PERFORMANCE MODELING AND RESOURCE PROVISIONING FOR DATA-INTENSIVE APPLICATIONS

by

ZHONGWEI LI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy at
The University of Texas at Arlington
Doctor of Philosophy
December, 2019

Arlington, Texas

Supervising Committee:

Prof. Hao Che, Supervising Professor
Prof. Jeff (Yu) Lei, Co-supervising Professor
Prof. Christoph Csallner
Prof. Hong Jiang

ABSTRACT

PERFORMANCE MODELING AND RESOURCE PROVISIONING FOR DATA-INTENSIVE APPLICATIONS

Zhongwei Li, Ph.D.

The University of Texas at Arlington, 2019

Supervised by: Dr. Hao Che & Dr. Jeff Lei

Performance evaluation and resource provisioning are two most critical factors to be considered for designers of distributed systems at modern warehouse data centers. The ever-increasing volumes of data in recent years have pushed many businesses to move their computing tasks to the Cloud, which offers many benefits including the low system management and maintenance costs and better scalability. As a result, most recent prominently emerging workloads are data-intensive, calling for scaling out the workload to a large number of servers for parallel processing. Questions can be asked as what factors impact the system scaling performance, and how to efficiently schedule tasks to the distributed comping resources. This dissertation introduces a new performance model to address the former problem and an effective hierarchical job scheduler for the latter.

The major contribution of this dissertation is to introduce our new performance modeling approach designed for data-intensive applications, which consists of two phases: 1) In-Proportion and Scale-Out-induced scaling model (IPSO), 2) Unified Scaling model for Big data Analytics (USBA). The first model we build is based on the traditional performance models including both Amdahl's and Gustafson's laws. We clearly demonstrate in this research why these classic models are insufficient and inadequate in today's parallel computing environment and how IPSO model may fill the gap. While at the second phase we extend IPSO for today's multi-staged workloads, such model can be easily adopted at modeling data analytic applications running at Spark platform. Both models are supported by our evaluations on well-known benchmarks and evidences from other publications. To the best of our knowledge, IPSO is the first variation of the classic Amdahl's model that can be directly applied to modern data-intensive applications. A light-weighted tool is also developed at the end of this research, which can be used for generating IPSO inputs or a Spark application log analyzer. The tool is developed as an open source project and accessible in public repository.

The second contribution of this dissertation is the Pigeon job scheduler we propose for the modern data centers. Pigeon is a distributed, hierarchical job scheduler based on a two-layer design. It offloads the service pressure in widely adopted centralized data center scheduler by quickly dispatching the incoming tasks to selected nodes known as masters, then guarantees the efficiency of task execution by enforcing its unique queuing mechanism on these masters. Pigeon can minimize the chance of head-of-line blocking for short jobs and avoid starvation for long jobs, and outperform Sparrow (distributed scheduler) and Eagle (hybrid scheduler) based on our evaluations. Pigeon is also an open sourced tool that can be accessed from public repository.

This dissertation is presented in an article-based format and includes three research papers. The first chapter is an introduction to all contents in this dissertation. The second chapter reports our performance evaluation model (IPSO). The third chapter reports IPSO's extended model for multi-staged workloads (USBA). The fourth chapter reports our work on Pigeon scheduler. Finally the fifth concludes all work and the plan for the following research target.

ACKNOWLEDGMENTS

I would like to thank both of my supervising professors Dr. Hao Che and Dr. Jeff Lei for their continuous guidance. This work would not have been possible without their wisdom, enthusiasm, encouragement, patience and support. I wish to also thank my committee members, Dr. Hong Jiang and Dr. Christoph Csallner, for generously sharing their time and ideas.

I would also like to extend my appreciation to my parents for their constant love and support. To my friends for their support, interest and advice. And to my roommate Feng, who always supports me like brother and helps me out. Finally special thanks to my ex-girlfriends for leaving me so I still remain single, focused and rational.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Performance modeling is a major technique for system architecture designer and maintainers to understand the behavior of their running system and thus is widely studied by the scientists. Ever since Gene M. Amdahl published his initial work on the performance modeling for parallel computing architectures at 1960's, along with the evolution of hardware and software gradually, a variety of descendant models based on the Amdahl's model have been brought up by researchers to meet the requirements of the new HPC (High Performance Computing) environment for each generation. Recent years have witnessed the emergence of data-intensive applications as the new era of Cloud Computing rises, however the empirical studies indicate the insufficiency of traditional performance models. The key observation is how to abstract the modern parallel computing architectures into certain level that the trivial execution details can be neglected, yet enough accuracy in guiding the resource provisioning at scaling out can still be retained. In this dissertation, we introduce IPSO and USBA models as our approaches to achieve such goal.

On the other hand, resource provisioning is yet another topic that has received significant attention in recent decades. In today's datacenters, one key observation is that job heterogeneity makes it difficult for schedulers to simultaneously meet latency requirements and maintain high resource utilization. And the state-of-the-art datacenter schedulers, including centralized, distributed, and hybrid schedulers, fail to ensure low latency for short jobs in large-scale and highly loaded systems. We identify that the scalability in centralized schedulers, ineffective and inefficient probing and resource sharing in both distributed and hybrid schedulers are the key issues. Targeting resolving such problems, we propose Pigeon, a distributed, hierarchical job scheduler based on a two-layer design. Pigeon is also an open source project under MIT license, a free distribution of its source code can be obtained through the linked address to our public repository[1].

1.1 Research Overview

IPSO (In-Proportion and Scale-Out-induced scaling model) is our first performance model that generalizes the existing scaling models in two important aspects. First, it accounts for the possible *in-proportion scaling*, i.e., the scaling of the serial portion of the workload in proportion to the scaling of the parallelizable portion of the workload. Second, it takes into account the possible *scale-out-induced scaling*, i.e., the scaling of the collective overhead or workload induced by scaling out. IPSO exposes scaling properties of data-intensive workloads, rendering the existing scaling laws its special cases. In particular, IPSO reveals two new pathological scaling properties. Namely, the speedup may level off even in the case of the fixed-time workload underlying Gustafson's law, and it may peak and then fall as the system scales out. Extensive MapReduce and Spark-based case studies demonstrate that IPSO successfully captures diverse scaling properties of data-intensive applications. As a

[1] Pigeon available at `https://github.com/ruby-/pigeon`

result, it can serve as a diagnostic tool to gain insights on or even uncover counter-intuitive root causes of observed scaling behaviors, especially pathological ones, for data-intensive applications. Finally, preliminary results also demonstrate the promising prospects of IPSO to facilitate effective resource provisioning to achieve the best speedup-versus-cost tradeoffs for data-intensive applications.

Unified Scaling model for Big data Analytics (USBA) is the second model we propose based on IPSO that addresses two additional features: 1) it accounts for multi-stage workloads; 2) it's also a discretized workload model. USBA allows for flexible workload scaling unifying the fixed-size and fixed-time workload models underlying Amdahl's and Gustafson's laws, respectively, and flexible system scaling in terms of both number of stages and degree of parallelism per stage. Moreover, to faithfully characterize the scaling properties for big data analytics workloads, USBA accounts for variabilities of task response times and barrier synchronization. Finally, application of USBA to the scaling analysis of four Spark-based data mining and graph benchmarks demonstrates that USBA is able to adequately characterize the scaling design space and predict the scaling properties of real-world big data analytics workloads. This makes it possible to use USBA as a useful tool to facilitate job resource provisioning for big data analytics in datacenters.

An another important contribution of this dissertation is the design and implementation of Pigeon scheduler. Pigeon divides workers in a distributed computing environment into groups, each managed by a separate master. In Pigeon, upon a job arrival, a distributed scheduler directly distribute tasks evenly among masters with minimum job processing overhead, hence, preserving highest possible scalability. Meanwhile, each master manages and distributes all the received tasks centrally, oblivious of the job context, allowing for full sharing of the worker pool at the group level to maximize multiplexing gain. To minimize the chance of head-of-line blocking for short jobs and avoid starvation for long jobs, two weighted fair queues are employed in in each master to accommodate tasks from short and long jobs, separately, and a small portion of the workers are reserved for short jobs. Evaluation via theoretical analysis, trace-driven simulations, and a prototype implementation shows that Pigeon significantly outperforms Sparrow, a representative distributed scheduler, and Eagle, a hybrid scheduler.

