choose how to partition memory (cache space) between multiple applications. Likewise, a computer architect may have to choose between fewer, wider bus lanes for memory access, or several narrower bus lanes. In all cases, one must balance the effect on an individual job's response time with the effect on overall mean response time. Throughout this chapter, we will use the terminology of parallelizing jobs across servers, however all of our remarks can be applied equally to any setting where limited resources must be shared amongst concurrently running processes.

Our model

We assume that jobs arrive into an n server system according to a Poisson Process with rate Λ jobs/second where

$$\Lambda := \lambda n \quad (1)$$

for some λ. The random variable X denotes the size of a job, which represents the amount of inherent work associated with a job. We are interested in the common case where the remaining size of each job is unknown to the system at every moment in time. To model this, we assume that each job size, X, is drawn i.i.d. from an exponential distribution with rate μ (which may be unknown to the system). Due to the memoryless property of the exponential distribution, the remaining size of any job in the system is always distributed exponentially with rate μ, regardless of how long the job has been running. Hence, while the system may learn the mean job size over time, the system can never infer that one job has a smaller remaining size than another job. Any job can be run on any subset of the n servers, and running on additional servers increases the service rate that a job receives. Specifically, we define a speedup function, $s : \mathbb{R}_+ \to \mathbb{R}_+$, such that a job running on k servers receives a service rate of s(k) units of work per second. Hence, a job of size X which is parallelized across k servers would complete in X_k seconds, where

$$X_k = \frac{\mathrm{X}}{\mathrm{s(k)}}$$

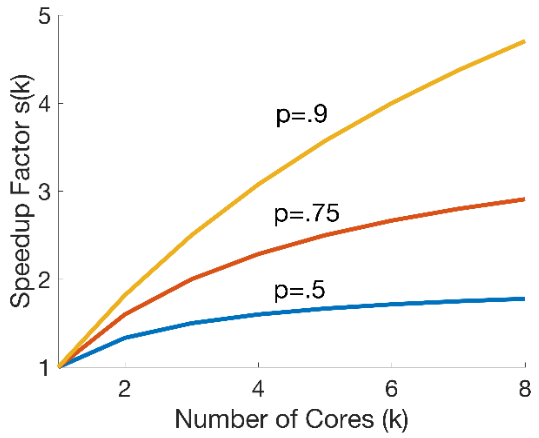
In general, the analysis in this chapter will not make use of the specific form of the speedup function. We will, however, make some mild technical assumptions about the speedup function that reflect how parallelizable jobs behave in practice. First, we normalize the service rate that a job receives on a single server to be one unit of work per second, and therefore s(1) = 1. We will assume that the speedup function, $s : \mathbb{R}_+ \to \mathbb{R}_+$, is non-decreasing and concave, in agreement with functions described in [15]. Note that, because servers can be shared between jobs in practice, the speedup function is defined for non-integer server allocations. Finally, we will focus on instances where jobs receive an imperfect speedup and thus s(k) is assumed to be sublinear: $s(k) < k$ for all $k > 1$ and there exists some constant $c > 0$ such that $s(k) < c$ for all $k \geq 0$. When a job runs on less than 1 server, we assume that there is no overhead due to parallelism, and hence the job receives a linear speedup up to 1 server. That is,

$$s(k) = k, \qquad \forall\, 0 \leq k \leq 1$$

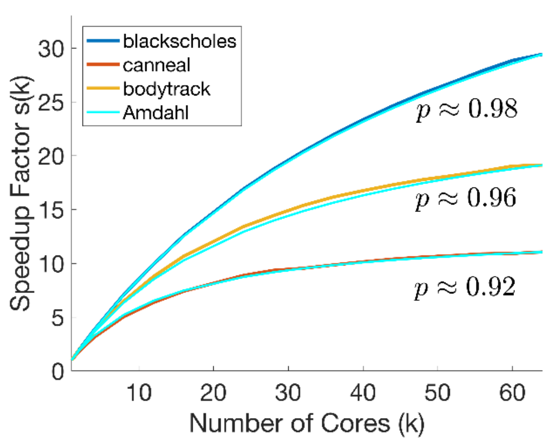
An example of a well-known speedup function which obeys the above conditions is Amdahl's law [10], which models every job as having a fraction of work, p, which is parallelizable. The speedup factor $s(k)$ is then a function of the parameter p as follows:

$$s(k) = \frac{1}{\frac{p}{k} + 1 - p}, \qquad \forall\, 0 \leq k \leq 1$$

Figure 1 shows Amdahl's law under various values of p. Although Amdahl's law ignores aspects of job behavior, we also see in Figure 1 that several workloads from the PARSEC-3 benchmark [27] follow speedup curves which can be accurately modeled by Amdahl's law. Hence, although our analysis will not rely on the specifics of the speedup function, we will use Amdahl's law in numerical examples. An allocation policy defines, at every moment in time, how many servers are allocated to each job which is currently in the system. Servers are assumed to be identical, and thus any job is capable of running on any server. An allocation policy can therefore allocate any number of servers to any job in the system as long as the total number of servers allocated to all jobs does not exceed n. Furthermore, jobs are assumed to be malleable, meaning a job can change the number of servers it runs on over time. Hence, an allocation policy is free to change the number of servers allocated to each job over time. We will first consider the case where jobs are homogeneous with respect to s - all jobs receive the same speedup due to parallelization. We will then consider the case of heterogeneous jobs which may follow different speedup functions.



(a) Amdahl's Law



(b) PARSEC-3

Figure 1. Various speedup curves under (a) Amdahl's law and (b) the PARSEC-3 Benchmark. We see that the PARSEC-3 speedup curves are accurately approximated by Amdahl's law; these approximations are shown in (b) in light blue.

