Thesis Proposal

A Principled Approach to Parallel Job Scheduling

Benjamin Berg

Carnegie Mellon University
Computer Science Department

May 10th, 2021

Thesis Committee

Mor Harchol-Balter, CMU (Chair)
Gregory R. Ganger, CMU
Benjamin Moseley, CMU
Christos Kozyrakis, Stanford
Weina Wang, CMU

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Contents

Chapter 1

Introduction

Parallelizable workloads are ubiquitous and appear across a diverse array of modern computer systems. Data centers, supercomputers, machine learning clusters, distributed computing frameworks, and databases all process jobs designed to be parallelized across many servers or cores. Unlike the jobs in more classical models, such as the M/G/k, which each run on a single server, parallelizable jobs are capable of running on multiple servers simultaneously. A job will receive some speedup from being parallelized across additional servers or cores, allowing the job to complete more quickly. However, jobs generally cannot be perfectly parallelized, and receive diminishing returns from being allocated additional servers.

When scheduling parallelizable jobs, a scheduling policy must decide how to best allocate servers or cores among the jobs in the system at every moment in time. The goal of this thesis is to develop and analyze improved scheduling policies for parallelizable jobs using the tools of performance modeling and stochastic analysis. In particular, given a set of K servers, we generally aim to design scheduling policies that reduce the *response times* of jobs – the times from when each job arrives to the system until it is completed.

The problem of scheduling parallelizable jobs presents unique challenges. Because jobs cannot typically be perfectly parallelized across additional servers, a scheduling policy must avoid highly inefficient allocations which give too many servers or cores to a single job. While the classic intuition in scheduling is to reduce the overall mean response time across jobs by completing short jobs before long jobs, it is not obvious what this means in the context of parallelizable jobs. In particular, a long job parallelized across sufficiently many servers can potentially be completed faster than a short job run on a single server. Furthermore, in practice, jobs have different levels of parallelizability. Some jobs might be highly parallelizable, running nearly k times faster when run on k servers. Alternatively, other jobs may be highly sequential, making highly inefficient use of additional servers. A good scheduling policy must decide how to account for both a job's remaining size and its level of parallelizability to determine efficient allocations that will result in low response times.

The State-of-the-Art In Scheduling Parallelizable Jobs

When scheduling parallelizable jobs, many real-world systems currently rely on reservation-based systems [22, 47, 41], where users reserve the number of cores or servers on which they want to run their jobs. Jobs are then packed onto servers in a largely first-come-first-served (FCFS) manner. Unfortunately, users tend to provision for the maximum degree of parallelism that their jobs will ever have, leaving reserved resources idle for a significant portion of the job's lifetime. For instance, a job with a fixed allocation of k servers might experience phases of sequential work where it is only able to utilize a much smaller number of servers. The result is that system designers are often forced to massively over-provision their systems in order to meet users' reservation requests and then run the systems at very low levels of utilization [47, 11].

Another approach used in real-world systems is to allow the system to control resource allocation for each job over time. Again, jobs are mostly served in FCFS order, but these systems employ a variety of greedy heuristic policies to try to maintain efficient allocations [32, 30, 11]. While these approaches have shown some success at reducing response times, these policies provide no provable performance guarantees and often require significant parameter tuning.

On the theoretical end, the theoretical computer science community (TCS) has been working on the problem of scheduling parallelizable jobs for the past 20 years (see Chapter 2). In online settings, where jobs arrive over time, strong lower bounds have been obtained on the achievable worst-case response time. In particular, the TCS community has shown that it is impossible to design a scheduling policy which performs within a constant factor of the optimal policy in the worst case [28].

The TCS community has employed various techniques and simplifications to circumvent these strong lower bounds. For instance, if all jobs are present at time 0, [14] shows that a simple policy called EQUI is within a factor of $2 + \sqrt{3}$ of the optimal policy in the worst case. The EQUI policy divides servers equally between all jobs in the system at every moment in time. When jobs arrive over time, the TCS literature again argues in favor of using EQUI [12] by employing a resource-augmentation argument to overcome the strong lower bounds.

In general, the the scheduling policies analyzed in the worst-case literature do not match the policies used in systems literature and deployed in real-world systems. This suggests an opportunity for continued research in the area of scheduling parallelizable jobs. This thesis seeks to address this gap by taking a three-pronged approach to the problem:

1. The theoretical models and style of analysis employed by the worst-case scheduling community has advocated the use of scheduling policies which do not perform well in real-world systems. We therefore begin by developing new models of parallelizable jobs in multiserver systems using the tools of stochastic performance modeling.

2. Using our newly developed models, we derive and analyze new scheduling policies which provably result in reduced job response times.

3. Finally, we validate our theoretical models through both simulation and real-world implementation. When necessary we design new heuristic policies based on our theoretical results that are suitable for deployment in real systems. Our goal here is to demonstrate that the policies inspired by our theoretical models represent an improvement over the heuristic policies currently used in the systems community.

Outline

We begin with an in-depth discussion of prior work in Chapter 2 and an overview of our modeling approach in Chapter 3. We then describe the central contributions of this thesis, which are as follows:

- **Scheduling with no information about jobs.** We begin by considering the case where a job's size is unknown to the system, and cannot be learned by the scheduler. The scheduler simply knows the average job size, and that all jobs follow a single speedup function. In this case we show that EQUI, a policy which dynamically allocates an equal number of servers to each job in the system at every moment in time, is optimal with respect to mean response time.

- **Scheduling jobs with size information.** Next, we consider the case where job sizes are known exactly to the system. When all jobs are present at time 0, we provide the optimal policy with respect to mean response time, heSRPT, which balances a tradeoff between biasing towards short jobs and maintaining overall system efficiency.

- **Scheduling jobs with different speedup functions.** Here we discuss the case where jobs are allowed to follow different speedup functions from job to job. We analyze a proportionally fair class of policies called GREEDY, and show that the an optimal policy must balance a tradeoff between maximizing the present efficiency of the system and the future efficiency of the system.

- **Scheduling jobs with multiple phases.** In addition to allowing speedup functions to differ from job to job we recognize that, in practice, a single job's scalability may vary over time based on the type of computation being performed. We develop a model for jobs composed of multiple phases, and provide optimality results in this model.

- **Scheduling jobs with general size distributions.** Allowing for fully general job size distributions increases the fidelity of our models, since real-world workloads often consist of jobs highly variable sizes. To analyze our models under these generally distributed workloads, we must develop a new form of heavy-traffic analysis which allows us to differentiate between policies for scheduling parallelizable jobs which follow general job size distributions.

Chapter 2

Prior Work

It is easiest to understand the prior theoretical work on scheduling parallelizable jobs in terms of the model of parallelism considered. We will therefore discuss several theoretical models of parallelism before considering prior work from the systems community on scheduling parallelizable jobs.

Jobs with Speedup Curves

The majority of our theoretical work considers the case where each job follows a *speedup function*, $s(k)$, that describes the speedup a job receives from running on k servers. Here, $s(k)$ is some positive concave non-decreasing function. Work using this model from the worst-case scheduling literature finds that, when job sizes are known, a generalization of `EQUI` is $\Omega(\log p)$-competitive with the optimal policy [24]. Moreover, `EQUI` is again shown to be constant competitive with constant resource augmentation [13, 15].

Overall, the general consensus from both the worst-case scheduling community is that `EQUI` should be used to achieve good or possibly optimal mean response time. This thesis will begin by showing conditions for the optimality of EQUI using our stochastic model using average-case analysis. However, we will see that these conditions are fairly restrictive (see Chapter 4, and that the performance of EQUI is often far from optimal both in theory and in practice. This discrepancy is largely due to the overly pessimistic nature of the prior work from the worst-case community, which all assumes that job sizes, arrivals, and speedup functions are adversarially chosen.

Jobs with Parallelizable Phases

Another body of work from the worst-case scheduling community [14, 13, 15, 12] considers jobs whose speedup functions change over time. This work considers the problem of scheduling parallelizable jobs composed of phases of differing parallelizability. However, due to the worst-case nature of the analysis, this work is forced to either consider an offline problem where all jobs arrive at time 0 [14], or to rely on resource augmentation[1] [13, 15, 12] to provide an algorithm which is within a (potentially large) constant factor of the optimal policy. This work concludes that the `EQUI` policy, as well as a generalization of it, is constant competitive given a small constant resource augmentation. We will see that the gap between EQUI and the optimal policy grows even wider using our stochastic models of jobs which consist of multiple phases. This difference has been corroborated by real-world systems experiments (see Chapter 7).