1.2 Dissertation organization

This dissertation is presented in an article-based format and includes the essential components of three research papers, hence the remainder of this dissertation is organized as follows. In chapter 2, we present the paper titled, "IPSO: A Scaling Model for Data-Intensive Applications", which is published in the 39th International Conference on Distributed Computing Systems (ICDCS'19), in 2019. This paper reports our first performance model for distributed systems. Note that this paper is titled as a co-authored work which also includes the following authors besides me: Feng Duan, Minh Nguyen, Hao Che, Yu Lei and Hong Jiang. I'm the primary author to this paper and take the lead of this project; Hao Che, Yu Lei and Hong Jiang are the project's supervisors; and all the rest co-authors contribute to the creation of this model and constructions of the paper. I also contribute to most of the experiments study and analysis results.

Chapter 3 presents our paper titled as "A Unified Scaling Model in the Era of Big Data Analytics", which is published in the proceedings of the 3rd International Conference on

High Performance Compilation, Computing and Communications. ACM, 2019. This paper presents the second phase of our work on performance modeling and our initial attempt on resource provisioning based on this prediction model. Note that this paper is titled as a co-authored work which also includes the following authors besides me: Feng Duan and Hao Che. I'm the primary author and project leader, and contribute most of the data collection and analysis parts in the paper, all the rest co-authors contribute to the modeling and writing. I'm also the leader of this open source project and maintain our public code repository.

Chapter 4 presents a paper titled as "Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler", which is published in ACM Symposium on Cloud Computing 2019 (SoCC '19). This paper presents a new hierarchical scheduler for datacenters, and as an additional to the original paper, I also provide some design and implementation details of the Pigeon open source project in this chapter. Note that this paper is titled as a co-authored work which also includes the following authors besides me: Zhijun Wang, Huiyang Li, Hao Che, Hong Jiang, Jia Rao and Xiaocui Sun. Zhijun Wang, Hao Che and Hong Jiang lead this project; Zhijun Wang provides most of the simulation part and analysis results; Huiyang Li conducts most of our experiments; Jiao Rao contributes to most of the introduction section of the paper. I am the leader of our Pigeon open source project and contribute to the majority of implementation code. All the rest co-authors also contribute to the work and help review the paper.

Finally Chapter 5 concludes and remarks all of our previous works then discusses the directions to our future research.

IPSO: A Scaling Model for Data-Intensive Applications

Li, Zhongwei, Feng Duan, Minh Nguyen, Hao Che, Yu Lei, and Hong Jiang. "IPSO: A Scaling Model for Data-Intensive Applications." In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 238-248. IEEE, 2019.

Chapter 2

IPSO MODEL

2.1 Introduction

Predominant applications in today's datacenters are data-intensive and scale-out by design, based on, e.g., MapReduce [18], Spark [64], and Dryad [33] programming frameworks. For such applications, job execution may involve one or multiple rounds of parallel task processing with massive numbers of tasks and the associated data shards being scaled out to up to tens of thousands of low-cost commodity servers, followed by a serial (intermediate) result merging process. Clearly, from both user's and datacenter provider's perspective, it is imperative to gain good understanding of the scaling properties of such applications so that informed datacenter resource provisioning decisions can be made to achieve the best speedup-versus-cost tradeoffs. Unfortunately, however, the existing scaling laws that have worked well for parallel, high-performance computing, such as Amdahl's law [5], Gustafson's law [27], and Sun-Ni's law [55], are no longer adequate to characterize the scaling properties of data-intensive workloads for two reasons.

First and foremost, the traditional scaling models underlying these laws are exclusively focused on the scaling of the parallelizable portion of the workload or *external scaling* (e.g., the fixed-size, fixed-time, and memory-bounded external scaling models underlying Amdahl's, Gustafson's, and Sun-Ni's laws, respectively), leaving the scaling of the serial portion of the workload or *internal scaling* a constant. Fig. 2.1 illustrates this, i.e., scaling out to three parallel processing units for the Amdahl's model in Fig. 1(b) and Gustafson's or Sun-Ni's model in Fig. 1(c) from the sequential execution case in Fig. 1(a). While the parallelizable portion of the workload stays unchanged (i.e., fixed-size) and grows by three times (i.e., fixed-time or memory-bounded), respectively, the serial portion of the workload remains unchanged. The rationale behind this assumption is the understanding that the serial portion of a program mostly occurs in the initialization phase of the program, which is independent of the program size [27]. This assumption, however, no longer holds true for data-intensive workloads. This is because as the parallelizable portion of a data-intensive workload increases, so does the serial portion of the workload in general. In other words, the (intermediate) results to be merged in each round of the job execution are likely to grow, in proportion to the external scaling, referred to as *in-proportion scaling* in this paper.

Second, the existing scaling models do not take the possible *scale-out-induced scaling* into account, i.e., the scaling of the collective overhead or workload induced by the external scaling. As being widely recognized (see Section 2 for details), for data-intensive applications, such workloads cannot be neglected in general and they may be induced for various reasons, e.g., task dispatching, data broadcasting, reduction operation, or any types of resource contentions among parallel tasks. Both in-proportion scaling and scale-out-induced scaling are responsible for the scalability challenges facing today's programming frameworks, such as Hadoop and Spark [45].

To overcome the above inadequacies of the existing scaling models, in this paper, we put

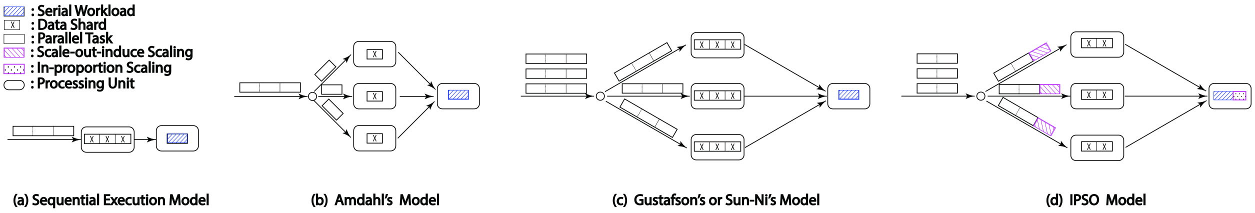


Figure 2.1: Speedup models: For data-intensive applications, Sun-Ni's model coincides with Gustafson's model (see Section 2.4 for details).

forward a new scaling model, referred to as In-Proportion and Scale-Out-induced scaling model (IPSO). IPSO augments the traditional scaling models with the in-proportion scaling and scale-out-induced scaling, as illustrated in Fig. 1(d) (note that the shard size can be one, two, or three), rendering the traditional scaling models and their respective scaling laws its special cases. In particular, IPSO reveals two new pathological scaling properties that are not captured by the existing scaling laws. Namely, the speedup may level off even in the case of the fixed-time workload, and it may peak and then fall as the system scales out, for which Gustafson's law says that the speedup should be unbounded. While the former scaling property is due to the in-proportion scaling, the latter may be attributed to either in-proportion scaling or scale-out-induced scaling. Moreover, the scale-out-induced scaling, in the worst case, may lead to negative speedups, which cannot be captured by the existing scaling laws.

Our extensive case studies for both MapReduce and Spark-based applications demonstrate that while the existing scaling laws fail to capture most of the scaling properties for these applications, IPSO is able to do so for all the cases studied. As a result, IPSO can serve as a diagnostic tool that can gain insights or even uncover counter-intuitive root causes of the observed scaling behaviors, especially, pathological ones, for data-intensive applications.

Finally, our preliminary results suggest that as long as the three scaling factors, including the external, internal, and scale-out-induced scaling factors, can be accurately estimated at small problem sizes, the speedups at large problem sizes may be predicted with high accuracy. This sheds light on the possible development of efficient, measurement-based resource provisioning algorithms to achieve the best speedup-versus-cost tradeoffs for data-intensive workloads.

The remainder of the paper is organized as follows. Section 2.2 provides the background information to motivate the current work. Section 2.3 introduces IPSO. Section 2.4 characterizes the IPSO solution space. Section 2.5 presents the application of IPSO to the MapReduce and Spark-based case studies. Finally, Section 2.6 concludes the paper and proposes future research.

2.2 Background, Related Work and Motivations

The traditional scaling laws for parallel computing were discovered in the context of high performance computing. Amdahl's law [5], Gustafson's law [27] and Sun-Ni's law [55] are the most notable examples of such laws. Recently, extensions of these laws are being proposed, e.g., in the context of multicore processors [28], multithreaded multicore processors [9], power consumption [62], and resource scaling in cloud [51]. However, none of these extensions takes the possible in-proportion scaling or scale-out-induced scaling into account.