The Problem

The fact that modern parallel jobs are capable of running on any number of servers begs the question of how many servers should be allocated to each job. Let T denote the response time of a job (the time between when the job arrives and when all of its pieces have completed). Our goal is to find and analyze scheduling policies that minimize the mean response time, $\mathbb{E}[T]$, across all jobs. Clearly, $\mathbb{E}[T]$ depends on the allocation policy, the speedup function, s, the arrival rate of jobs into the system, Λ, the mean job size, $\mathbb{E}[X]$, and the number of servers, n. We will denote the mean response time under a particular allocation policy, P, as

$$\mathbb{E}[T]^P.$$

We define the average system load, ρ, to be

$$\rho := \frac{\Lambda \mathbb{E}[X]}{n}$$

This is equivalently the fraction of time a server is busy when each job is run on a single server. Thus, $\rho < 1$ is a necessary condition for stability, regardless of the allocation policy used. We assume both ρ and $s(\cdot)$ are constant over time. By Little's law, minimizing the mean response time across jobs is equivalent to minimizing the mean number of jobs in the system. Hence, we will frequently consider the number of jobs in the system, N, with the goal of minimizing the mean number of jobs in the system, $\mathbb{E}[N]$. We denote the mean number of jobs in the system under a particular allocation policy, P, to be

$$\mathbb{E}[N]^P.$$

The EQUI Policy

EQUI is a policy which first appeared in [9]. Under EQUI, at all times, the n servers are equally divided among the jobs in the system. Specifically, whenever there are l jobs in the system, each job is parallelized across n/l servers. In Section 3, we prove that EQUI is optimal with respect to mean response time when jobs are homogeneous with respect to speedup (see Theorem 1).

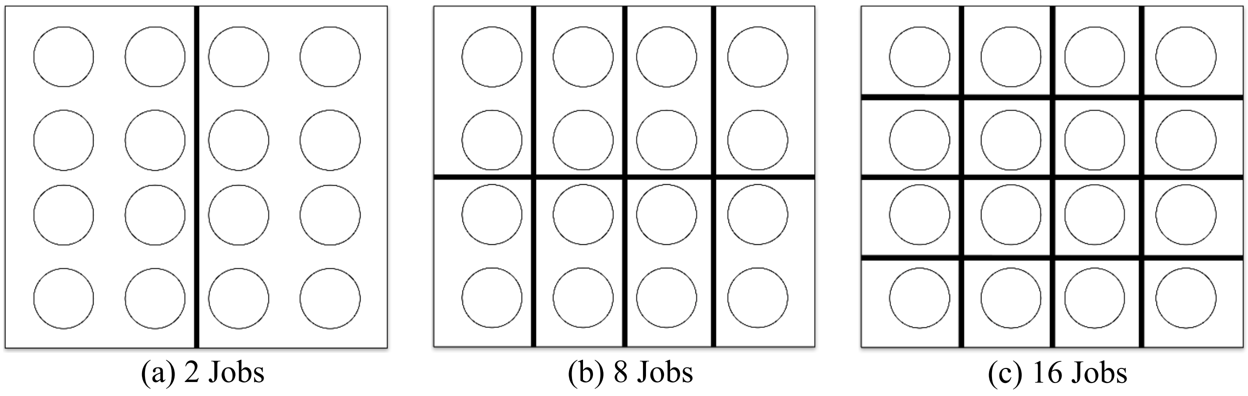


Figure 2. A view of the EQUI policy with various numbers of jobs in the system. At every moment in time, EQUI divides the n = 16 servers equally among the jobs currently in the system.

The GREEDY* Policy

We consider the case where jobs follow different speedup functions in Section 4. We show that, when jobs are permitted to have different speedup functions, EQUI is no longer optimal. We also show how to numerically compute the optimal policy, OPT, when jobs follow multiple speedup functions. Since finding OPT is computationally intensive we introduce a simple class of policies, called GREEDY, that performs well by maximizing the departure rate. In Section 4.4, we prove that one policy in this class, GREEDY*, dominates by both maximizing the overall departure rate and deferring parallelizable work (see Theorem 2). When jobs follow multiple speedup functions, GREEDY* achieves a mean response time within 1% of OPT in wide range of simulations.

PRIOR WORK

Prior work on exploiting parallelism has traditionally been split between several disparate communities. The SIGMETRICS/Performance community frequently considers scheduling a stream of incoming jobs for execution across several servers with the goal of minimizing mean response time, e.g., [11, 16, 19, 25, 28]. Job sizes are assumed to be random variables drawn from some general distribution, and arrivals are assumed to occur according to some stochastic arrival process. Typically, it is assumed that each individual job is run on just one server: jobs are not parallelizable. An exception is the work on Fork-Join queues. Here it is assumed that every job is parallelized across all servers [24]; the question of finding the optimal level of parallelization is not addressed. This community has also considered systems where the service rate that a job receives can be adjusted over time, but this work has focused on balancing a tradeoff between response time and some other variable such as power consumption [2] or Goodness of Service [21]. By contrast, the SPAA/STOC/FOCS community extensively studies the effect of parallelism, however it largely focuses on a single job. More recently, this community has considered the effect of parallelization on the mean response time of a stream of jobs. However, this has been through the lens of competitive analysis, which assumes the job sizes (service times), arrival times, and even speedup curves are adversarially chosen, e.g. [8, 9, 12]. Competitive analysis also does not yield closed form expressions for mean response time. We seek to analyze this problem using a more typical queueing theoretic model. Our workloads are drawn from distributions and our analysis yields closed-form expressions for mean response time as well as expressions for the optimal level of parallelization. The advent of moldable jobs which can run on any number of cores has pushed the high performance computing (HPC) systems community to consider how to effectively allocate cores to jobs [13, 22, 23, 26]. These studies tend to be empirical rather than analytical, each looking at specific workloads and architectures, often leading to conflicting results between studies. By contrast, this chapter considers this problem from a stochastic, analytical point of view, deriving optimal scheduling algorithms in a more general model.

EQUI: AN OPTIMAL POLICY

In order to effectively parallelize jobs, it makes sense to consider policies where the level of parallelization of each job depends on the state of the system. This may sound discouraging, since systems with state dependent service rates are in general hard to analyze. However, we will show that a very simple policy, EQUI, is both analytically tractable and optimal.
EQUI [9] is a generalization of Processor Sharing [17] to systems with multiple servers. Under EQUI, whenever there are l jobs in the system, each job runs on n/l servers. This corresponds to a service rate of $s(n/l)$ for each job when there are l jobs in the system. Recall that job sizes are exponentially distributed with rate μ. Hence, when there are l jobs in the system, the total rate at which jobs complete is $l\mu s(n/l)$ which is equal to nμ when $l \geq n$. Because job sizes are exponentially distributed and arrivals occur according to a Poisson process, we can represent a system under the EQUI policy via a continuous time Markov chain. Figure 3 shows the Markov chain for EQUI, where state i corresponds to having i jobs in the system. Note that this Markov chain is 1-Dimensionally infinite and is repeating when there are more than n jobs in the system. Hence, the mean number of jobs under EQUI can be analyzed via standard techniques [14, 17]. It is not immediately clear, however, that EQUI is optimal. Intuitively, EQUI appears to use the smallest degree of parallelism possible for each job without idling servers. While this prevents one from giving a highly inefficient allocation to any job, it seems possible that some policy could improve upon EQUI by exploiting parallelism more aggressively. In Section 3.1 we prove that EQUI is indeed optimal with respect to mean response time when all jobs follow the same speedup function and are exponentially distributed. We then explain the intuition behind this policy in Section 3.2.