DAG Jobs

A separate branch of theoretical work on scheduling parallel jobs that developed concurrently with the above models considers every parallel job as consisting of a set of tasks with precedence constraints specified by a Directed-Acyclic-Graph (DAG). In this model, introduced in [7], a task can only run on a single server, but any two tasks that do not share a precedence relationship can be run in parallel. Much of the work

[1] Resource augmentation analysis is a relaxation of competitive analysis that, for some $s > 1$, compares an algorithm using speed s processors against the optimal policy using speed 1 processors.

in this area is concerned with how to efficiently schedule a *single* DAG job onto a set of servers [7, 8, 6]. When multiple DAG jobs arrive online, there are strong lower bounds on the competitive ratio of any online algorithm for mean response time [28]. Recently, [2] considered the online problem of scheduling a stream of DAG jobs to minimize the worst case mean response time. Using a resource augmentation argument, they show that `EQUI` and its generalization are constant competitive with constant resource augmentation.

The stochastic community has considered the related problem of *fork-join parallel* queueing. This model considers a special case of DAG scheduling where a system of K servers processes a stream of jobs consisting of K tasks with no precedence constraints. When a job arrives to the system, one task is immediately dispatched to each of the K servers. Results in this model have been hard to come by – the exact mean response time for this system is known only when task sizes are exponentially distributed and $K = 2$ [35]. Recent work has considered the idea of *limited fork-join* queueing, where each job consists of $k < K$ tasks [49], but this work provides only upper bounds on response time. In all of the work on fork-join queueing, all jobs have the same, fixed level of parallelism which cannot be changed by the system to improve response times.

Systems Literature

The need to schedule jobs with sublinear speedup functions has been corroborated across a wide range of systems. Perhaps most famously, the computer architecture community identified Amdahl's law [21] around the advent of multicore architectures. This problem is similarly known in the context of data center scheduling [11], supercomputing [45, 39], distributed machine learning [29], databases [19], and distributed computing frameworks such as MapReduce [48, 10]. Existing schedulers in these contexts are highly dependent on heuristics [33, 11, 20, 40], often require significant parameter tuning, and do not provide formal guarantees about performance.

Major advances from the systems community in the area of scheduling parallelizable jobs have instead focused more heavily on building scalable schedulers which are capable of checkpointing, preempting, and resuming jobs while maintaining low overhead and mitigating straggler effects. The Hopper system [38] demonstrates techniques for greatly reducing straggler effects by which the completion of a parallelizable computation is delayed by a small number of tasks which take longer to complete. We therefore assume that the scalability of a job can be measured by a speedup curve rather than modeling a parallelizable job at the task level. Meanwhile, advances in the Operating Systems community have demonstrated the ability to implement centralized scheduling policies which can preempt running jobs with mere microseconds of overhead in a multicore machine [25] or even a full rack of servers [54]. Similar results have been observed in the ability to share and dynamically re-allocate hardware accelerators such as GPUs for use in the training of machine learning models [51]. We will therefore consider scheduling policies which have the ability to preempt jobs and change their resource allocations with no overhead.

Our goal is to improve upon the heuristic policies used in these state-of-the-art systems by providing practical policies with provably optimal or near-optimal performance.

Chapter 3

Our Model

We now provide a description of the basic underlying model that will be used in our theoretical results. Throughout this thesis, we will continually tweak the specific assumptions or parameters of the model we are considering to derive results in different scenarios. However, this chapter will outline the basic assumptions of our model and the accompanying notation we will use throughout the thesis.

We assume that our system consists of K homogeneous servers. While we will use the terminology of parallelizing jobs across *multiple servers* in our theoretical results, the policies we derive could apply equally to scheduling jobs across *multiple cores* in a multicore machine, rack-scale computer, or supercomputer.

We are interested in the case where a stream of parallelizable jobs arrive to the system over time. We will therefore assume that jobs arrive to the system according to a stationary stochastic process with an arrival rate of λ jobs per second. We will assume this arrival process is Poisson unless otherwise noted.

Each arriving job has an associated *inherent job size*, X, which is defined as the job's running time when run on a single server. A job's size represents the amount of work associated with a job. For instance, a database query may have some number of table rows it must examine, or an ML training job may be required to complete some number of iterations of gradient descent. To standardize the definition of job size across these applications, we use the job's single-server running time, which we assume is directly proportional to the amount of work the job must perform. We will generally consider each job size, X, to be a random variable whose value and distribution may be known or unknown to the system.

We are often interested in how long a job will run if allocated $k > 1$ servers. We therefore define a speedup function $s(k) : \mathbb{R}^+ \to \mathbb{R}^+$ which tells us how many times faster a job will complete if run on k servers instead of a single server. We assume that $s(k)$ is a sublinear, non-decreasing, concave function, a phenomenon that has been observed across a wide array of systems and applications [53, 29, 36]. Given a job's size, X, and speedup function, $s(k)$, the running time of a job on k servers, X_k is defined to be

$$X_k = \frac{X}{s(k)} \quad \forall k > 0.$$

One popular speedup function described in the literature is *Amdahl's law*[21], which considers every job as being composed of some fraction, f, of work which is perfectly parallelizable while the remaining $(1 - f)$ fraction of the job receives no speedup from parallelization. Hence, Amdahl's law defines $s(k)$ to be

$$s(k) = \frac{1}{\frac{f}{k} + (1 - f)}.$$

Other theoretical work has used a simple *power law speedup function* to approximate a variety of empirically derived speedup curves[24]. A power law speedup function is defined to be of the form

$$s(k) = k^p \quad 0 < p \leq 1.$$

Because a job's server allocation may change over time, we must define the running time of a job whose server allocation is not fixed for its entire lifetime. We define a job's *remaining size* at time t, $X(t)$, to be

the remaining time required to complete the job using a single server. If a job receives an allocation of k servers from time t until time t', the job's remaining size at time t' is defined to be

$$X(t) - (t' - t)s(k).$$

Here, $s(k)$ is effectively denoting how many times faster a job's work is completed on k servers than on a single server. This notion is completely compatible with our prior definition.

We are generally interesting in maintaining low *response times* across the stream of incoming jobs. A job's response time, which we denote with the random variable T, is defined to be the time between when a job arrives to the system and when it is completed. We will be interested in scenarios where the limiting distribution of T exists, in which case we will often seek to minimize the *mean response time*, $\mathbb{E}[T]$, across jobs. It will also often be useful to consider a related quantity, the random variable N, which denotes the number of jobs in the system.

In order for the limiting distribution of response time to exist, it suffices to show that the system is *stable*. We define the *system load*, ρ, of a system to be

$$\rho = \frac{\lambda \mathbb{E}[X]}{K}.$$

Following standard queueing-theoretic techniques, one can often show that a system is stable if

$$\rho < 1 \quad \text{and} \quad \mathbb{E}[X^2] < \infty.$$

We will explicitly note when these conditions do not suffice for stability.

Our goal is to derive and analyze *scheduling policies* which define the number of servers allocated to each job in the system at every moment in time. We will assume that jobs can receive fractional server allocations. Hence, for a scheduling policy P, we will define an allocation function, $\boldsymbol{\theta}^P(t)$, which defines the fraction of the K servers allocated to each job in the system at time t. Specifically, at time t, when there are $N(t)$ jobs in the system,

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

where

$$0 \leq \theta_i^P(t) \leq 1 \quad \forall 1 \leq i \leq N(t) \quad \text{and} \quad \sum_{i=1}^{N(t)} \theta_i^P = 1.$$

We define the instantaneous efficiency of the system under a policy P at time t to be the sum of the speedups received by the jobs currently in the system. That is,

$$\text{System efficiency at time } t = \sum_{i=1}^{N(t)} s(\theta_i^P(t) \cdot K).$$

As long as speedups are not superlinear, the maximum system efficiency is K. System efficiency can be thought of as a measure of how much overhead due to paralellism is being incurred by the current server allocation.

Chapter 4

Parallelizable Jobs with No Information

To begin, we will examine the case where a scheduler is limited in its knowledge of the job sizes and scaling behavior. Specifically, we assume that all job sizes are unknown to the system and are drawn from a single, exponential distribution with rate μ. Hence, while the average job size is known, the scheduler cannot differentiate between jobs by predicting which jobs have a smaller remaining size. Furthermore, the scheduler cannot learn additional information about remaining a job's size by running the job. Finally, all jobs are assumed to follow the same concave, increasing speedup curve, $s(k)$, which is known to the system.