Meanwhile, with the advent and proliferation of scale-out, data-intensive applications, rich scaling properties for such applications continue to reveal themselves, most of which, however, cannot be adequately characterized by the existing scaling laws. Here are some examples. It was found [14] that for a fixed-size iterative computing and broadcast scale-out Spark-based workload, the job stops scaling at about $n = 60$, beyond which the speedup decreases due to linear increase of the broadcast overhead, where n is the number of computing nodes for parallel processing. TCP-incast overhead was found to be responsible for the speedup reduction for many big data analytics applications [13]. Centralized job schedulers used in some popular programming frameworks, such as Hadoop and Spark, were found to pose performance bottlenecks for job scaling, due to a quadratic increase of the task scheduling rate as n increases [45]. In fact, a queuing-network-model-based analysis [9] reveals that any resource contention among parallel tasks is guaranteed to induce an effective serial workload, resulting in lower speedup than that predicted by the existing laws.

The scaling analysis of data mining applications [39] reveals that the reduction operations in each merging phase are induced by external scaling, resulting in much lower speedup than that predicted by Amdahl's law. As we shall demonstrate in Section 2.5, even for some simple MapReduce-based applications, including Sort and TeraSort, their scaling properties cannot be captured by the existing scaling laws, largely due to the in-proportion scaling. The Spark-based case studies in Section 2.5 further reveal that parallel scaling in both fixed-time and fixed-size dimensions, underlying Gustafson's and Amdahl's laws, respectively, exhibit scaling behaviors that significantly deviate from those predicted by these scaling laws.

The above examples clearly demonstrate the inadequacy of the existing scaling laws in capturing the scaling properties of data-intensive applications. The importance and the urgency of the ability to do so cannot be overemphasized for two main reasons. First, the existing scaling laws may lead to overly optimistic prediction of the scaling performance for data-intensive workloads. They may even make qualitatively incorrect prediction when a pathological situation occurs (see Section 2.5 for examples). In our opinion, the lack of a sound scaling model is largely responsible for the unsettled debate over whether scaling-out is indeed better than scaling-up or not [40]. Second, as the existing scaling laws are increasingly being adopted to not only characterize the scaling properties, but also facilitate resource provisioning for data-intensive workloads [68], it becomes urgent to develop a comprehensive scaling model that can help pinpoint the exact conditions under which the existing scaling laws may be applied.

The importance and the urgency to develop a comprehensive scaling model for data-intensive applications motivate the work presented in this paper.

2.3 IPSO Modeling

First, we must realize that the main goal of scaling analysis for parallel computing is to capture the scaling properties of the speedup for parallel computing over sequential computing, when the problem size becomes large. A scaling model is considered to be a good one, as long as it captures in a qualitative fashion (e.g., bounded or unbounded, linear or non-linear, monotonic or peaked) major scaling properties of the applications in question. Due to the need to deal with large problem sizes and the tolerability of quantitative imprecision, idealized scaling models that overlook much of the system and workload details are generally adopted, targeting at analytical results that can scale to large problem sizes. The IPSO model is depicted in Fig. 1, together with the Amdahl's, Gustafson's and Sun-Ni's models. Scaling modeling generally involves the modeling of both the system and workload.

System Model: In the same spirit as the existing scaling models, IPSO adopts the same idealized system model that underlies all three existing scaling laws, i.e., Amdahl's, Gustafson's and Sun-Ni's laws. This system model, in the context of data-intensive applications, can be viewed as a homogeneous Split-Merge model with $n + 1$ identical processing units [9], as illustrated in Fig. 2.1, with $n = 3$. There are n processing units in the split phase, processing the parallelizable portion of the workload in parallel and one processing unit in the merge phase, processing the serial portion of the workload sequentially.

Specifically, the Split-Merge model characterizes the execution of a job composed of one round of parallel task processing with barrier synchronization in the split phase, followed by sequential result merging in the merge phase. Here n is a measure of the degree of scale-out and hence, called *scale-out degree* hereafter. This model can also be applied to the case where there are multiple rounds of the split and merge phases with the same number of processing units in each split phase.

Workload Model: The main effort in developing IPSO is the modeling of the workload. IPSO generalizes and augments the workload models underlying the three speedup laws, hence making them the special cases of IPSO.

For data-intensive applications, the offered workload at each parallel processing unit is proportional to the data shard size at that unit. As a result, as n increases, the total data shard remains to be n (i.e., three as in Fig. 1(b)) and increases by n times (i.e., 9 as in Fig. 1(c)) for the fixed-size and fixed-time/memory-bounded cases, respectively. The IPSO model allows the fixed-size, fixed-time, or anywhere in between as the scale-out degree increases, e.g., doubling the total shard size for the example in Fig. 1(d). In general, the task processing time for the task mapped to processing unit i in the split phase is a random variable, denoted as $T_{p,i}(n)$, serving as a measure of the workload corresponding to task i, for $i = 1, 2, \cdots, n$. As a result, $T_{p,i}(n)$ may grow in (linear) proportion to the size of the data shard mapped to the processing unit i. The processing time for serial result merging in the merge phase is again a random variable, denoted as $T_s(n)$, which, for data-intensive applications, may grow in proportion to the size of the total working data set or total shard size mapped to the split phase, as shown in Fig. 1(d), whereas its counterparts in Fig. 1(b) and (c) stay unchanged. Now, let $W_p(n)$ and $W_s(n)$ represent the total parallelizable and serial portions of the job workload, respectively, and define,

$$W_p(n) = \mathbb{E}[\sum_{i=1}^{n} T_{p,i}(n)] \tag{2.1}$$

$$W_s(n) = \mathbb{E}[T_s(n)] \tag{2.2}$$

where $\mathbb{E}[x]$ represents the mean of random variable x. Here $W_p(n)$ and $W_s(n)$ should be interpreted as the average amount of time it takes to process the parallelizable and serial portions of the job workload sequentially using one processing unit[1]. Further define,

$$W_p(n) = W_p(1) \cdot EX(n) \tag{2.3}$$

$$W_s(n) = W_s(1) \cdot IN(n) \tag{2.4}$$

where $EX(n)$ and $IN(n)$ are called *external* and *internal* scaling factors, corresponding to the scaling of the parallelizable and serial portions of the workload, respectively. These scaling factors enable in-proportion scaling. We further define in-proportion scaling ratio,

[1] Note that by definition, the sequential job execution does not generate scale-out-induced workload, hence $W_o(n)$ does not appear in the numerator.

$\epsilon(n)$, as follows,

$$\epsilon(n) = \frac{EX(n)}{IN(n)} \tag{2.5}$$

As we shall see shortly, a rich set of scaling properties can be uncovered by properly selecting this ratio.

Now we further introduce the scale-out-induced workload shown in Fig. 1(d) and denote it as $W_o(n)$. $W_o(n)$ represents the collective overhead induced by the scale-out itself, e.g., due to job scheduling, data shard distribution, and the queuing effect for result merging. We define,

$$W_o(n) = \frac{W_p(n)}{n} q(n) \tag{2.6}$$

where $q(n)$ is called *scale-out-induced scaling factor*, which is a non-decreasing function of n and equals zero at $n = 1$. It captures the effective workload induced solely by the scale-out degree n, independent of the task workload size. In contrast, its coefficient, $\frac{W_p(n)}{n}$, i.e., the per-task workload, captures the possible dependency of $W_o(n)$ on the task workload size. For example, the data shard distribution overhead grows with both n and the task workload size or data shard size.

Finally, with the barrier synchronization and randomness of parallel task processing times, the mean job response time with respect to parallel task processing is given by the slowest task, i.e., $\mathbb{E}[\max\{T_{p,i}(n)\}]$.