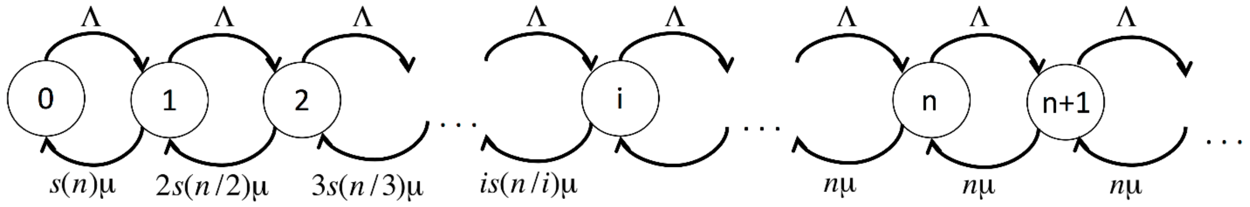


Figure 3. Markov chain representing the total number of jobs under EQUI.

Proving that EQUI is Optimal

To prove that EQUI is optimal, we first prove a helpful lemma, Lemma 1.

Lemma 1 *For any concave, sublinear function, s, the function $i \cdot s\left(\frac{n}{i}\right)$ is increasing in i for all $i < n$, and is non-decreasing in i for all $i \geq n$.*

Proof To see that $i \cdot s\left(\frac{n}{i}\right)$ is increasing in i when $i < n$, we can consider the following difference for any $\delta > 0$:

$$i \cdot (1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - i \cdot s\left(\frac{n}{i}\right) = i\left((1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - s\left(\frac{n}{i}\right)\right).$$

Since $\frac{n}{i} > 1$, s increases sublinearly, and $(1+\delta)s\left(\frac{n}{i\cdot(1+\delta)}\right) > s\left(\frac{n}{i}\right)$. Thus

$$i \cdot (1+\delta)s\left(\frac{n}{i \cdot (1+\delta)}\right) - i \cdot s\left(\frac{n}{i}\right) > 0$$

and $\boldsymbol{i \cdot s\left(\frac{n}{i}\right)}$ is increasing in i.
For any $\boldsymbol{i \geq n}$, we have assumed $\boldsymbol{s\left(\frac{n}{i}\right) = \frac{n}{i}}$. Thus,

$$i \cdot s\left(\frac{n}{i}\right) = n, \qquad \forall i \geq n,$$

which is non-decreasing in $\boldsymbol{i}$.
We can now prove the optimality of EQUI.
Theorem 1 Given malleable jobs with exponentially distributed job sizes which all follow the same concave, sublinear speedup function, s,

$$\mathbb{E}[T]^{EQUI} \leq \mathbb{E}[T]^{P}$$

for any allocation policy P.
Proof Let P be an allocation policy which processes malleable jobs and currently has i active jobs (the system is in state $\boldsymbol{i}$). In every state, $\boldsymbol{i}$, P must decide (i) how many jobs, j, to run and (ii) how to allocate the n cores amongst the j jobs. Let $\theta = (\theta_1, \theta_2, \ldots, \theta_j)$ denote the fraction of the n servers allocated to each of the j jobs that run while the system is in state i. For example, job 1 receives an allocation of $n\theta_1$ servers and runs at rate $s(n\theta_1)$ while the system is in state i. Given that each job departs at rate μ when run on a single server, the total rate of departures from state i under P is

$$\mu \sum_{k=1}^{j} s(n\,\theta_k) \quad (2)$$

where $0 < j \leq i, \theta_k > 0$ for all $1 \leq k \leq j$, and $\sum_{k=1}^{j} \theta_k = 1$. We also know that

$$\frac{1}{j}\sum_{k=1}^{j} s(n\,\theta_k) \leq s\left(\frac{n}{j}\right)$$

again, by the concavity of s. Hence,

$$\sum_{k=1}^{j} s\,(n\,\theta_k) \leq j \cdot s\left(\frac{n}{j}\right), \qquad \forall\, 0 < j \leq i, \forall\, \theta \in (0,1]^j$$

and thus an upper bound on P's total rate of departures from any state is of the form

$$j \cdot s\left(\frac{n}{j}\right)\mu \quad (3)$$

By Lemma 1, $j \cdot s\left(\frac{n}{j}\right)$ is non-decreasing in j. Thus, an upper bound on P's rate of departures from any state $\boldsymbol{i}$ is

$$i \cdot s\left(\frac{n}{i}\right)\mu$$

Furthermore, we can see that EQUI achieves this departure rate in every state, i. Hence, (3) is the maximal rate of departures from any state, $\boldsymbol{i}$. We can now compare the policy P with EQUI. Clearly, the arrival rate in any state i is the same under both policies. If we consider each state $\boldsymbol{i}$ where P has a strictly lower rate of departures than EQUI, we could reduce the rate of departures under EQUI by idling some servers in order to match the rate of departures under P. We call the resulting policy with idle servers P'. Because P' has the same rate of departures and arrivals as P in every state, it is clear that

$$\mathbb{E}[N]^{P'} = \mathbb{E}[N]^{P}$$

And hence, by Little's Law,

$$\mathbb{E}[T]^{P'} = \mathbb{E}[T]^{P}.$$

However, P' was obtained from EQUI by simply idling servers in some states. It is easy to see that any allocation of these idle servers does not increase mean response time. Hence, we have that

$$\mathbb{E}[T]^{P'} = \mathbb{E}[T]^{P} \geq \mathbb{E}[T]^{EQUI}$$

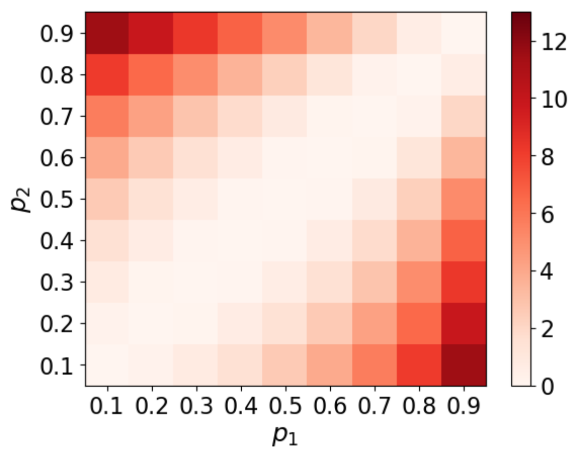
as desired.