We first consider the case where jobs are *malleable*, meaning their allocations may change over time with no overhead. We then consider the case where jobs are *moldable*, meaning a job can run on any number of servers, but may not change the servers on which it runs over time.

4.1 Motivation

This problem warrants study for two main reasons.

First, some system schedulers do not use remaining job size predictions to make scheduling decisions. These frameworks are forced to be agnostic about the workloads that they run either because information on job size is not available, or computing and utilizing this information to make scheduling decisions introduces too much overhead. For example, consider a general-purpose operating system scheduler in a multicore server. If this operating system is tasked with running several instances of the same parallelizable application, it must decide how to schedule each process without using predictions about the processes remaining size. In this case, it is also reasonable to assume that each process will roughly follow the same speedup curve.

Second, this problem provides a first queueing-theoretic model of scheduling parallelizable jobs that follow speedup curves. If job sizes are known, or are unknown but follow a general distribution, deriving optimality results for parallel scheduling requires more advanced analytical techniques. We will address these more complex methods later in this thesis. However, by making these jobs follow a single exponential size distribution and single speedup curve, we isolate the problem of how to best exploit each job's speedup function in order to minimize mean response time. This will provide a foundation that will help us reason about more complex settings later in the thesis.

4.2 Contributions

The core results in the model come from [3]. Namely:

- When jobs are malleable, EQUI is optimal with respect to mean response time. EQUI minimizes mean response time by maximizing efficiency at every moment in time. This is analogous to prior worst-case results that show EQUI is $O(1+\epsilon)$-speed competitive with the optimal policy [12]. Our proof argues

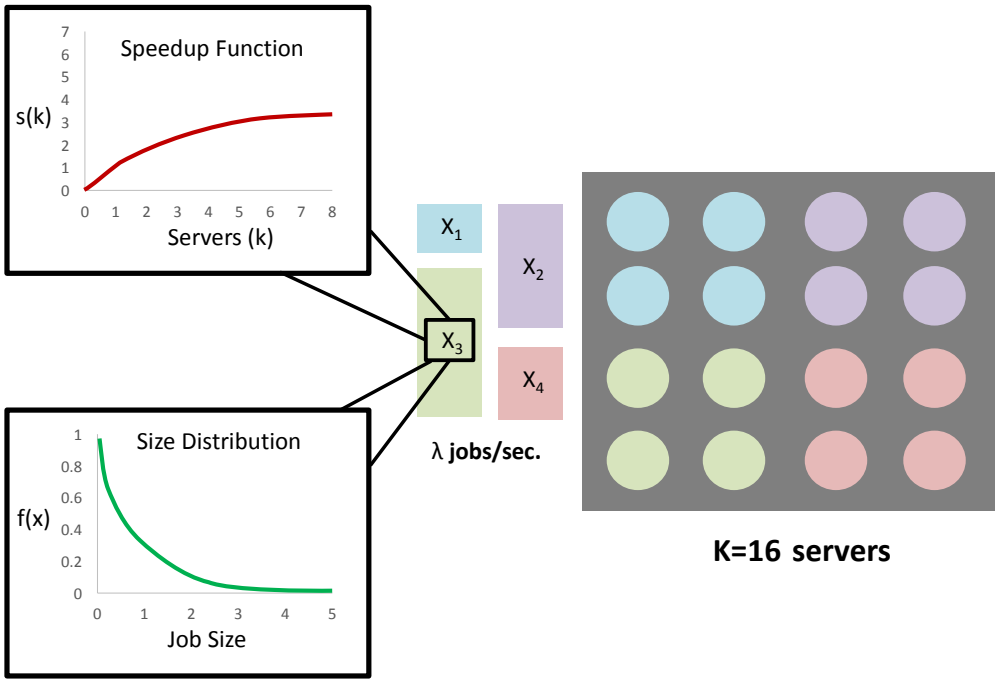

Figure 4.1: The model analyzed in Chapter 4. A system of $K = 16$ servers processes a stream of jobs which arrive according to a Poisson process with rate λ. All jobs follow the same speedup function, $s(k)$. All job sizes are unknown and exponentially distributed with rate μ. Here, there are 4 jobs in the system, and the servers are split equally according to the EQUI policy.

that by maximizing system efficiency in every state, EQUI minimizes the mean response time across jobs.

- When jobs are moldable but not malleable, we analyze fixed-width policies with both random routing and JSQ routing. As expected, there is a big advantage to doing JSQ routing. We use our analysis to compute the best chunk size for a fixed-width policy. In addition to working with moldable jobs, fixed-width policies do not require any preemptions. An example of this analysis is shown in Figure 6.1.

- As the number of servers becomes large, we show that the optimal fixed-width policy is approximately as good as EQUI. Thus, we can achieve near-optimal performance in this large-system limit without requiring *any* preemptions or malleable jobs. This result follows from a limiting analysis of the Markov chain for EQUI which leverages precedence relations [9].

4.3 Impact

The worst-case literature introduced EQUI almost 20 years ago[12]. Since then, EQUI and its variants have been shown to be competitive under various sets of modeling assumptions [14, 12, 13, 15] *in the worst case*. Because this work uses competitive analysis, it was not previously clear whether EQUI would actually perform well in the average case, if EQUI is only competitive with optimal under highly adversarial workloads, or if EQUI is simply a mediocre policy that happens to be amenable to worst case analysis.

The above stochastic results give context to the prior work surrounding EQUI. In particular, we show that under a somewhat restrictive set of assumptions, EQUI actually *is* the optimal policy. In some sense, our result corroborates the result from [12] in finding that EQUI performs well when a scheduler cannot

make decisions based on the sizes and speedup functions of each job. The setting considered in this seminal worst-case paper is one where the scheduler has no information about job sizes or a job's speedup curve, but must compete against an adversary with perfect information. Our work finds that EQUI is optimal if the scheduler is similarly deprived of information about the jobs – in our setting job sizes are not known, sizes cannot be learned over time, and all jobs follow the same speedup function.

However, our analysis also illustrates the power of a stochastic modeling approach over worst-case analysis. When we begin to change the assumptions of our model to better suit a range of real-world systems (as we will do in the subsequent chapters of this thesis), EQUI quickly becomes suboptimal, and provably worse than a variety of other policies we will consider. Worst-case analysis is so pessimistic that it cannot differentiate between the performance of many (possibly impractical) policies. By specifying the conditions for the optimality of EQUI, we are laying the groundwork to distinguish between these policies using stochastic analysis.

Completion Percentage: 100%

Chapter 5

Parallelizable Jobs with Size Information

We now consider the case where job sizes are exactly known to the system. These job sizes may be drawn from some distribution, but the scheduler has, at every moment in time, perfect information about the remaining sizes of each job in the system. We continue to consider the case where all jobs follow the same, single speedup function, which is known to the system. Here, we constrain the form of the speedup function to be $s(k) = k^p$ for the purposes of analytical tractability.

5.1 Motivation

In a wide variety of systems, highly accurate estimates of a job's inherent size can be made available to the scheduler [31]. Unlike the previous chapter, where we assumed that jobs sizes could not be estimated or learned over time, we now consider the far other end of the spectrum. Here, we assume that the system either has a very well-defined notion of inherent work, as it would in a database or ML training context, or that the system has run for long enough that job sizes have been learned with a high degree of accuracy.

Our prior assumptions were designed to prevent the scheduler from having to make decisions based on the remaining sizes of each job, but we now wish to examine the trade-offs that different job sizes introduce to the system. Specifically, classical queueing and scheduling theory tells us that completing shorter jobs before longer jobs will reduce mean response time [44, 16]. In fact, in single-server scheduling, giving priority to the job with Shortest Remaining Processing Time (SRPT) is optimal.

While we are capable of running SRPT, allocating all servers to the shortest remaining job at every moment in time, this can be highly inefficient due to the sublinear speedup that each job receives from additional servers. Hence, the optimal policy must balance a new tradeoff. On the one hand, biasing our allocations towards the shortest jobs in the system will cause short jobs to complete before long jobs. On the other hand, EQUI still maximizes efficiency in this setting, and thus biasing our allocations towards any one job or set of jobs will reduce to overall efficiency of the system. The optimal policy in this case must carefully balance this tradeoff between overall system efficiency and the desire to favor short jobs.