With the above scaling model, the speedup, $S(n)$, can then be expressed as follows,

$$S(n) = \frac{W_p(n) + W_s(n)}{\mathbb{E}[\max\{T_{p,i}(n)\}] + W_s(n) + W_o(n)} \tag{2.7}$$

While the numerator is the average amount of time it takes to process the entire job workload sequentially using one processing unit, the denominator is the average amount of time it takes to process the entire job workload in parallel with n processing units, plus the workload due to scale-out-induced scaling. Substituting Eqs. (2.1)-(2.6) into Eq. (2.7), we have,

$$S(n) = \frac{\eta EX(n) + (1-\eta)IN(n)}{\frac{\mathbb{E}[\max\{T_{p,i}(n)\}]}{\mathbb{E}[T_{p,1}(1)]+\mathbb{E}[T_s(1)]} + (1-\eta)IN(n) + \frac{\eta EX(n)q(n)}{n}} \tag{2.8}$$

where η is the percentage of the parallelizable portion of the job workload at $n = 1$, i.e.,

$$\eta = \frac{W_p(1)}{W_p(1) + W_s(1)} \equiv \frac{\mathbb{E}[T_{p,1}(1)]}{\mathbb{E}[T_{p,1}(1)] + \mathbb{E}[T_s(1)]} \tag{2.9}$$

An executable, sequential job execution model must be defined to allow the numerator in Eq. (2.7) or (2.8) to be measurable in practice. It will be given in Section 2.4, after the workload types are defined (i.e., Eq. (2.13)).

2.4 IPSO Solution Space Characterization

The IPSO workload model developed above is a statistic model that accounts for the possible randomness of the task execution times. The statistic modeling is important in practice if the scaling analysis attempts to capture the scaling properties of an application both qualitatively and quantitatively. For example, to capture the impact of long-tail effects of task service time on the speedup performance, e.g., due to stragglers [67] or the possible task queuing effects [46], the mean job response time for the split phase must be characterized statistically by $\mathbb{E}[\max\{T_{p,i}(n)\}]$ (see Eq. (2.8)). However, since $\mathbb{E}[\max\{T_{p,i}(n)\}]$ is upper bounded as the problem size in terms of n becomes large, given that the tail length of the task response time must be finite in practice, whether to use statistic or deterministic modeling will not make a difference in terms of capturing the qualitative scaling properties of an application. The reason that we formulate IPSO as a statistic model is to allow accurate scaling prediction that may serve as the basis for future development of a measurement-based job resource provisioning approach for data-intensive applications. So in the rest of the paper, we shall focus on the deterministic model only for simplicity and ease of presentation. The deterministic IPSO refers to a special case where $T_{p,i}(n) = t_p(n)\ \forall\ i$ and $T_s(n) = t_s(n)$. Here $t_p(n)$ and $t_s(n)$ are deterministic functions of n. In this case, $\mathbb{E}[\max\{T_{p,i}(n)\}] = t_p(n)$. Hence, from Eq. (2.8), we have,

$$S(n) = \frac{\eta EX(n) + (1-\eta)IN(n)}{\frac{\eta EX(n)}{n}(1+q(n)) + (1-\eta)IN(n)} \tag{2.10}$$

where η in Eq. (2.9) can be rewritten as,

$$\eta = \frac{t_p(1)}{t_p(1) + t_s(1)} \tag{2.11}$$

Clearly, by viewing $W_p(n)$, $W_s(n)$ and $W_o(n)$ as the sum of the corresponding workloads in all rounds, the above IPSO model can be applied to the case involving multiple rounds of the same scale-out degree, n.

Relation to the well-known speedup laws: With the notations defined in this paper, the three well-known speedup laws can be written as,

$$S(n) = \begin{cases} \frac{1}{\frac{\eta}{n}+(1-\eta)}\ , & \text{Amdahl's law;} \\ \eta n + (1-\eta)\ , & \text{Gustafson's law;} \\ \frac{\eta\bar{g}(n)+(1-\eta)}{\frac{\eta\bar{g}(n)}{n}+(1-\eta)}\ , & \text{Sun-Ni's law.} \end{cases} \tag{2.12}$$

The scaling properties for these laws can be derived from Eq. (2.10), by letting $IN(n) = 1$ and $q(n) = 0$, $\forall\ n$, i.e., without considering the possible in-proportion scaling and scale-out-induced scaling, and,

$$EX(n) = \begin{cases} 1\ , & \text{fixed-size: Amdahl's law;} \\ n\ , & \text{fixed-time: Gustafson's law;} \\ \bar{g}(n)\ , & \text{memory-bounded: Sun-Ni's law.} \end{cases} \tag{2.13}$$

meaning that the total parallelizable portion of the workload stays unchanged for fixed-size workload; linearly increases for fixed-time workload; and scales with the memory size for memory-bounded workload, respectively, as the system scales out or n increases. Here $\bar{g}(n)$ is the external scaling factor, constrained by the total memory space, which in turn, is determined by n, assuming that the maximum affordable memory space to accommodate part of the working data set at each parallel processing unit is a given, e.g., 128 MB [55]. For all the cases studied in this paper where the working data sets are memory bounded, $\bar{g}(n) \approx n$ with high precision (see Fig. 2.6), i.e., almost the same as that for the fixed-time workload. For this reason, we assume that the Gustafson's and Sun-Ni's models are the same (see Fig. 1(c)) in the context of data-intensive applications, and in what follows, we exclusively focus on fixed-size and fixed-time workload types only.

A Remark: We observe that in the context of data-intensive workloads, the fixed-size and fixed-time workload models capture two extreme scenarios, i.e., resource-abundant and resource-constrained, respectively. By resource-abundant, we mean that the parallelizable portion of the workload can be processed in its entirety by one processing unit. In this case, one is interested in characterizing the scaling behaviors when the fraction of the parallelizable workload on each processing unit decreases as the scale-out degree, n, increases, i.e., the Amdahl's case. By resource-constrained, we mean that each processing unit can only handle a fraction of the total parallelizable portion of the workload, e.g., the case when the memory allocated to each processing unit is fully occupied by the data shard assigned to it. In this scenario, the workload linearly grows with n, i.e., the Gustafson's case. In general, however, as the system scales out, a data-intensive workload may scale in either way or anywhere in between. As a result, a comprehensive scaling model should be able to cover both fixed-time and fixed-size workload types, as is the case for IPSO.

Finally, with the workload types defined in Eq. (2.13), now we are in a position to define an executable sequential job execution model underlying the numerator in Eq. (2.7) or (2.8). For fixed-time workload, our sequential job execution model works as follows. It first runs n tasks in the split phase sequentially using one processing unit. It then merges task results in the merging phase using another processing unit. Since the merging phase may not start until all the tasks finish, due to barrier synchronization, this model is equivalent to using only one processing unit to execute the job sequentially, which agrees with the common understanding of what a sequential execution model is supposed to be. For fixed-size workload, the same sequential job execution model applies. The only difference is that now $n = 1$, i.e., only one task is executed in the map phase over the entire working data set that is assumed to fit into the memory in a single processing unit. This model is in line with the sequential job execution model, implicitly used by Amdahl to evaluate the numerator in the speedup formula, i.e., the entire workload is executed as one task using one processing unit.

Analysis of scaling properties of IPSO: We are interested in exploring major scaling properties of IPSO in the entire solution space spanned in three dimensions, including $EX(n)$, $IN(n)$, and $q(n)$. In other words, the scaling behaviors of IPSO are fully captured so long as these factors are known. As explained before, for scaling analysis, one is interested in the qualitative scaling behaviors of the speedup when n becomes large. In this case, $\epsilon(n)$ in Eq. (2.5) can be written approximately as (i.e., only the highest order term is kept),

$$\epsilon(n) \approx \alpha n^{\delta} \qquad \text{as } n \text{ becomes large.} \tag{2.14}$$

where α is a nonnegative coefficient and δ determines the relative order of "speed" of external scaling versus internal scaling. Likewise, $q(n)$ can be approximated as follows,

$$q(n) \approx \beta n^{\gamma} \qquad \text{as } n \text{ becomes large.} \tag{2.15}$$

where β is a nonnegative coefficient and $\gamma \geq 0$. Here $\gamma = 0$ corresponds to the case without scale-out-induced workload, i.e., $q(n) = 0$.

With Eqs. (2.14), (2.15) and (2.5), Eq. (2.10) can be rewritten as follows,

$$S(n) \approx \frac{\eta \alpha n^{\delta} + (1-\eta)}{\eta \alpha n^{\delta-1}(1+\beta n^{\gamma}) + (1-\eta)} \tag{2.16}$$

Note that for the workload without a serial portion, i.e., $W_s(n) = 0$ or $\eta = 1$, Eqs. (2.14) and (2.5) are undefined. In this case, from Eq. (2.10), we have,

$$S(n) = \frac{n}{1+\beta n^{\gamma}} \tag{2.17}$$

With these two formulas, we are now ready to explore the entire IPSO solution space. We consider fixed-time and fixed-size workload types, separately.