What is EQUI doing

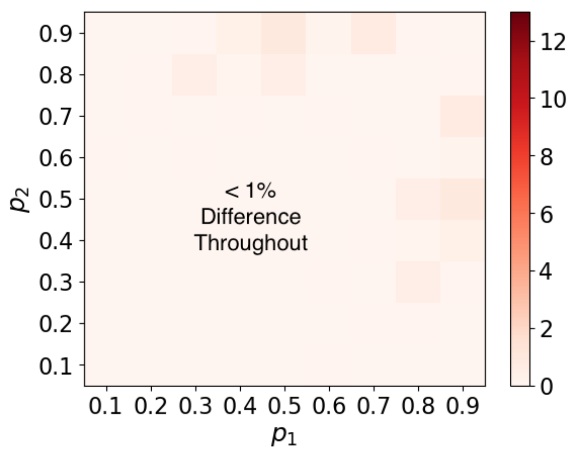
The above proof provides nice intuition about why EQUI is a good policy. In particular, we proved that in each state, EQUI maximizes the total rate of departures of active jobs. Following a similar argument, it is easy to see that in any state $\boldsymbol{i}$, EQUI is also maximizing the total service rate of jobs in the system given that there are i jobs. That is, EQUI maximizes the total number of units of work per second completed across all i jobs. Hence, we say that EQUI maximizes system efficiency in every state $\boldsymbol{i}$. It is not obvious that maximizing system efficiency should lead to the optimal policy. For example, consider the case where jobs have known sizes and the system is given a job of size 1 and a job of size 100. The optimal policy in this case can reduce mean response time by favoring the shorter job, and EQUI in general will not be optimal. In the case of known job sizes, the optimal policy is willing to sacrifice system efficiency in order to reduce the time until the next completion.

In our setting with unknown job sizes, however, EQUI also minimizes the expected time until the next completion. Because all job sizes are drawn i.i.d. from the same exponential distribution, the allocation policy does not have the ability to favor jobs which it believes are shorter. More formally, for any allocation of servers to jobs which results in a total service rate of z, the expected time until the next completion is $\boldsymbol{1/z}$. Hence, maximizing the total service rate in state i also minimizes the expected time until the next completion. Any policy which favors one job over another is increasing the expected time until the next completion and decreasing system efficiency, which could also have been accomplished by starting with an equal allocation and then idling some servers.

MULTIPLE SPEEDUP FUNCTIONS



(a) EQUI vs. OPT



(b) GREEDY* vs. OPT

Figure 3. Heat maps showing the percentage difference in mean response time, $\mathbb{E}[T]$, between (a) EQUI and OPT and between (b) GREEDY and OPT, in the case of two speedup functions where s_1 and s_2 are Amdahl's law with parameters p_1 and p_2 respectively. Here $\mathbb{E}[X] = \frac{1}{2}$ and $\Lambda_1 = \Lambda_2 = 5$. The axes represent different values of p_1 and p_2. GREEDY*, EQUI, and OPT were evaluated numerically using the MDP formulation given in Section 4.5. These heat maps look similar under various values of Λ_1 and Λ_2.*

Thus far, we have assumed that jobs are homogeneous with respect to speedup. In this section, we consider the case where jobs may have different speedup functions, and where the system knows the speedup function for each job. We will see in Section 4.2 that, in the case of multiple speedup functions, EQUI is no longer the optimal policy. In Section 4.3, we propose a class of policies called GREEDY which maximize the departure rate in every state. We then describe the optimal GREEDY policy, GREEDY* in Section 4.4. While GREEDY* is not optimal in general (see Section 4.6), we show that GREEDY* performs near-optimally in a wide range of settings (Sections 4.5 and 4.6).

Why Multiple Speedup Functions

There are situations in which it would be reasonable to expect all jobs to follow a single speedup function, such as when all jobs are instances of a single application. The PARSEC-3 benchmark provides many examples of workloads for which this is the case [27]. In practice, however, it may be the case that there are 2 or more classes of jobs, each with a unique speedup function reflecting the amount of sequential work, number of IO operations, and communication overhead that its jobs will experience when run across multiple servers. We consider the case where jobs may belong to one of two classes, each of which has its own speedup function, $\boldsymbol{s_i}$ (the classes may also have different arrival rates, $\boldsymbol{\Lambda_i}$, but we assume all job sizes to be exponentially distributed with rate $\mu = 1/\mathbb{E}[X]$). Without loss of generality, we assume that class 1 jobs are less parallelizable than class 2 jobs, that is

$$s_1(k) < s_2(k), \qquad \forall\, k > 1$$

For example, class 1 jobs could follow Amdahl's law with p = .5 while class 2 jobs follow Amdahl's law with p = .75. As usual, our goal is to describe and analyze scheduling policies which minimize overall mean response time across all jobs. We will assume that an allocation policy can differentiate between job classes when making scheduling decisions.

EQUI is No Longer Optimal

We have seen that EQUI is optimal when jobs are homogeneous with respect to speedup. One might assume that, since EQUI bases its decisions on the number of jobs in the system rather than the jobs' speedup functions, EQUI could continue to perform well when there are multiple speedup functions. However, it turns out that EQUI's performance is suboptimal even when there are just two speedup functions (see Figure 4). While EQUI's performance is actually close to optimal in the cases where $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ are similar ($\boldsymbol{p_1}$ is close to $\boldsymbol{p_2}$ in Figure 4), we see that EQUI's performance relative to OPT becomes worse as the difference between the speedup functions increases (p1 is far from p2 in Figure 4).
To see why EQUI is suboptimal in this case, recall that EQUI's optimality stems from the fact that it maximizes the rate of departures in every state when jobs follow a single speedup function (see proof of Theorem 1). When jobs are permitted to have different speedup functions, maximizing the rate of departures will require allocating more servers to class 2 jobs and fewer servers to class 1 jobs. Hence, EQUI no longer maximizes the total rate of departures when jobs follow different speedup functions.