5.2 Contributions

heSRPT: How to schedule jobs of known size

To isolate and study this tradeoff, we consider the case where jobs are all present at time 0 and follow a single speedup function of the form $s(k) = k^p$ [5]. In this case we can find the optimal policy with respect to mean response time by decomposing the problem into two parts. First, we show that the optimal policy should complete jobs in shortest-job-first order. Then, we prove a property of the optimal policy called the *scale-free property* that shows us how to find the optimal policy that completes jobs in shortest-job-first order.

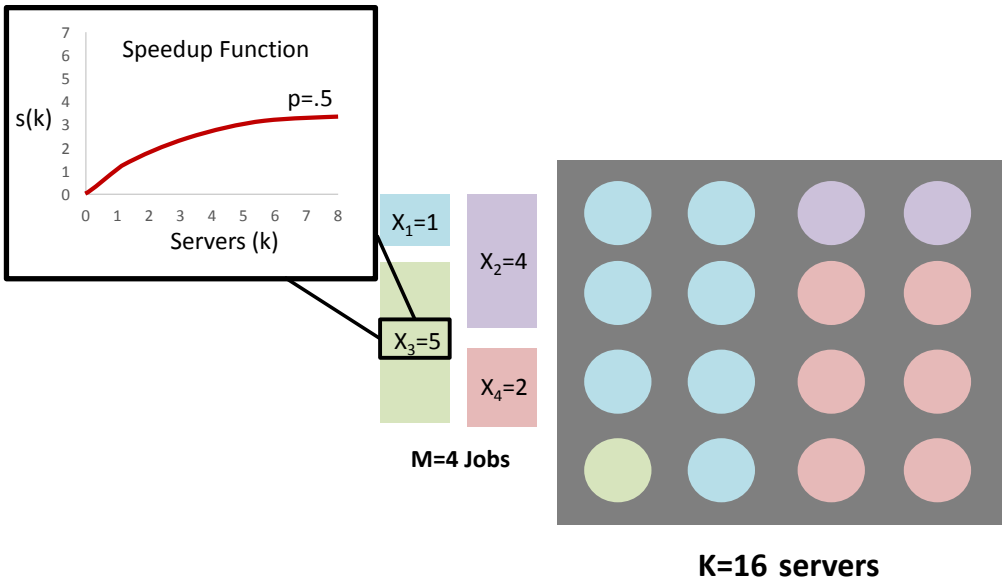

Figure 5.1: The model analyzed in Chapter 5. A system of $K = 16$ servers processes a set of $M = 4$ jobs of known size All jobs follow the same speedup function, $s(k)$, which assumed to be of the form $s(k) = k^p$. Here, $p = .5$. The optimal policy, heSRPT, allocates more servers to the shorter jobs at every moment in time. We also consider the case where jobs arrive to the system over time in Chapter 5.

Metrics beyond response time

We also show how this result generalizes for a wider class of weighted response time metrics beyond mean response time. For each job, i, we assign a weight w_i. The weighted response time for a set of M jobs is then defined to be

$$T^W = \sum_{i=1}^{M} w_i \cdot T_i.$$

Without loss of generality we define

$$x_M < x_{M-1} < \ldots < x_1$$

to be the sizes of the M jobs. We say that the weights *favor small jobs* when

$$w_M > w_{M-1} > \ldots > w_1.$$

We show how our result generalizes to provide the optimal policy for any weighted flow time metric where weights favor small jobs. Crucially, this class of metrics includes mean slowdown, where $w_i = \frac{1}{x_i}$. Intuitively, slowdown measures how much each job was interfered with by the other jobs in the system, and is a popular metric when scheduling parallelizable jobs in many server systems [34, 23, 50].

Adding Arrivals

Adding arrivals makes this problem very difficult. This problem is at least as difficult as scheduling sequential jobs of known size on k parallel servers, which is along standing open problem in both the queueing and worst-case scheduling communities. However, we use our offline policy to generate a heuristic policy, *Adaptive heSRPT* (A-heSRPT), which uses the allocations recommended by heSRPT, recalculating the all allocations on each arrival or departure. Figure 5.2 shows how A-heSRPT greatly outperforms a variety of other heuristic policies from the literature in simulation when calibrated to minimize mean slowdown. Specifically, we compare A-heSRPT to EQUI, HELL and KNEE [29], RS[50], and SRPT. We see similar results when A-heSRPT is calibrated to minimize mean response time.

To move beyond heuristic policies in the known-job-size case, we must use asymptotic analysis developed by the performance modeling community. For example, SRPT-K is known to be heavy-traffic optimal [16],

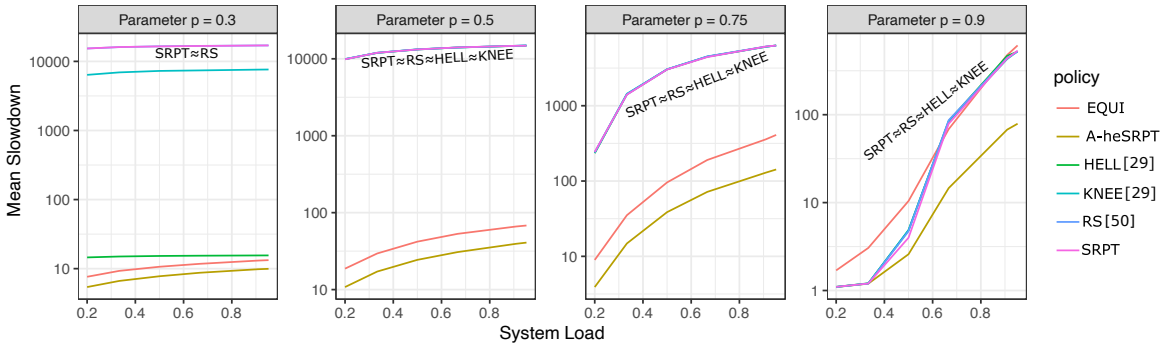

Figure 5.2: Mean slowdown of A-heSRPT and policies from the literature in the online setting. Simulations assume $N = 10,000$ servers and a Poisson arrival process. Job sizes are Pareto($\alpha = 1.5$) distributed. Each graph shows mean slowdown for a different power law speedup function, $s(k) = k^p$, with one value of the speedup parameter, p. A-heSRPT often dominates by an order of magnitude.

and one might wonder whether an analogous result holds for heSRPT. We will defer our discussion of asymptotic analysis to Chapter 8.

5.3 Impact

This result shows that by making a simple tweak to the assumptions in our model, the performance of EQUI becomes greatly suboptimal. By failing to bias toward shorter jobs in the system, EQUI can be more than a factor of 2 worse than heSRPT when all jobs are present at time 0.

When we move to the online case, even though we are employing a heuristic policy based on heSRPT, we see that by balancing efficiency and the desire to complete short jobs, heSRPT greatly outperforms the competition. Particularly when considering weighted response time metrics such as slowdown, heSRPT can outperform competitor policies by orders of magnitude.

Another important aspect of our result is that it provides good intuition for how to design policies when job sizes are unknown but generally distributed. In these cases, we will not have exact job sizes, but rather estimates based on the ages of the jobs in the system. Understanding how to schedule jobs when sizes are known exactly is the first step in scheduling under the uncertainty of generally distributed job sizes. We will discuss how to build in this direction by using the Gittins Index in Chapter 8.

The problem of how to schedule jobs with known sizes is particularly important in modern systems where job size variability is quite high [46]. In these systems, there will be and outsized benefit to prioritizing short jobs over long jobs in terms of reducing mean response time. Furthermore, there are a wide variety of systems where job sizes are known or can be estimated with ahigh degree of accuracy. Databases, for example, use cost-based query optimizers [42], giving the scheduler a natural way to compare the relative sizes of different queries. Cluster scheduling frameworks have also been shown to be capable of accurately estimating a job's run time[11]. When training machine learning models, the number of iterations of gradient descent that user wishes to perform is also an excellent indicator of a job's size [29]. In each of these cases, mean response time could be greatly reduced by using heSRPT to favor short jobs while maintaining overall system efficiency.