Fixed-time workload type ($EX(n) = n$): In this case, $0 \leq \delta \leq 1$. This is because in practice, as the parallel portion of the workload scales up linearly fast, $IN(n)$ is unlikely to scale down or scale up superlinearly fast. From Eqs. (2.16) and (2.17), we identify the following four distinct types of speedup scaling behaviors, as depicted in Fig. 2.2:

- I_t: This type is Gustafson-like, i.e., the speedup linearly grows and degenerates to Gustafson's law at $\alpha = 1$. As shown in Fig. 2.2, it occurs when there is no scale-out-induced workload (i.e., $\gamma = 0$, or equivalently, $q(n) = 0$) and either $\delta = 1$ (i.e., with no internal scaling) or in the absence of the serial workload (i.e., $\eta = 1$);

- II_t: Speedup grows sublinearly but still unbounded. It occurs when $q(n)$ grows slower than linear, i.e., $\gamma < 1$, and either $0 < \delta \leq 1$ or $\eta = 1$;

- III_t : This type is pathological, i.e., the speedup grows monotonically but is upper-bounded. There are two sub-types here, i.e., $III_{t,1}$ and $III_{t,2}$, with distinct upper bounds, as depicted in Fig. 2.2, corresponding to sublinear and linear scale-out scalings, respectively;

- IV_t: This type is even more pathological as the speedup peaks and falls, and finally enters negative speedup region. It occurs when $q(n)$ scales up superlinearly fast, i.e., $\gamma > 1$, regardless of how the other scaling factors behave.

Fixed-size workload type ($EX(n) = 1$): In this case, $\delta = 0$. This is because without scaling the parallel portion of the workload, the serial portion of the workload will not scale, i.e., $IN(n) = 1$, and any workload added as n increases should be viewed as part of $W_o(n)$, i.e., scale-out-induced workload. Again, from Eqs. (2.16) and (2.17), four distinct types of speedup scaling behaviors are identified, as depicted in Fig. 2.3 (note that although they look the same as their counterparts in Fig. 2.2, the associated scaling factors are different):

- I_s: $S(n) = n$. it occurs when there is no scale-out-induced workload (i.e., $\gamma = 0$) and

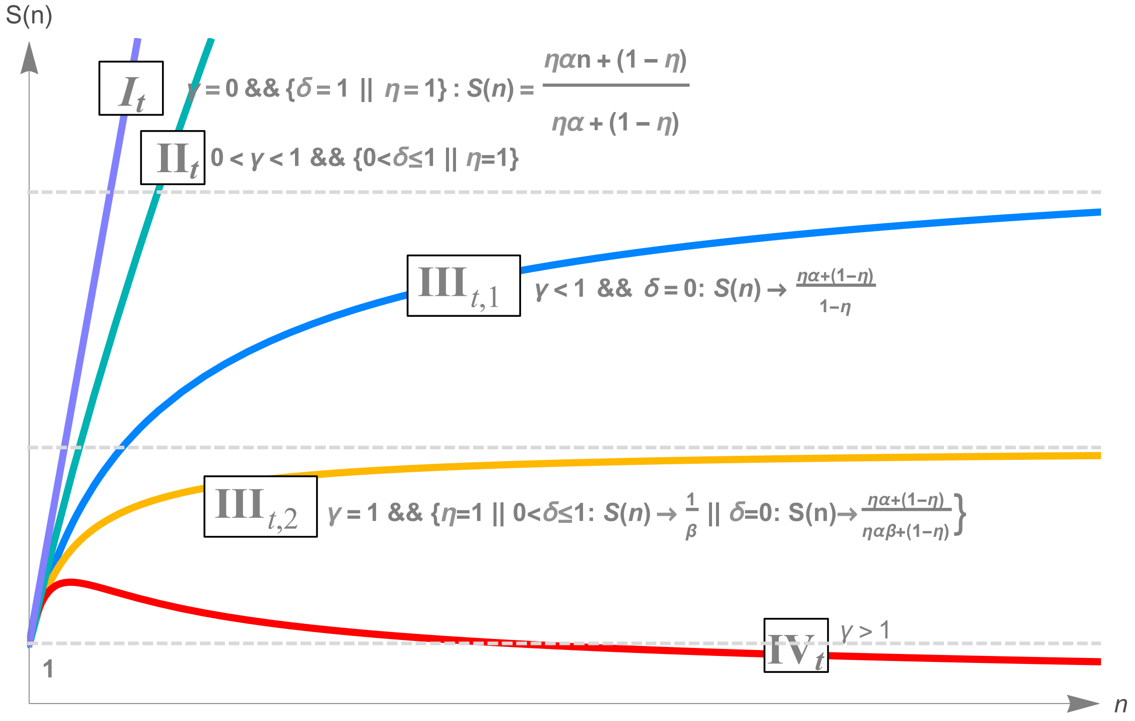


Figure 2.2: Four distinct IPSO scaling behaviors for the fixed-time workload type: I_t: Gustafson-like linear scaling; II_t: unbounded sublinear scaling; III_t: pathological, upper-bounded scaling; and IV_t: pathological, peaked scaling.

$\eta = 1$ (i.e. no serial portion of the workload), a very special case;

- II_s: Speedup grows sublinearly and unbounded. It occurs when $q(n)$ grows slower than linear, i.e., $\gamma < 1$ and $\eta = 1$, also a special case;

- III_s : It is Amdahl-like. It grows monotonically and is upper-bounded. Again, it is composed of two subtypes, i.e., $III_{s,1}$ and $III_{s,2}$, with distinct upper bounds, similar to $III_{t,1}$ and $III_{t,2}$, as depicted in Fig. 2.3. Clearly, Amdahl's law is a special case of $III_{s,1}$ at $\gamma = 0$ and $\alpha = 1$;

- IV_s: This type is pathological and behaves similar to IV_t given in Fig. 2.2. It occurs when $q(n)$ scales up superlinearly fast, regardless of how the other scaling factors behave.

In summary, both fixed-time and fixed-size workloads may suffer from pathological types of scaling, i.e., IV_t and IV_s, respectively, which by all means, should be avoided. The root cause for both IV_t and IV_s points to the superlinear scaling of $q(n)$, often seen in the case related to centralized job scheduling and data shard broadcasting. A case study of such kind will be given in the following section. The next scaling behavior that is also pathological and should be avoided is III_t. This is because I_t (or Gustafson's law) and II_t suggest that unbounded speedup should be achievable for the fixed-time workload type. On the other hand, upper-bounded speedup or III_s has long been understood to be inevitable for the fixed-size workload type, since the discovery of Amdahl's law. This is because unbounded speedup, i.e., I_s and II_s, for this workload type only occur under very special circumstances. Nevertheless, the achievable upper bound for III_s may vary and effort should be made to attain the highest possible bound.

2.5 Application of IPSO to Scaling Analysis of Data-Intensive Applications

The main focus of this section is to test the ability of IPSO in capturing the scaling properties, and hence, its suitability in serving as a diagnostic tool for scaling analysis of data-intensive applications. As a byproduct, the ability of IPSO to predict the scaling properties for some simple cases is also explored, demonstrating the promising potentials of IPSO to facilitate effective resource allocation for data-intensive applications. This study includes a total of nine cases, including four Map-Reduce-based and four Spark-based case studies, performed on the Amazon EC2 cloud, and one Spark-based case study, extracted