A GREEDY* Class of Policies

Because EQUI fails to maximize the rate of departures when there are multiple speedup functions, we now identify a class of policies that does maximize the total rate of departures of the system. Specifically, we define the GREEDY class of policies to be the class of policies which achieve the maximal total rate of departures in every state.
To describe the policies in GREEDY, we consider any state $(\boldsymbol{x_1}, \boldsymbol{x_2})$ where there are $\boldsymbol{x_1}$ class 1 jobs and x_2 class 2 jobs in the system. Attaining the maximal rate of departures can be thought of as a two step process. First, a policy must decide how many servers, a1, to allocate to the x1 class 1 jobs. The remaining $\alpha_2 = n - \alpha_1$ servers will be allocated to class 2 jobs. Second, the policy must decide how to divide the $\boldsymbol{\alpha_1}$ servers among the class 1 jobs and the $\boldsymbol{\alpha_2}$ servers among the class 2 jobs. We have seen that, when dividing servers among a set of jobs with a single speedup function, EQUI maximizes the total rate of departures of this set of jobs. For a given choice of a1, the a1 servers should thus be evenly divided among the class 1 jobs and the α_2 servers should be evenly divided among the class 2 jobs in order to maximize the total rate of departures. To find the correct choice of a1, we define $\boldsymbol{\beta}(\boldsymbol{x_1}, \boldsymbol{x_2})$ to be the maximum rate of departures from the state $(\boldsymbol{x_1}, \boldsymbol{x_2})$ given that servers will be divided evenly within each class:

$$\beta(x_1, x_2) = \max_{\alpha \epsilon [0,n]} x_1 s_1 \left(\frac{\alpha}{x_1}\right) \mu + x_2 s_2 \left(\frac{n-\alpha}{x_2}\right) \mu. \quad (4)$$

Any policy in GREEDY must then choose α_1 such that

$$\alpha_1 \in \left\{\alpha : x_1 s_1 \left(\frac{\alpha}{x_1}\right) \mu + x_2 s_2 \left(\frac{n-\alpha}{x_2}\right) \mu = \beta(x_1, x_2)\right\}. \quad (5)$$

This same two-step process will generalize to the case when jobs follow more than 2 speedup functions. Crucially, note that GREEDY is truly a class of policies, since, in a given state, there may be multiple choices of a1 which satisfy (5). That is, there could be multiple allocations which achieve the maximal total rate of departures. For example, consider a system with n = 4 servers. If there are 4 class 1 jobs and 4 class 2 jobs, any choice of $\alpha_1 \in [0,4]$ results in the maximal rate of departures, $n\mu$. This begs the question of which policy from the GREEDY class achieves the best performance.

The Best GREEDY Policy: GREEDY*

We now define GREEDY*, a policy which dominates all other GREEDY policies with respect to mean response time. Consider two GREEDY policies, P_1 and P_2. In any state (x_1, x_2), both policies achieve the same maximal rate of departures. However, P_1 might achieve this rate by having a higher departure rate for class 1 jobs and a lower departure rate for class 2 jobs as compared to P_2. If class 1 jobs are less parallelizable than class 2 jobs, we say that P_1 "defers parallelizable work" in this state, which is a strategy that could benefit P_1 in the future. The GREEDY* policy is the GREEDY policy which in all states opts to defer parallelizable work when possible. Specifically, GREEDY* allocates a_1^* servers to class 1 jobs (the less parallelizable class), where a_1^* is the maximum value of a1 satisfying (5). That is,

$$a_1^* \in \left\{\alpha : x_1 s_1\left(\frac{\alpha}{x_1}\right)\mu + x_2 s_2\left(\frac{n-\alpha}{x_2}\right)\mu = \beta(x_1, x_2)\right\}.$$

In other words, a_1^* allows GREEDY* to attain the maximal rate of departures while also maximizing the rate at which class 1 jobs are completed. Theorem 2 shows that GREEDY* dominates all other GREEDY policies.

Theorem 1 *For any GREEDY policy, P,*

$$\mathbb{E}[T]^{GREEDY^*} \leq \mathbb{E}[T]^P.$$

Proof Consider the performance of GREEDY* and P on the state space $S = \{(x_1, x_2) : x_1, x_2 \in \mathbb{N}\}$. We will use the technique of precedence relations (see, e.g., [1,7]) to compare the mean number of customers, $\mathbb{E}[N]$, under GREEDY* to that under P. This requires that we define a value function for P, $V^P(x_1, x_2)$, and a cost function, $c(x_1, x_2)$. We define the cost of being in state (x_1, x_2) to be

$$c(x_1, x_2) = x_1 + x_2$$

so that the average cost of performing policy P is equal to the mean number of customers, $\mathbb{E}[N]$. We then define $V^P(x_1, x_2)$ to be the asymptotic total difference in the accrued cost under P when starting in state (x_1, x_2) as opposed to some designated reference state (see Appendix 6.1 for details). We require the following lemma which establishes a useful property of V^P.

Lemma 2 *For any GREEDY policy, P, and any* $(x_1, x_2) \in S$,

$$V^P(x_1 + 1, x_2) > V^P(x_1, x_2 + 1).$$

Proof See Appendix 6.1 for proof of Lemma 2.