Completion Percentage: 100%

Chapter 6

Parallelizable Jobs with Different Speedup Functions

In this chapter, we consider jobs which follow different speedup functions from job to job. Specifically, we assume the existence of 2 *classes* of job, each with its own speedup function (e.g. $s_1(k)$ and $s_2(k)$). To examine the effects of having these heterogeneous classes of jobs, we again assume that job sizes are exponentially distributed, where the jobs of each class follow a corresponding exponential distribution (e.g. $Exp(\mu_1), Exp(\mu_2)$). We assume that the scheduler has knowledge of each job's class. That is, each job's size distribution and speedup function are known to the scheduler.

6.1 Motivation

In addition to job sizes varying between jobs, the scalability of each job may be different in practice. For example, some database queries might be highly parallelizable table scans for analytical queries while others might be point lookups which are not easily parallelized. Distributed ML frameworks also have this issue – training jobs are designed to be highly parallelizable while model-serving queries are less parallelizable.

From the outset, it is clear that EQUI will no longer be the optimal policy in this case. The proof of optimality for EQUI in Chapter 4 relies on the fact that EQUI maximizes system efficiency when all jobs follow the same speedup function. This is clearly no longer the case when jobs follow different speedup functions. Here, an efficiency maximizing allocation must bias towards the more parallelizable jobs.

This begs the question of whether or not an efficiency-maximizing policy is still optimal when jobs follow different speedup functions. In particular, are there any potential trade-offs introduced by having multiple speedup functions that could make such a policy suboptimal? Also, how does the problem change when a job's size might be correlated to it speedup function. Specifically, how should we handle the common case where larger jobs tend to be more parallelizable than shorter jobs?

6.2 Contributions

General Speedup Functions

To address this question, we begin by considering the setting where jobs follow a single exponential size distribution, but each job now belongs to one of two scalability classes.

- The natural generalization of EQUI to the case of multiple speedup functions is GREEDY, a class of policies which maximizes the instantaneous efficiency of the system at every moment in time. Note that this is similar to the concept of a Proportionally Fair scheduling policy where we account for each individual job's speedup function, and thus is an intuitively fair policy. We can show by counterexample, however, that GREEDY is not optimal. These counterexamples hinge on the concept of *deferring parallelizable work*. That is, one can choose a non-GREEDY allocation which allocates fewer servers

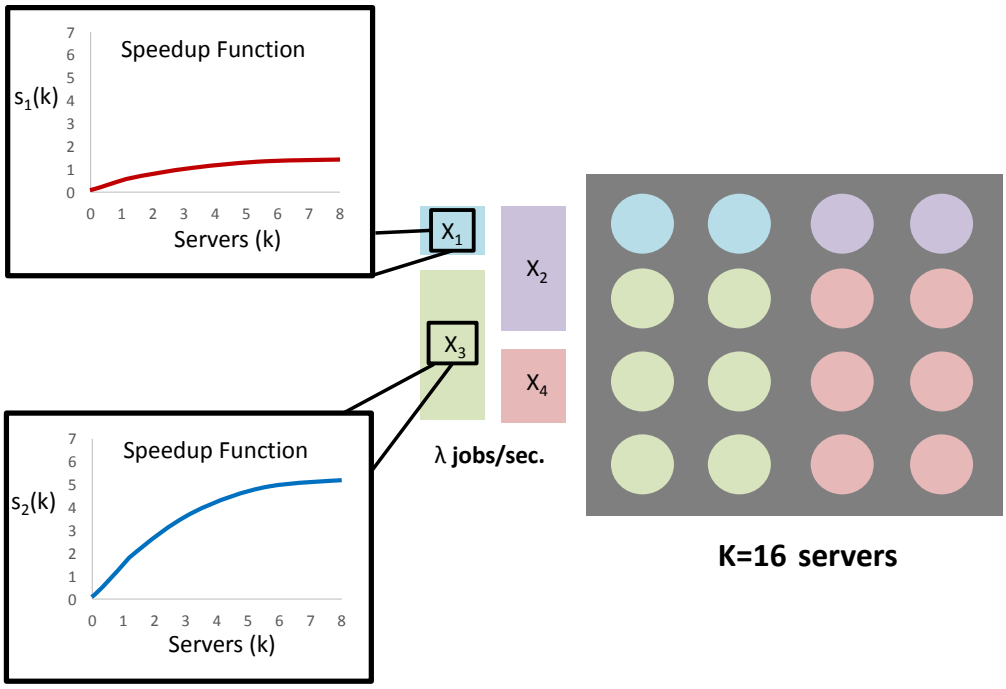

Figure 6.1: The model analyzed in Chapter 6. A system of $K = 16$ servers processes a stream of jobs arriving to the system over time according to a Poisson process with rate λ. Jobs follow may follow a less parallelizable speedup function, $s_1(k)$, or a more parallelizable speedup function, $s_2(k)$. Here, servers are allocated according to a GREEDY policy, which maximizes system efficiency in the present state by biasing towards more parallelizable jobs.

to the more parallelizable jobs, lowering the instantaneous efficiency of the system. However, this non-GREEDY allocation may increase the future efficiency of the system massively by saving highly parallelizable jobs for times when fewer jobs are in the system. During these future periods with few jobs, the remaining parallelizable jobs will be able to efficiently use the available servers.

- To underscore the importance of deferring parallelizable work, we show that the best GREEDY policy is GREEDY*, which defers parallelizable work as much as possible while maintaining the maximal instantaneous system efficiency. GREEDY* is not optimal in general, but performs within 1% of the optimal policy in a wide range of simulations.

Elastic and Inelastic Jobs

Matters become even more complex when job sizes are correlated with job scalability. For instance, in the case of database queries, we know that point lookups are relatively short queries while analytical queries require much more inherent work. In fact, this is the common case – programmers make larger jobs highly parallelizable to shorten their run times, and leave the shorter jobs less parallelizable due to the dubious benefits of parallelization in these cases (e.g. overheads are too high to make parallelism effective here).

To model this practical case, we specifically consider the case where some jobs are small on average and are totally sequential (inelastic) while others are large on average and are perfectly parallelizable (elastic) [4].

- The optimal policy in this case is Inelastic-First (IF) which gives strict preemptive priority to inelastic jobs. We note that IF is equivalent to GREEDY* in this case.

- The key observation here is that the desire to defer parallelizable work and the desire to complete short jobs are aligned.

- When elastic jobs are smaller on average than inelastic jobs, which is much less common in practice, we don't know the optimal policy. However, we can differentiate between the performance of IF and the Elastic-First policy (EF), which gives strict priority to elastic jobs.

6.3 Impact

Our results about GREEDY policies again show that EQUI is far from optimal under even mild, realistic assumptions about the workload being processed. However, our results about deferring parallelizable work also illustrate how the heuristics commonly used in real-world systems [11, 54] can fall short. The common wisdom in the systems community is to greedily maximize the "throughput" of the system, which in our open-system model of parallelizable jobs is equivalent to instantaneous system efficiency. However, this heuristic can be suboptimal.

Greedily maximizing the instantaneous system efficiency fails to consider the future value of having parallelizable jobs in the system. As such, GREEDY policies will tend to drift towards states with a relatively higher percentage of non-parallelizable work in the system, reducing future system efficiency. While finding the exact closed-form of the policy which best balances this tradeoff is complex, we provide several results which successfully blend the idea of deferring parallelizable work with the popular heuristic of maximizing instantaneous efficiency. Specifically, we prove that GREEDY*, which is both GREEDY and defers parallelizable work, is the best GREEDY policy. This result shows an improvement over both the existing worst-case theory results and state-of-the-art systems scheduling policies.

Despite the concern the GREEDY policies may not be optimal in general, our theoretical results on scheduling elastic and inelastic jobs identify many highly practical scenarios where a GREEDY policy, IF, will perform optimally. For example, modern machine learning frameworks [33] aim to both train models via highly elastic training jobs and serve models via largely inelastic inference jobs. Similarly, HTAP databases aim to both process highly elastic analytical queries and highly inelastic transactions [37]. Even in Supercomputing environments the scheduler must decide how schedule both malleable (elastic) jobs and moldable (inelastic) jobs [17]. All of these scenarios are good candidates for the use of an IF scheduling policy to reduce mean repsonse time.

Completion Percentage: 100%

Chapter 7

Parallelizable Jobs With Multiple Phases

In this chapter, in addition to allowing speedup curves to vary from job to job, we allow a single job's speedup curve to change over time. To capture this phenomenon, we consider jobs composed of *phases*, where each phase has an associated inherent size and speedup function. The phases of a job must all be completed in order for a job to be considered complete. We are interested in the setting where each job's current phase is visible to the scheduler, and thus the scheduler can leverage a job's phase information to improve scheduling decisions.