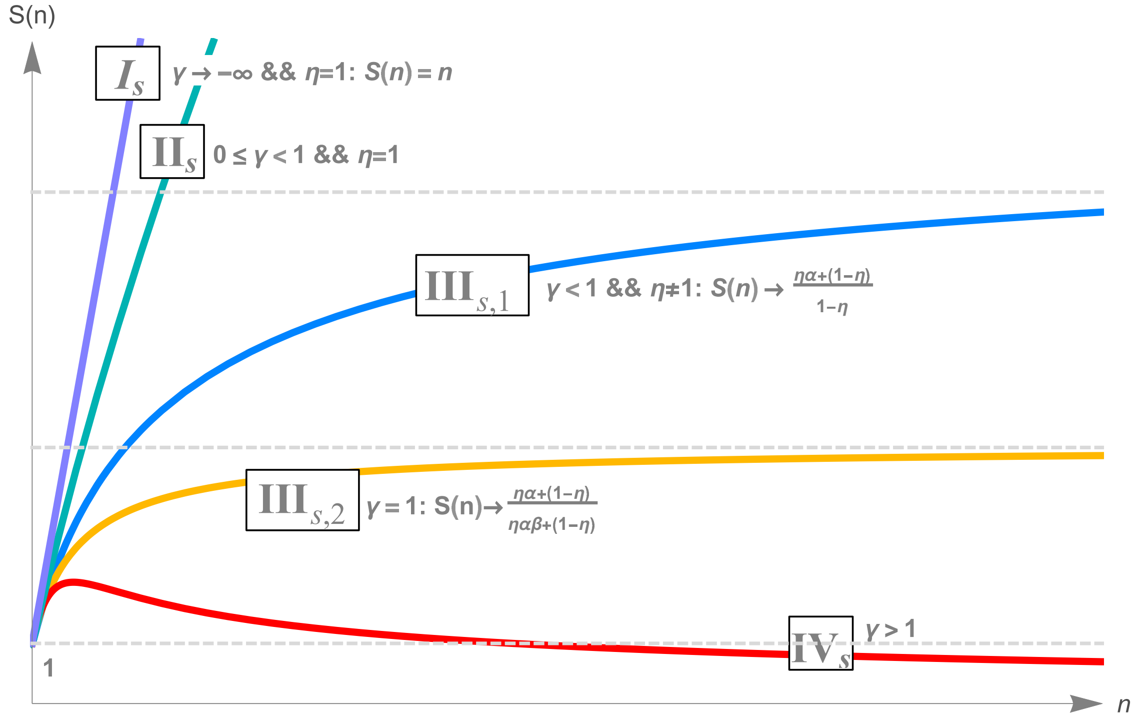


Figure 2.3: Four distinct IPSO scaling behaviors for the fixed-size workload type: I_s: linear scaling; II_s: unbounded sublinear scaling; III_s: Amdahl-like upper-bounded scaling; and IV_s: pathological, peaked scaling.

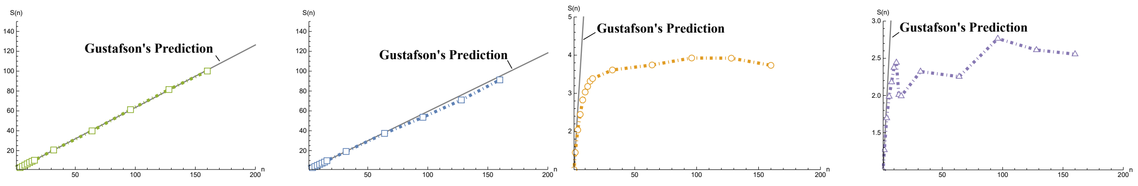
from [14]. This allows a sufficiently wide range of both single-stage (i.e., single-round) and multi-stage applications with rich scaling properties to be uncovered. These case studies also reveal the inability of the existing scaling laws in characterizing the scaling properties of data-intensive applications in general. In fact, out of the nine cases studied, only two simplest MapReduce cases follow the existing scaling laws. Finally, we note that the data presented are average results of multiple experimental runs.

2.5.1 Single-Stage Cases

This section focuses on the cases with only one round of job execution. They are either fixed-time or fixed-size, which are studied separately.

Fixed-Time Workload: The cases studied include three representative micro benchmarks from the HiBench suite [31], a widely adopted benchmark suite for Hadoop, i.e., WordCount, Sort, and TeraSort [42]. The working data sets for WordCount and Sort are randomly generated text, drawn from a UNIX dictionary that contains 1000 words. TeraSort uses a working data set derived directly from an embedded TeraGen program. Also included in this study is a QMC Pi program from Apache Hadoop examples [1]. This program uses Quasi Monte Carlo method (QMC) to estimate the value of π. These four applications are chosen because they are among the most widely studied MapReduce-based applications that involve a single round of parallel processing and merging.

On the hardware side, all of the four experiments were carried out on Amazon EC2 cloud with EMR (elastic MapReduce) support. An m4.4xlarge virtual machine (VM) instance is



(a) QMC (b) WordCount (c) Sort (d) TeraSort

Figure 2.4: Measured speedups for the four selected intel-HiBench micro benchmarks, together with the speedups predicted by Gustafson's law.

chosen as the master node and m4.large VM instances as processing units. We configure the resource manager to launch only one container per processing unit. And all the MapReduce jobs in these experiments are configured as involving a single reducer with synchronization barrier.

Results: The measured speedups for the four cases are presented in Fig. 2.4, together with the ones predicted by Gustafson's law. We make the following observations.

First, the QMC case in Fig. 2.4(a) matches Gustafson's law well. For this case, there is no serial workload, i.e., $\eta = 1$ and the fact that it matches Gustafson's law well means that it must belong to I_t, as depicted in Fig. 2.2. This further implies that there is little scale-out-induced workload involved, since $\gamma = 0$ for I_t.

Second, the WordCount case in Fig. 2.4 (b) must be either I_t or II_t, as it is close to linear growth as predicted by Gustafson's law, but more data samples at larger scale-out degree are needed to be certain which one it belongs to. Nevertheless, at least, we know from I_t and II_t that for this case, the scale-out-induced workload is very small and the speedup is very likely to be unbounded, i.e., a benign case in terms of scaling.

Third, it is clear that the Sort and TeraSort cases in Fig. 2.4 (c) and (d), respectively, significantly deviate from that predicted by Gustafson's law and fall into $III_{t,1}$ or $III_{t,2}$, or somewhere in between. Our detailed scaling factor analysis, as will be given shortly, indicates that both are more on the $III_{t,1}$ side than the $III_{t,2}$ side, since δ is close to zero and γ is likely to be small.

The above case studies clearly demonstrate that IPSO can provide significant insights on the possible root causes of the scaling behaviors for data-intensive applications. However for some cases, to exactly pin down the root cause, the exact scaling parameters, e.g., δ and γ, must be estimated. In what follows, we make a first attempt to estimate these parameters for the above cases. The aim is twofold: (a) demonstrating the promising potentials of IPSO to serve as a capable means for scaling prediction and hence, resource allocation; and (b) exposing the challenges in this pursuit.

Scaling Prediction: Regardless of the implementation details, every MapReduce job execution time can be roughly broken down into four parts: (a) the execution environment initialization and job scheduling; (b) the map phase (i.e., the split phase); (c) the communication between map and reduce phases; and (d) the reduce phase (i.e., the merge phase). The reduce phase can be further divided into a shuffle stage (the stage during which the reducer pulls all mappers' results from DFS), a merge stage (the stage when the reducer merges specific amount of mapper results to get intermediate results), and a reduce stage (the last merging process that produces the final results). With reference to this background information, the following describes how we identify, based on measurement, the scaling behaviors of the three workloads, $W_o(n)$, $W_p(n)$, and $W_s(n)$, in terms of three scaling factors, $q(n)$, $EX(n)$, and $IN(n)$, respectively.

First, by comparing the execution times of individual non-workload processing parts and stages in the scale-out execution against the corresponding times in the sequential execution, the scale-out-induced workload, $W_o(n)$, if any, can be identified (i.e., the overheads that grow with n in the scale-out execution, which are absent from the sequential job execution). $W_o(n)$ is more likely to come from parts (a) and (c). By detailed measurement, we find that $W_o(n)$, if any, is negligibly small for all four cases, due to the dominance of parts (b) and (d) and also the fact that the workload is the fixed-time one. We then further inspect the shuffle stage to see if there is any discrepancies between the two, which turns out to be negligible as well.

The next two steps are to measure $EX(n)$ and $IN(n)$. To this end, we first note that part (b), i.e., the map phase, is the sole contributor to the parallel processing phase and the rest can be attributed to the sequential merging phase. With this understanding, we estimate $EX(n)$ and $IN(n)$ by curve fitting based on the measured data at problem sizes no greater than $n = 16$ for WordCount, Sort, and QMC Pi.