We now prove Theorem 2 by contradiction. We begin by assuming that there exists a GREEDY policy $P \neq GREEDY^*$ which is optimal in terms of $\mathbb{E}[N]$ and thus $\mathbb{E}[N]^P \leq \mathbb{E}[N]^{P'}$ for any GREEDY policy P'. Since $P \neq GREEDY^*$, there exists some state, $(x_1, x_2) \in S$ where P and GREEDY* take different actions. Note that both x_1 and x_2 must be non-zero in this state, because otherwise there is only one action which will achieve the maximal rate of departures, and P and GREEDY* must therefore take the same action. We now consider a policy P' which takes the same action as GREEDY in state (x_1, x_2), and the same action as P in every other state. We can apply the technique of precedence relations described in [1,7] to show that $\mathbb{E}[N]^{P'} < \mathbb{E}[N]^P$. We start by defining $\gamma_i^Q(x_1, x_2)$ to be the rate of departures of class i jobs from the state (x_1, x_2) under policy Q. P' can now be obtained from P by taking $\gamma_2^P(x_1, x_2) - \gamma_2^{G^*}(x_1, x_2)$ away from the total completion rate of class 2 jobs and adding $\gamma_2^{G^*}(x_1,x_2) - \gamma_2^P(x_1,x_2)$ to the total completion rate of class 1 jobs, where G^* denotes GREEDY*. Theorem 3.1 in [1] tells us that $\mathbb{E}[N]^{P'} < \mathbb{E}[N]^P$ if:

$$\left(\gamma_2^{G^*}(x_1,x_2) - \gamma_2^P(x_1,x_2)\right) V^P(x_1-1,x_2) < \left(\gamma_2^P(x_1,x_2) - \gamma_2^{G^*}(x_1,x_2)\right) V^P(x_1,x_2-1)$$

To see that this property holds, first note that

$$\left(\gamma_1^{G^*}(x_1,x_2) - \gamma_1^P(x_1,x_2)\right) - \left(\gamma_2^P(x_1,x_2) - \gamma_2^{G^*}(x_1,x_2)\right)$$
$$= \left(\gamma_1^{G^*}(x_1,x_2) - \gamma_2^{G^*}(x_1,x_2)\right) - \left(\gamma_1^P(x_1,x_2) - \gamma_2^P(x_1,x_2)\right) = 0$$

since GREEDY* and P have the same (maximal) total rate of departures in every state. Thus,

$$\gamma_1^{G^*}(x_1,x_2) - \gamma_1^P(x_1,x_2) = \gamma_2^P(x_1,x_2) - \gamma_2^{G^*}(x_1,x_2)$$

By Lemma 2, we know that

$$V^P(x_1-1,x_2) < V^P(x_1,x_2-1).$$

This implies that $\mathbb{E}[N]^{P'} < \mathbb{E}[N]^P$ which contradicts our assumption that P is optimal in terms of $\mathbb{E}[N]$. By Little's Law, we can reformulate this in terms of $\mathbb{E}[T]$.
While GREEDY* is the best GREEDY policy, it will turn out that it is not optimal (see Section 4.6). Hence, we now turn our attention to computing the optimal policy.

Computing the Optimal Policy

The optimal policy, OPT, must not only consider the current state of the system when choosing how to determine the best partition (α_1,α_2), but must also consider the probabilities of transitioning to future states as well. To find a policy which balances this tradeoff between performance in the current state and future states, we formulate the problem as a Markov Decision Process (MDP).
We will consider an MDP with state space $S = \{(x_1,x_2) : x_1,x_2 \in N\}$, where x_i represents the number of class i jobs in the system. The action space in any state is given by $A = \{(\alpha_1,\alpha_2) : \alpha_1+\alpha_2 = n\}$. Let the arrival rate of class i jobs be given by Λ_i. Given an allocation of α_i servers to x_i type i jobs, it is optimal to run the x_i jobs on these servers using EQUI. Thus, given a state (x_1,x_2) and an action (α_1,α_2), the total departure rate of class i jobs from the system is given by

$$\mu_i(\alpha_i,x_i) := min\{\alpha_i,x_i\}\cdot\mu\cdot s\left(max\left\{1,\frac{\alpha_i}{x_i}\right\}\right).$$

We choose the cost function

$$c(x_1,x_2) = x_1 + x_2$$

such that the average cost per period equals the average number of jobs in the system, $\mathbb{E}[\boldsymbol{N}]$. We uniformize the system at rate 1 (always achievable by scaling time) and find that Bellman's optimality equations [18] for this MDP are given by

$$\mathbb{E}[\mathrm{N}^{\mathrm{OPT}}] + \mathrm{V}^{\mathrm{OPT}}(\mathrm{x}_1,\mathrm{x}_2) = \mathrm{A}^{\mathrm{OPT}}(\mathrm{x}_1,\mathrm{x}_2) + \mathrm{H}^{\mathrm{OPT}}(\mathrm{x}_1,\mathrm{x}_2)$$

Where

$$A^{OPT}(x_1,x_2) = c(x_1,x_2) + \Lambda_1(V^{OPT}(x_1+1,x_2) - V^{OPT}(x_1,x_2)) + \Lambda_2(V^{OPT}(x_1,x_2+1) - V^{OPT}(x_1,x_2)), \qquad (6)$$

And

$$H^{OPT}(x_1,x_2) = V^{OPT}(x_1,x_2) + \min_{(\alpha_1,\alpha_2)\in A} \{\mu_1(\alpha_1,x_1)(V^{OPT}((x_1-1)^+,x_2) - V^{OPT}(x_1,x_2)) + \mu_2(\alpha_2,x_2)(V^{OPT}(x_1,(x_2-1)^+) - V^{OPT}(x_1,x_2))\}. \qquad (\mathbf{7})$$

Here, the value function $V^{OPT}(x_1,x_2)$ denotes the asymptotic total difference in accrued costs when using the optimal policy and starting the system in state (x_1,x_2) instead of some reference state. While these equations are hard to solve analytically, the optimal actions can be obtained numerically by defining

$$V_{n+1}^{OPT}(x_1,x_2) = A_n^{OPT}(x_1,x_2) + H_n^{OPT}(x_1,x_2)$$

with $V_0^{OPT}(\cdot,\cdot) = 0$. Here, A_n^{OPT} and H_n^{OPT} are defined as in (6) and (7), but in terms of V_n^{OPT}. We can then perform value iteration to extract the optimal policy [20]. We use the results of this value iteration to compare the performance of OPT to both EQUI and GREEDY* in Figure 4. Note that using the same MDP formulation when there exists only one speedup function results in a much simpler expression for V^{OPT} which clearly yields EQUI.

GREEDY* is Near-Optimal

Surprisingly, even GREEDY* is not optimal for minimizing mean response time as shown in Figure 4 (although it is always within 1% of OPT in the figure). The same intuition that led us to believe that GREEDY* is the best GREEDY policy can be used to explain why GREEDY* is not optimal. We have already seen that deferring parallelizable work is advantageous to GREEDY*. However, we were only comparing GREEDY* to policies which use the maximal overall departure rate. It turns out that the advantage of deferring parallelizable work can be so great that a policy stands to benefit from using a submaximal overall rate of departures, deferring even more parallelizable work than GREEDY*.