To approximate how a job's scalability changes over time, we will assume that each job phase is either elastic or inelastic. To model how jobs move from phase to phase over time, we will assume that each job can modeled as a Markov chain of the form shown in the Phase Transitions section of Figure 7.1. Hence, the duration of each phase is exponentially distributed according to the transition rates of the underlying Markov chain. A job can start in either an elastic or inelastic phase, and must be run until it reaches the absorbing *completion state*, at which point the job is considered complete. We assume that all jobs transition between phases according to the same underlying Markov chain.

7.1 Motivation

Up to this point, we have allowed job parallelizability and job size to change from job to job. All of the prior models we have considered are firmly rooted in the view that a job's scalability can be fully captured by its speedup function. This is a useful model, particularly when a job oscillates between parallelizable and sequential work so quickly that a scheduler cannot effectively differentiate between these phases.

However, a speedup function fundamentally measures a job's average scalability over the course of its entire lifetime. In many cases, however, a job's scalability at any single moment in time is determined by the type of computation occurring at that moment. Prior work on a wide range of systems [27, 29, 10] has repeatedly acknowledged that jobs often consist of multiple phases which are discernible to the scheduler. Some of the phases may be highly parallelizable, while others are less parallelizable or even completely sequential. In these cases, a scheduler could benefit from explicitly considering a job's current and future phases rather than just considering its average scalability over time via a speedup function.

For instance, a job may consist of some sequential work to load data in to memory, some highly parallelizable work to process this data, and some sequential work to coalesce the results of the data processing. If we assume that the job spends 90% of its time in the parallelizable phase when run on a single core, this job would have a speedup function that is well-modeled by Amdahl's law with a parallelizable fraction of $f = .9$. The scheduling policies developed in the previous 3 chapters would not change this job's allocation as it moved from phase to phase. However, we can easily see that this is highly inefficient. A better scheduling policy would allocate relatively few servers to this job during is first and final phase, and would allocate many servers to the job during its parallelizable phase. Hence, we must investigate how to effectively leverage knowledge of the phases of a job without incurring a prohibitive number of expensive preemptions.

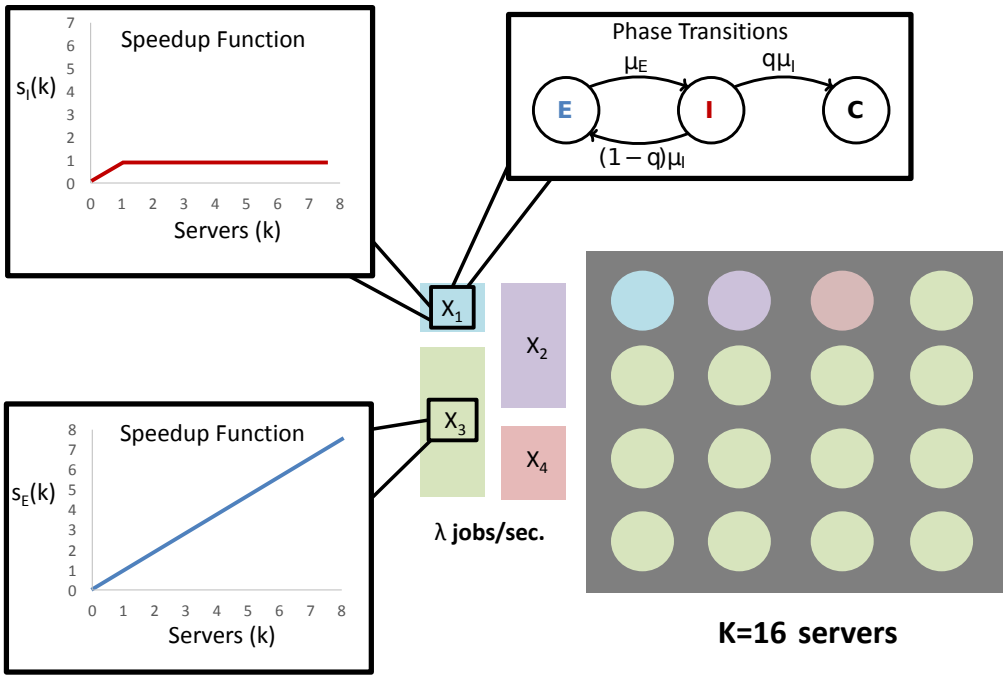

Figure 7.1: The model analyzed in Chapter 7. A system of $K = 16$ servers processes a stream of jobs arriving to the system over time according to a Poisson process with rate λ. Each job is composed of elastic and inelastic phases, and transitions between these phases according to continuous time Markov chain. A jobs speedup function depends on its current phase at every moment in time. Here, servers are allocated according to the IF-Phase policy, which gives strict priority to jobs which are in an inelastic phase (jobs 1,2, and 4) and allocates any remaining servers to jobs in an elastic phase (job 3).

7.2 Contributions

Elastic and Inelastic Phases

Using the Markov chain structure shown in Figure 7.1, we can investigate the mean response time under a variety of scheduling policies.

- We show that the pipeline phases of various queries from the Star Schema Benchmark (SSB) [36] greatly resemble the series of elastic and inelastic phases considered by our theoretical model of jobs with phases. This is shown in Figure 7.2.
- For any setting of the rates in Figure 7.1, we can show that the optimal policy with respect to mean response time is to give strict priority to the jobs which are in inelastic phases. We call this policy IF-Phase. IF-Phase greatly outperforms EQUI, FCFS, and a policy which gives priority to jobs in an elastic phase (EF-Phase).
- While the Markov chain model we use restricts the size distributions of each phase to be exponential, we can show that our performance advantage over other intuitive scheduling policies largely holds for more variable phase-size distributions.
- Even when we consider real world phase distributions taken from the SSB in databases, we find a significant benefit to using IF-Phase over the state of the art scheduling policies used in real databases.

Scheduling Phase-Based Jobs in Practice

The scheduling policies discussed in prior chapters were simple enough to be implemented using state-of-the-art techniques in a wide variety of systems. That is, implementing these simple policies would represent

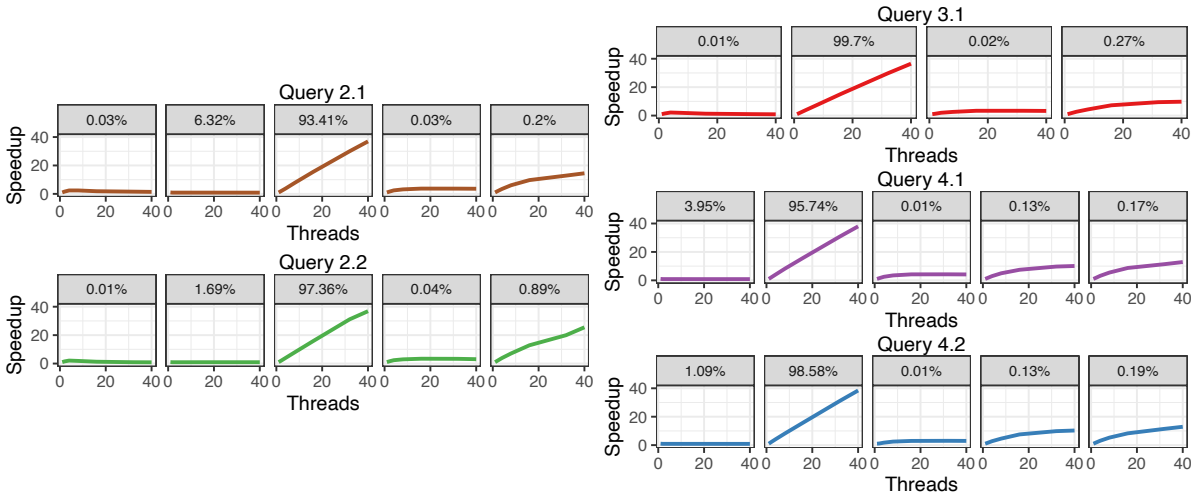

Figure 7.2: Speedup functions of each pipeline phase of various queries from the Star Schema Benchmark. Each pipeline phase was run on between 1 and 40 threads. Each phase is labeled by the percentage of time spent in the phase when the query was run on a single thread. For example, roughly 93.4% of query 2.1 was spent in an elastic phase. Hence, the overall speedup function of this query should look like Amdahl's law with parameter $f = 0.934$.

an engineering challenge, but not a research contribution. However, it is much rarer to encounter systems schedulers that are capable of changing job resource allocations in response to real-time feedback about job scalability without significant modification. We therefore demonstrate that a scheduler can implement the proposed scheduling policy, IF-Phase, while incurring minimal overhead due to preemption. Furthermore, one might worry that real-world workloads change phases too rapidly for a scheduler to effectively leverage this information to reduce response times. Hence, we also show that it is possible to leverage query phase information to reduce the mean response time across a stream of incoming database queries in a state-of-the-art database.