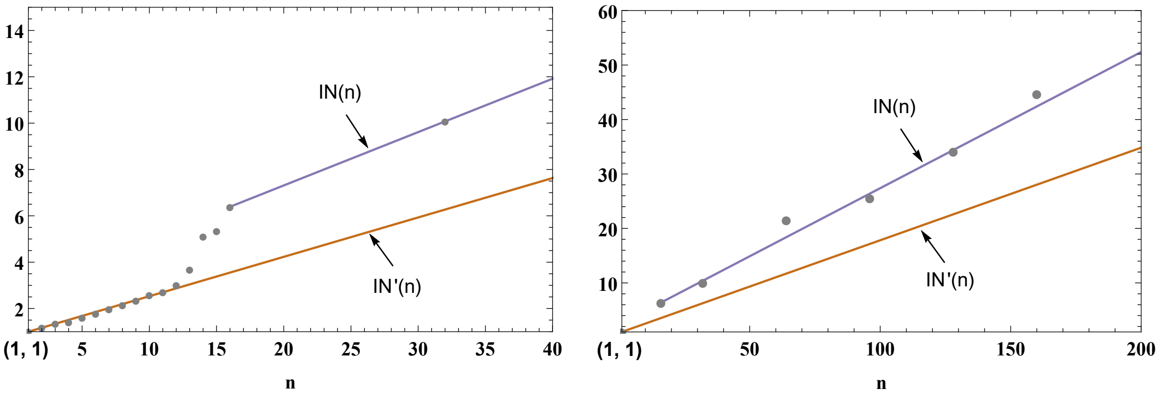


Figure 2.5: The internal-scaling factor of TeraSort demonstrates a step-wise property. As shown there are two distinct functions: a linearly increasing function $IN'(n)$ with a slower growth pace for small problem size, and $IN(n)$ at a faster growing speed when the problem size grows bigger.

For TeraSort, we use the measured data between $n = 16$ and $n = 64$. This is because for TeraSort, the data input size which grows linearly with n exceeds the preconfigured reducer memory size (∼2GB) at about $n = 15$, when the disk I/O is involved and the internal scaling factor is burst by over 30% with its slope increasing from 0.15 to 0.25 (Fig. 2.5). This phenomenon is reflected in Fig. 2.4(d) where there is a small surge of the speedup around $n = 15$ and then falls back before it grows again. It indicates that for scaling factor prediction, one must monitor the possible program execution environment changes as the problem size increases, e.g., memory capacity overflow, surging queuing delay, or onset of network or I/O bottleneck.

The predicted $IN(n)$ and $EX(n)$, together with the measured ones for the problem size reaching $n = 160$ are depicted in Fig. 2.6. As shown in the figure, the parallelizable workloads are memory bounded (i.e., using maximal block size of 128 MB per processing unit) and $EX(n)$ closely follows fixed-time workload, i.e., $EX(n) \approx n$, as expected. This confirms our earlier claim that a memory-bounded workload is no different from a fix-time workload.

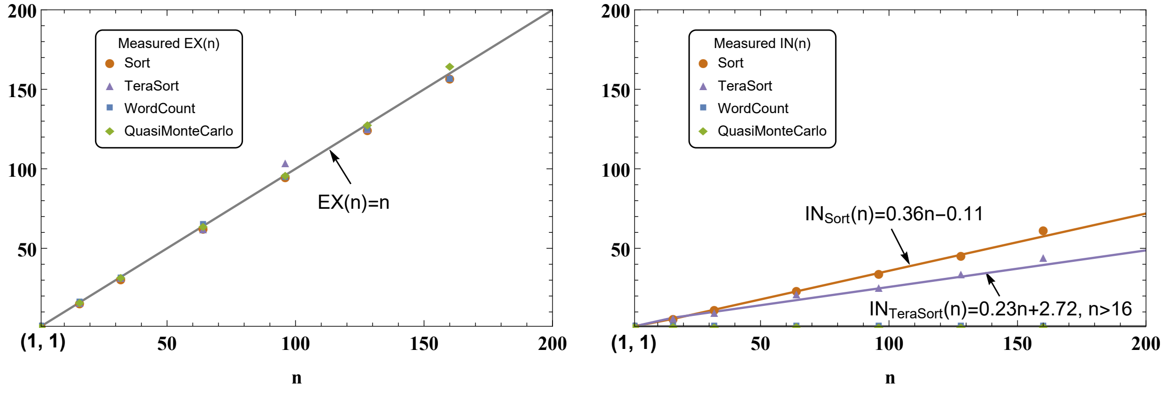
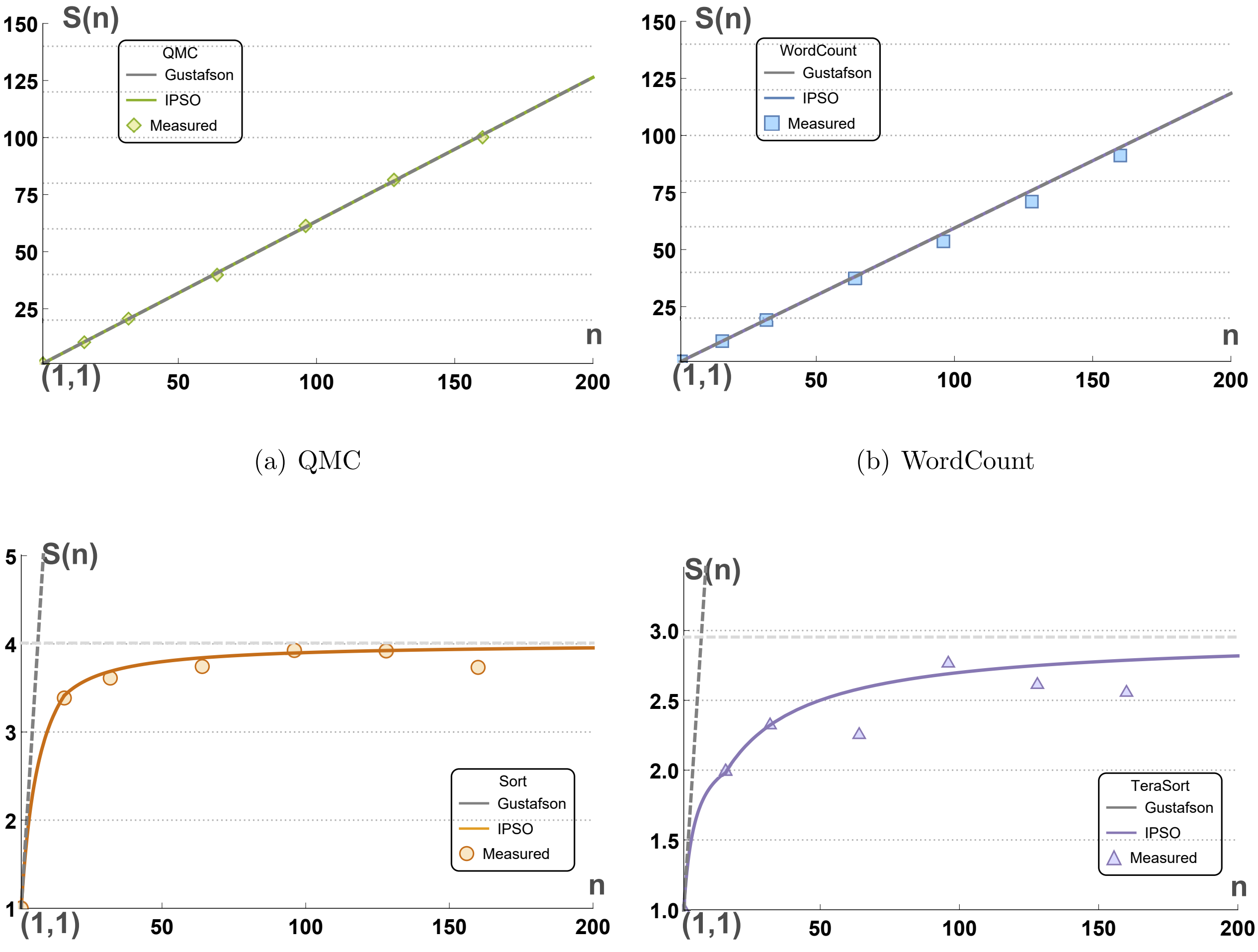


Figure 2.6: $EX(n)$ and $IN(n)$ for the four cases. For all the cases, $EX(n) \approx n$, agreeing with the Gustafson's assumption on the external scaling (see Eq. (2.13)).

For $IN(n)$ in Fig. 2.6, while WordCount and QMC pi follow the existing workload model, i.e., $IN(n) = 1$, both Sort and Terasort do not. For both applications, the predicted $IN(n)$'s based on linear regression match the measured data well.

By using both predicted and measured $EX(n)$ and $IN(n)$ above, as well as measured

$\mathbb{E}[max\{T_{p,i}(n)\}]$, $\mathbb{E}[T_{p,1}(1)]$ and $\mathbb{E}[T_s(1)]$ as input into Eqs. (2.9) and (2.8), one arrives at the speedups for all four cases, as depicted in Fig. 2.7. As one can see, overall, IPSO predicts the scaling properties for all four cases very well.