FUTURE RESEARCH DIRECTIONS

One limitation of the model described in this chapter is that it assumes that jobs have no internal dependency structure. In reality, as the SPAA community has pointed out, a single job usually consists of multiple interdependent tasks. Hence, the effective speedup a job receives from parallelization may change over time. The problem of how to schedule a stream of jobs which have these dependency structures is open, even in the case of a single processor [29]. Scheduling these jobs on multiple servers therefore presents an exciting direction for future work. Additionally, it may be interesting to consider the case where the system has some partial information about job sizes. If job sizes were not exponentially distributed, for example, an allocation policy could try to predict a job's remaining size based on how long the job has been running. Similarly, the user may provide the system a noisy estimate of a job's size when submitting the job. In either case, the optimal allocation policy must both maintain system efficiency and also favor the jobs that are likely to have smaller remaining sizes. Although the results of this chapter do not model all of these aspects, they provide a starting point for further research in this area.

CONCLUSION

This chapter discusses the question of how to allocate servers to jobs in a stochastic model of a data center where jobs have sublinear speedup functions. While it is typical in these data centers for the user to specify the desired number of servers for her jobs, the results of this chapter suggest that allowing the system to schedule jobs can benefit overall mean response time. In the case where all jobs are malleable and follow the same speedup function, we prove that the well-known EQUI policy is optimal given that job sizes are exponentially distributed. When jobs are permitted to have multiple speedup functions, the question of optimal scheduling becomes even harder. We can show that EQUI is no longer optimal for scheduling malleable jobs when jobs follow multiple speedup functions. We see that the optimal policy (OPT) must balance the tradeoff between maximizing the total rate of departures in every state and deferring parallelizable work to preserve future system efficiency. We provide an MDP based formulation of OPT. As an alternative, we define a simple policy, GREEDY*, which can be easily implemented and performs near-optimally in simulation.

REFERENCES

1. A. Bušić, I. Vliegen, and A. Scheller-Wolf. Comparing Markov chains: aggregation and precedence relations applied to sets of states, with applications to assemble-to-order systems. Mathematics of Operations Research, 37:259-287, 2012.
2. A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In ACM SIGMETRICS Performance Evaluation Review, volume 37, pages 157-168. ACM, 2009.
3. Benjamin Berg, Jan-Pieter Dorsman, and Mor Harchol-Balter. Towards optimality in parallel scheduling. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 1(2):1-30, 2017.
4. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT ’08, pages 72-81, New York, NY, USA, 2008. ACM.
5. Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices, 49(4):127-144, 2014.
6. G. M. Koole. Monotonicity in Markov reward and decision chains: Theory and applications. Foundations and Trends in Stochastic Systems, 1:1-76, 2006.
7. I. Adan, G. J. J. A. N. van Houtum, and J. van der Wal. Upper and lower bounds for the waiting time in the symmetric shortest queue system. Annals of Operations Research, 48:197-217, 1994.
8. J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. In Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’09, pages 685-692, New York, NY, USA, 2009. ACM.
9. J. Edmonds. Scheduling in the dark. Theoretical Computer Science, 235:109-141, 1999.
10. J. McCool, M. Robison, and A. Reinders. Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, 2012.
11. J. N. Tsitsiklis and K. Xu. On the power of (even a little) centralization in distributed processing. ACM SIGMETRICS Performance Evaluation Review, 39:121-132, 2011.
12. K. Agrawal, J. Li, K. Lu, and B. Moseley. Scheduling parallelizable jobs online to minimize the maximum flow time. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’16, pages 195-205, New York, NY, USA, 2016. ACM.

13. K.-C. Huang, T.-C. Huang, Y.-H. Tung, and P.-Z. Shih. Effective processor allocation for moldable jobs with application speedup model. In Advances in Intelligent Systems and Applications - Volume 2, pages 563-572. Springer, 2013.
14. L. Kleinrock. Queueing Systems, Volume I: Theory. Wiley, New York, 1975.
15. M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. Computer, 41:33-38, 2008.
16. M. Harchol-Balter, A. Scheller-Wolf, and A. R. Young. Surprising results on task assignment in server farms with high-variability workloads. ACM SIGMETRICS Performance Evaluation Review, 37:287-298, 2009.
17. M. Harchol-Balter. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, 2013.
18. M. L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Chichester, 1994.
19. R. D. Nelson and T. K. Philips. An approximation for the mean response time for shortest queue routing with general interarrival and service times. Performance Evaluation, 17:123-139, 1993.
20. S. A. Lippman. Semi-Markov decision processes with unbounded rewards. Management Science, 19(7):717-731, 1973.
21. S. Chaitanya, B. Urgaonkar, and A. Sivasubramaniam. Qdsl: a queuing model for systems with differential service levels. ACM SIGMETRICS Performance Evaluation Review, 36(1):289-300, 2008.
22. S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. In Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER '03, pages 92-99, 2003.
23. S. V. Anastasiadis and K. C. Sevcik. Parallel application scheduling on networks of workstations. Journal of Parallel and Distributed Computing, 43:109 - 124, 1997.
24. S.-S. Ko and R. F. Serfozo. Response times in M/M/s fork-join networks. Advances in Applied Probability, 36:854-871, 2004.
25. V. Gupta, M. Harchol-Balter, K. Sigman, and W. Whitt. Analysis of join-the-shortestqueue routing for web server farms. Performance Evaluation, 64:1062-1081, 2007.
26. W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. Journal of Parallel and Distributed Computing, 62:1571-1601, 2002.
27. 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.
28. Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. Performance Evaluation, 68:1056-1071, 2011.
29. Z. Scully, G. Blelloch, M. Harchol-Balter, and A. Scheller-Wolf. Optimally scheduling jobs with multiple tasks. In Proceedings of the ACM Workshop on Mathematical Performance Modeling and Analysis, 2017.