- We provide an implementation of a scheduler which can incorporate real-time scalability feedback in the NoisePage database[1].
- We devise scheduling policies that leverage information from the database query planner to reduce query latency.
- We show that the overhead from preempting queries based on phase changes in small compared to the benefit of making smarter, more efficient scheduling decisions, leading to an overall reduction in mean query latency.

Our preliminary work in this area consists of simulation results using real query size distributions derives from the SSB. The results of these simulations, shown in Figure 7.3, suggests that IF-Phase represents a drastic improvement over EQUI, as well as the phase-aware FCFS policy currently used in many database systems. If we make a simple tweak to have IF allocate cores to inelastic phases belonging to the queries with the shortest remaining processing times, this advantage improves even further.

Our plan is to implement our scheduler in the open-source NoisePage database, and we have begun building a prototype scheduler using this code base. Simulation numbers were based on measurements taken from NoisePage.

7.3 Impact

In previous chapters, the scheduling policies we derived were all designed with an eye towards improving state-of-the-art systems by devising simple new scheduling policies. This chapter takes a different approach,

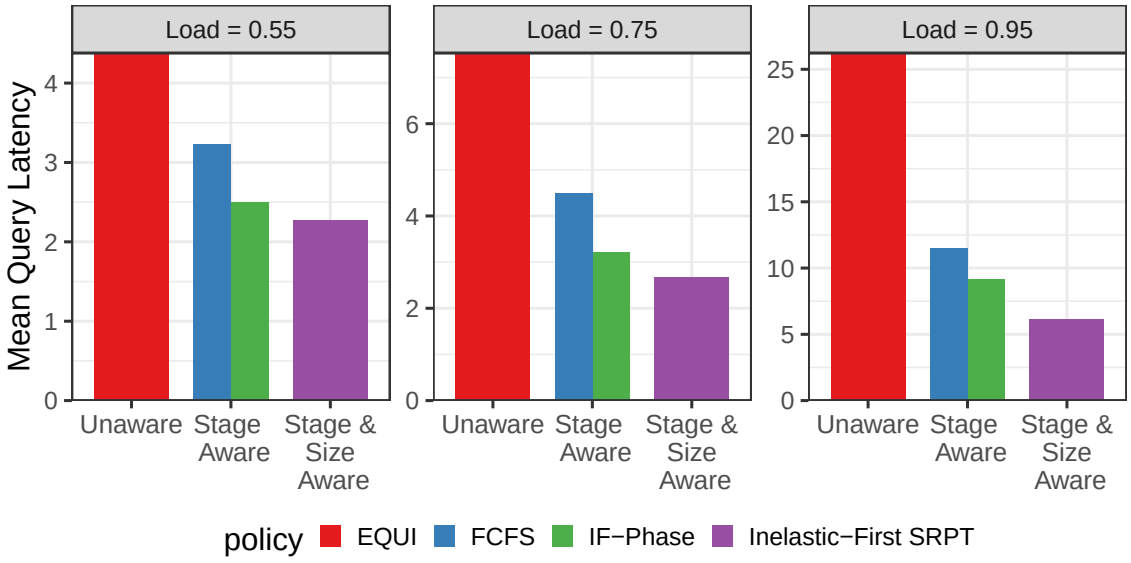

Figure 7.3: Mean response times in simulations based on the Star Schema Benchmark under various scheduling policies. EQUI, which does not take each queries current phase into account, is highly inefficient. FCFS, which is currently used in many databases, is roughly 35% worse than IF-Phase. If we allow IF-Phase to bias towards queries with shorter remaining processing times, the resulting policy, Inelastic First SRPT, is roughly 50% better than FCFS.

which is to simultaneously design the next generation of system schedulers and the scheduling policies they should use.

We first examine the potential benefit of providing additional information to the scheduler in the form of real-time feedback about instantaneous job scalability. While we are able to provide optimality results that show a significant advantage to leveraging information about a job's phases, we recognize that most systems are not currently capable of implementing this new scheduling policy. Hence, we simultaneously develop a new scheduler which can implement IF-Phase in a state-of-the-art database.

This demonstrates the power of first understanding the simpler models of parallelizable jobs presented in the previous chapters. By understanding the underlying trade-offs of scheduling parallelizable jobs in a variety of simpler scenarios first, we are able to devise new scheduling capabilities, such as considering real-time scalability feedback, that will greatly reduce job response times when implemented.

Furthermore, we have demonstrated that our models are detailed enough to accurately predict system dynamics before a real-world implementation exists. Hence, we were able in this chapter to predict the benefits of developing an improved scheduler before having to spend the time to develop the an actual implementation. This reduces the need to spend time building system prototypes which implement new features, only to find that the benefits these new features offer are minimal or require extensive hand tuning to deliver performance improvements.

While we explicitly consider database scheduling in this chapter, there are several systems which process jobs with phases that stand to benefit from this work. Distributed computing platforms such as Hadoop [43] and Apache Spark [52] explicitly encode the parallel and sequential phases of each job through their use of the MapReduce paradigm [10]. Distributed training of machine learning models is split into parallelizable iterations and sequential parameter updates [29]. Supercomputing jobs commonly consist of highly sequential startup phases followed by massively parallel computation phases, and these phases are handled explicitly by the SLURM workload manager commonly used in supercomputing centers [40]. Each of these systems has mechanisms for understanding which phase each job is in at a given moment in time, and could therefore benefit from implementing scheduling policies like IF-Phase.

Completion Percentage: 50%

Chapter 8

Parallelizable Jobs with General Size Distributions

In this chapter, we devise new methods for analyzing queueing systems with parallelizable jobs in heavy traffic. That is, we want to analyze how the mean response time of given scheduling policy compares to mean response time of the optimal policy as system load, ρ goes to 1. Here, we assume that each job's size is unknown, but drawn from a general distribution which is known to the scheduler. Furthermore, each job follows a speedup function which is known to the scheduler.

8.1 Motivation

When jobs follow more general size distributions, such as empirically derived distributions, the analysis of our systems becomes much more complex. In particular, exact analysis of these systems is as hard as analyzing an $M/G/k$ system, a notorious open queueing problem [18]. Queueing theorists have developed many techniques for analyzing these kinds of systems that do not require exactly calculating the moments of response time.

In particular, recent work has considered analyzing multi-server systems with non-parallelizable jobs under heavy-traffic, in the limit as the system load goes to 1. Unfortunately, while these techniques generalize to the case of parallelizable jobs, the results one can derive in the heavy-traffic limit are largely uninteresting. Namely, any policy which is heavy-traffic optimal when jobs are not parallelizable is also heavy traffic optimal when jobs are parallelizable. Intuitively, under heavy traffic, there are generally more than k jobs in the system. This means that k jobs can be run in parallel without any loss of efficiency due to parallelism.

This begs the question, however, of whether or not the tools of heavy-traffic analysis can be adapted to make meaningful distinctions between scheduling policies when jobs are parallelizable.

8.2 Contributions

Developing Heavy-Traffic Bounds

We develop a framework for the heavy-traffic analysis of systems with parallelizable jobs that differentiate between scheduling policies which exploit parallelism carefully and those which do so wastefully.

- We will show that tighter bounds on heavy-traffic performance can be derived for policies which more carefully exploit parallelism, such as IF or IF-Phase.
- We will show that the tightening of these bounds corresponds to improved mean response time in simulation over variety of loads, thus guiding the deployment of more effective policies for scheduling parallelizable jobs.

- We will show the sensitivity of these bounds to the overall scalability of the workload. For example, how well the system performs in heavy-traffic if jobs are, on average, more parallelizable versus less parallelizable. This will tell system designers how they can predict to benefit from investing time to increase the scalability of their jobs.