APPENDIX 1

Proof of Lemma 2

Lemma 2 Given any GREEDY policy, P, for any $(x_1, x_2) \in S$,

$$V^P(x_1+1, x_2) > V^P(x_1, x_2+1)$$

Proof We begin by defining V_n^P as follows:

$$V_{n+1}^P(x_1, x_2) = A_n^P(x_1, x_2) + I_n^P(x_1, x_2)$$

Where

$$\begin{aligned} A_n^P(x_1, x_2) = c(x_1, x_2) + \Lambda_1\big(V_n^P(x_1+1, x_2) - V_n^P(x_1, x_2)\big) \\ + \Lambda_2(V_n^P(x_1, x_2+1) - V_n^P(x_1, x_2)\,) \end{aligned}$$

And

$$\begin{aligned} I_n^P = \gamma_1^P(x_1, x_2)\big(V_n^P((x_1-1)^+, x_2) - V_n^P(x_1, x_2)\big) \\ +\gamma_2^P(x_1, x_2)(V_n^P(x_1, (x_2-1)^+) - V_n^P(x_1, x_2)\,). \end{aligned}$$

and $V_0^P(x_1, x_2) = x_1 + x_2 + \frac{x_1}{x_1+x_2+1}$ for all $(x_1, x_2) \in S$.
Recall that $\gamma_i^P(x_1, x_2)$ denotes the departure rate of class i jobs from state (x_1, x_2) under policy P, Λ_i denotes the arrival rate of class i jobs, and $c(x_1, x_2) = x_1 + x_2$. From [18] we know that

$$\lim_{n\to\infty} V_n^P(x_1, x_2) - V_n^P(y_1, y_2) = V^P(x_1, x_2) - V^P(y_1, y_2).$$

Thus, if we can prove that our claim holds for $V_n^P(x_1, x_2)$ for all n ≥ 0, then it must hold for $V^P(x_1, x_2)$ as well (see, for example, [6]). We will now prove that all three of the following properties of V_n^P hold for all n ≥ 0 by induction:

$$\begin{gathered} V_n^P(x_1+1, x_2) > V_n^P(x_1, x_2) \\ V_n^P(x_1, x_2+1) > V_n^P(x_1, x_2) \\ V_n^P(x_1+1, x_2) > V_n^P(x_1, x_2+1) \end{gathered}$$

Note that the first two properties will be necessary for our proof of the third property, and the lemma follows directly from the third property.
We can easily verify that all three properties hold when n = 0 due to our choice of V_0^P. We now wish to show that if these properties hold for V_n^P, they must hold for V_{n+1}^P.
To prove property 1, we wish to show that

$$\begin{gathered} V_{n+1}^P(x_1+1, x_2) - V_{n+1}^P(x_1, x_2) \\ = A_n^P(x_1+1, x_2) - A_n^P(x_1, x_2) \\ +I_n^P(x_1+1, x_2) - I_n^P(x_1, x_2) > 0 \end{gathered}$$

We can easily see that $A_n^P(x_1+1, x_2) - A_n^P(x_1, x_2) > 0$, since the cost function $c(x_1, x_2)$ is increasing in x_1 and $V_n^P(x_1, x_2)$ is increasing in x_1 by the property 1 of the inductive hypothesis. To see that $I_n^P(x_1 + 1, x_2) - I_n^P(x_1, x_2) > 0$, we can expand the terms as follows:

$$\begin{gathered}
I_n^P(x_1+1,x_2) - I_n^P(x_1,x_2) \\
= \gamma_1^P(x_1,x_2)\big(V_n^P(x_1,x_2) - V_n^P((x_1-1)^+,x_2)\big) \\
+\gamma_2^P(x_1,x_2)\big(V_n^P(x_1,x_2) - V_n^P(x_1,(x_2-1)^+)\big) \\
+\gamma_2^P(x_1+1,x_2)\big(V_n^P(x_1+1,(x_2-1)^+) - V_n^P(x_1,x_2)\big) \\
+\Big(1-\gamma_1^P(x_1+1,x_2) - \gamma_2^P(x_1+1,x_2)\Big)\Big(V_n^P(x_1+1,x_2) - V_n^P(x_1,x_2)\Big).
\end{gathered}$$

Each line here is positive by the inductive hypothesis and the fact that

$$(1-\gamma_1^P(x_1+1,x_2) - \gamma_2^P(x_1+1,x_2)) \geq 0$$

since the system was uniformized to one. Thus property 1 holds. The proof of property 2 follows a very similar argument.
To show property 3 we wish to show that

$$\begin{gathered}
V_{n+1}^P(x_1+1,x_2) - V_{n+1}^P(x_1,x_2+1) \\
= A_n^P(x_1+1,x_2) - A_n^P(x_1,x_2+1) \\
+I_n^P(x_1+1,x_2) - I_n^P(x_1,x_2+1) > 0
\end{gathered}$$

We can easily see that $A_n^P(x_1+1,x_2) - A_n^P(x_1,x_2+1) > 0$ by the inductive hypothesis. To see that $I_n^P(x_1+1,x_2) - I_n^P(x_1,x_2+1) > 0$, we can expand the terms as follows:

$$\begin{gathered}
I_n^P(x_1+1,x_2) - I_n^P(x_1,x_2+1) \\
= \gamma_1^P(x_1,x_2+1)\big(V_n^P(x_1,x_2) - V_n^P((x_1-1)^+,x_2+1)\big) \\
+\gamma_2^P(x_1+1,x_2)\big(V_n^P(x_1+1,(x_2-1)^+) - V_n^P(x_1,x_2)\big) \\
+\Big(\gamma_1^P(x_1,x_2+1) + \gamma_2^P(x_1,x_2+1) - \gamma_1^P(x_1+1,x_2) - \gamma_2^P(x_1+1,x_2)\Big) \\
\times\big(V_n^P(x_1+1,x_2) - V_n^P(x_1,x_2)\big) \\
+\Big(1-\gamma_1^P(x_1,x_2+1) - \gamma_2^P(x_1,x_2+1)\Big) \\
\times(V_n^P(x_1+1,x_2) - V_n^P(x_1,x_2+1)).
\end{gathered}$$

We know, by assumption, that $s_1(k) < s_2(k)$ for $k > 1$, and thus the coefficient $\gamma_1^P(x_1,x_2+1) + \gamma_2^P(x_1,x_2+1) - \gamma_1^P(x_1+1,x_2) - \gamma_2^P(x_1+1,x_2)$ is non-negative. Therefore, all terms in this sum are positive by the inductive hypothesis, and property 3 holds.
All three properties therefore hold by induction, and $V^P(x_1+1,x_2) > V^P(x_1,x_2+1)$ as desired.