8.3 Impact

This chapter present a brand new framework for the analysis of scheduling policies for parallelizable jobs. We have consistently shown, in prior chapters, the shortcomings of a worst-case style of analysis. Then, in the prior chapters of this thesis, we considered improved stochastic models but we were forced to make certain simplifying assumptions to avoid bumping up against the known hard problem of multi-server scheduling with generally distributed job sizes. We therefore develop a new style of heavy-traffic analysis.

This new analytical framework creates many new open problems both for the scheduling of parallelizable jobs specifically, but also in the field of scheduling more generally. First, one must ask if tighter performance bounds than those presented in this chapter can be obtained for IF and IF-Phase. Next, one must consider if there are other intuitive policies for which improved heavy-traffic bounds can be obtained, such as policies which intelligently defer parallelizable work. Finally, even when jobs are not parallelizable, one must consider whether the improved heavy-traffic analysis we develop can differentiate between policies that were previously grouped together under the heading of "heavy-traffic optimality".

Completion Percentage: 25%

Chapter 9

Conclusion and Future Work

Conclusion

This thesis demonstrates how the optimal policy for scheduling parallelizable jobs depends strongly on the amount of information available to the scheduler about the size and scalability of each job. When the scheduler has no information, and cannot differentiate between large jobs and small jobs or more and less parallelizable jobs, we find that the EQUI policy, which maximizes system efficiency at every moment in time, is optimal with respect to mean response time. However, when the scheduler can differentiate between jobs which follow different speedup functions, EQUI becomes far from optimal. Here, even a GREEDY scheduling policy which maximizes system efficiency in the present state will not be optimal, since there is value to deferring parallelizable work in order to maintain the future efficiency of the system. If a job's speedup function is known to vary over time in a way that is visible to the scheduler, a GREEDY policy can suddenly appear to be highly inefficient. In this case, phase-aware scheduling policies are needed to reduce mean response time. Finally, when job sizes or size distributions are known to the scheduler, response times can be further reduced by biasing towards the jobs with shorter remaining processing times.

The results of this thesis are not solely intended to prescribe the particular scheduling policy that should be used by an existing system, but to also describe the potential benefits of building systems which provide additional information to their schedulers. For example, the developers of the NoisePage database asked us to model, based on their benchmarks, how much benefit they could expect by building a phase-aware scheduler, since this would represent a significant development effort. We were able to show using simulations derived from these benchmarks that phase-aware scheduling would significantly improve mean query latency, and was thus worth implementing. Each of the scenarios considered in this thesis represents a similar opportunity for improvement for system developers. By building schedulers with a greater awareness of job size and scalability information, our theoretical results show that we can greatly reduce response times, often by an order of magnitude.

Future Work

Although we address a wide range of settings in this thesis, the problem of how to schedule parallelizable jobs is far from solved.

The most prominent of these questions is how one should schedule jobs with more complex sequences of phases. For example, job phases could change according to a Markov process consisting of a greater number of phases with generally distributed phase sizes and speedup functions which lie in between the elastic and inelastic speedups considered in this thesis. Furthermore, the Markov process corresponding to each job could be unique. This model begins to resemble the well-studied DAG model, and thus promises to be harder to analyze. However, the generality of this model makes it well-suited to accurately modeling a wide variety of parallel computations.

Another direction to consider is to extend the analyses of this thesis to consider the tail of response time rather than the mean. System developers and administrators are often interested in the 99th percentile (P99) of response time, for example. It is interesting to ask whether policies like heSRPT which favor shorter jobs in order to reduce mean response time could actually be bad for the P99 of response time since they bias

away from longer jobs which are more likely to populate the tails of the response time distribution. The question here is whether heSRPT is effective enough in reducing queueing times to offset its bias against large jobs.

Finally, the question of how to schedule parallelizable jobs in systems composed of heterogeneous servers is becoming increasingly important. Data centers are increasingly composed of servers with different processors, different ratios of memory to CPU, and different hardware acceleration capabilities [47]. This can cause the run time of a job to depend heavily on not just how many servers it is allocated, but the type of servers it is allocated. These trends are being mirrored at the individual server level, and even on consumer-grade machines such as laptops and mobile devices[26]. This heterogeneity poses important new trade-offs not considered in this thesis. For instance if several jobs are capable of running faster on a GPU than a CPU, which job should be run on which type of hardware? Should we favor shorter jobs, maximize system efficiency, or balance load between separate CPU and GPU queues? Fully accounting for the growing heterogeneity of modern systems will require both significant theoretical work and significant systems work to make modern schedulers capable of measuring and balancing these new trade-offs.

Bibliography

[1] Noisepage. 2021. `https://noise.page`.

[2] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallel DAG jobs online to minimize average flow time. In *SIAM Symposium on Discrete Algorithms*, pages 176–189. SIAM, 2016.

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

[4] Benjamin Berg, Mor Harchol-Balter, Benjamin Moseley, Weina Wang, and Justin Whitehouse. Optimal resource allocation for elastic and inelastic jobs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 75–87, 2020.

[5] Benjamin Berg, Rein Vesilo, and Mor Harchol-Balter. heSRPT: Parallel scheduling to minimize mean slowdown. *Performance Evaluation*, 144:102–147, 2020.

[6] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM (JACM)*, 46(2):281–321, 1999.

[7] Robert D Blumofe and Charles E Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[8] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[9] 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.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[12] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 1999.

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

[14] Jeff Edmonds, Donald D Chinn, Tim Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.

[15] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the l_k-norms of flow time without conservation of work. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 109–119. SIAM, 2011.

[16] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. Srpt for multiserver systems. *Performance Evaluation*, 127:154–175, 2018.

[17] A. Gupta, B. Acun, O. Sarood, and L. Kalé. Towards realizing the potential of malleable jobs. In *HiPC*. IEEE, 2014.

[18] Varun Gupta, Mor Harchol-Balter, JG Dai, and Bert Zwart. On the inapproximability of m/g/k: why two moments of job size distribution are not enough. *Queueing Systems*, 64(1):5–48, 2010.

[19] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, 2007.

[20] Chen He, Ying Lu, and David Swanson. Matchmaking: A new mapreduce scheduling technique. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 40–47. IEEE, 2011.

[21] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[23] Esa Hyytiä, Samuli Aalto, and Aleksi Penttinen. Minimizing slowdown in heterogeneous size-aware dispatching systems. *SIGMETRICS*, 40(1):29–40, 2012.

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

[25] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 345–360, 2019.

[26] Myungsun Kim, Kibeom Kim, James R Geraci, and Seongsoo Hong. Utilization-aware load balancing for the energy efficient operation of the big. little processor. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014.

[27] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014.

[28] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, 73(6):875–891, 2007.

[29] S. Lin, M. Paolieri, C. Chou, and L. Golubchik. A model-based approach to streamlining distributed training for asynchronous SGD. In *MASCOTS*. IEEE, 2018.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[31] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, 2018.

[32] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[33] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[34] Benjamin Moseley, Kirk Pruhs, and Cliff Stein. The complexity of scheduling for p-norms of flow and stretch. In *IPCO*. Springer, 2013.

[35] Randolph Nelson and Asser N Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE transactions on computers*, 37(6):739–743, 1988.

[36] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[37] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. Adaptive htap through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2043–2054, 2020.

[38] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *ACM SIGCOMM Computer Communication Review*, 45(4):379–392, 2015.

[39] Gerald Sabin, Matthew Lang, and P Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 94–114. Springer, 2006.

[40] SchedMD. SLURM workload manager. 2021. `https://slurm.schedmd.com/heterogeneous_jobs.html`.

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[42] Manik Sharma, Gurvinder Singh, and Rajinder Singh. A review of different cost-based distributed query optimizers. *Progress in Artificial Intelligence*, 8(1):45–62, 2019.

[43] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10, 2010.

[44] Donald R Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978.

[45] 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.

[46] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[47] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EUROSYS*. ACM, 2015.

[48] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506, 2010.

[49] Weina Wang, Mor Harchol-Balter, Haotian Jiang, Alan Scheller-Wolf, and Rayadurgam Srikant. Delay asymptotics and bounds for multitask parallel jobs. *Queueing Systems*, 91(3):207–239, 2019.

[50] Adam Wierman, Mor Harchol-Balter, and Takayuki Osogami. Nearly insensitive bounds on SMART scheduling. *SIGMETRICS*, 33(1):205–216, 2005.

[51] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[52] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

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

[54] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1225–1240, 2020.