(a) QMC

(b) WordCount

(c) Sort

(d) TeraSort

Figure 2.7: The speedups based on IPSO, measurement, and Gustafson's law for the four cases in the case of the fixed-time workloads

With all the scaling factors captured, IPSO can now help characterize the scaling properties for the above four cases with much higher precision than before. First, it is now clear that the in-proportion scaling has led to the $III_{t,1}$ pathological scaling or upper bounded speedup for Sort and Terasort, which cannot be predicted by the existing scaling laws. For example, for TeraSort, even though the scaling ratio is high, i.e., $\epsilon(n) = 4.3$, meaning that the external scaling is more than 4 times faster than the internal scaling, the speedup for TeraSort is upper bounded by the value of 3. In other words, even with a relatively small internal scaling, compared with the external scaling, the speedup becomes quite limited and upper bounded.

The above examples demonstrate the promising prospects of IPSO for scaling prediction and hence, resource provisioning. In the meantime, we also realize that to this end, it entails detailed understanding and analysis of the applications and execution procedures. Clearly, it is very unlikely that an one-size-fits-all prediction solution exists. Instead, the IPSO prediction solution may need to be developed on a per programming framework and per type of applications basis.

Fixed-size Workload: Unfortunately, for all the four cases studied so far, in the case of fixed-size workload, as the scale-out factor n grows beyond 8, the parallel task response times in the map phase drop to subseconds, which cannot be measured, since in our experiments the precision of measurement is one second. Hence, instead of studying the four cases, we consider the following one.

Collaborative Filtering: As explained in [14], some iterative MapReduce machine learning applications in Spark may include significant data broadcasts from the master node to all the worker nodes per iteration. In particular, [14] describes and provides the experimental data for a collaborative filtering application (see [14] for detailed description of this application).

Table 2.1: Measured external and scale-out-induced workloads for Collaborative Filtering

n	$\mathbb{E}[\max\{T_{p,i}(n)\}]$	$W_o(n)$
10	209.0	5.5
30	79.3	17.7
60	43.7	36.0
90	31.1	54.3

In each iteration, there are two feature vectors to be updated alternately, involving two rounds of broadcast and two Map phases with barrier synchronization. For this application, each broadcast is induced as a result of the system scale-out, generating scale-out-induced workload. Moreover, the lack of a reduce phase in each iteration means that there is no sequential merging phase or $W_s(n) = 0$. Since the scale-out degrees for both rounds are the same, IPSO can be applied as aforementioned. Namely, by combining the corresponding workloads in the two rounds into one, each iteration agrees with our IPSO model, and hence, experimental data for each iteration in the form of histograms, as given in Fig. 2.3 in [14], can be leveraged by IPSO for the scaling analysis of this application. We are able to convert the experimental data in [14] into a table format in Table 2.1 and then the data are plotted in Fig. 2.8, together with the matched curves based on nonlinear regression.

First, with $W_o(n)$ given in Fig. 2.8(a) and according to Eqs. (2.6) and (2.15) (note that $W_p(n) = W_p(1) = \mathbb{E}[T_{p,1}(1)]$ is a constant for the fixed-size workload), we have, $\gamma = 2$. By inspecting Fig. 2.3, it becomes clear that the speedup for this case must fall into IV_s, the worst-case scenario, rather than the best-case scenario, i.e., I_s, as would be predicted by

Amdahl's law (note that $\eta = 1$ as there is no serial workload).

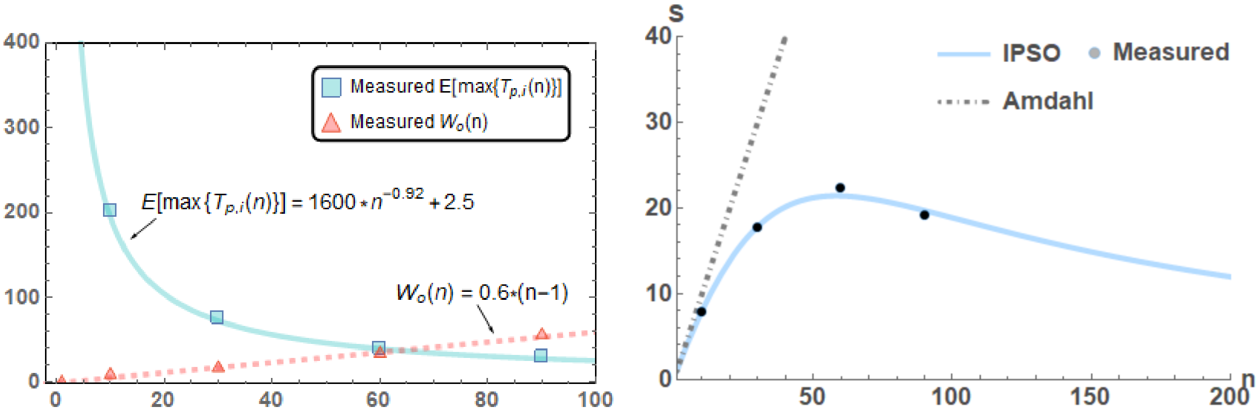
To verify that IPSO indeed predicts the scaling property for the current case accurately, in what follows, we calculate $S(n)$ based on the statistic version of the IPSO model.

First, given $W_p(n) = W_p(1) = \mathbb{E}[T_{p,1}(1)]$, the speedup, $S(n)$, for this application, according to Eq. (2.7), can then be written as follows,

$$S(n) = \frac{\mathbb{E}[T_{p,i}(1)]}{\mathbb{E}[\max\{T_{p,i}(n)\}] + W_o(n)} \tag{2.18}$$

By extrapolating the matched curve for $\mathbb{E}[\max\{T_{p,i}(n)\}]$ in Fig. 2.8(a) to $n = 1$, we have $\mathbb{E}[T_{p,1}(1)] = 1602.5$. Finally, $S(n)$ is evaluated by substituting the matched functions in Fig. 2.8(a) into Eq. (2.18).

Fig. 2.8(b) depicts the measured speedup and IPSO speedup, along with the speedup predicted by Amdahl's law. As one can see, both measured and IPSO speedups confirm that the scaling behaviors follow IV_s, rather than I_s, as predicted by Amdahl's law.



(a)

(b)

Figure 2.8: The measured and IPSO speedups, together with that predicted by Amdahl's law for Collaborative Filtering.

The scaling properties, depicted in Fig. 2.8(a), is pathological and disappointing. It simply states that even in the absence of the serial portion of the workload, the linear speedup scaling, predicted by Amdahl's law, is not guaranteed. The dismal speedup of 21 at its peak and continued trending towards zero, as predicted by IPSO, indicates that the scale-out-induced scaling can pose serious threats to the scalability of scale-out applications. Collaborative Filtering deals with a fixed problem size, i.e., $EX(n) = 1$, meaning that scaling out beyond $n = 60$ can only do harm to the parallel computing, hence setting a hard scale-out degree upper bound, beyond which the parallel computing performance deteriorates.

2.5.2 Multi-Stage Cases

In this section, IPSO is applied to the analysis of Spark-based applications, involving multiple stages/rounds of parallel task processing per job execution, where the scale-out degree may vary from one stage to another. However, as we mentioned earlier, the IPSO model can only be directly applied to the multi-stage cases with the same scale-out degree.

Fortunately, in the Spark-based program, two configuration parameters, i.e., the problem size, N, and parallel degree, m, must be set. Here N is the nominal number of tasks to be executed in a stage and m the nominal number of executors (i.e., processing units) to run in parallel in a stage. In general, the actual numbers of tasks and executors in the first stage are indeed equal to N and m, respectively. In other words, each of the m executors in the first stage needs to execute N/m tasks sequentially. Although the numbers of tasks and executors in the subsequent stages may not be equal to their respective nominal counterparts, they are strong functions of these counterparts. This implies that the three workloads for Spark-based applications can be defined in terms of N and m, i.e., $W_p = W_p(N, m)$, $W_s = W_s(N, m)$, and $W_o = W_o(N, m)$ in general.

To allow IPSO to be applied to the cases with multi-stage without modification, we simply define the fixed-time and fixed-size cases as when N/m and N are fixed, respectively, while scaling m, i.e., the scale-out degree, $n = m$. By doing so, all the above three workloads become functions of n only for either case, the same as the ones defined in IPSO. Hence, all the results derived from IPSO apply to the Spark-based multi-stage cases. In what follows, we perform case studies for four representative Spark benchmarks.

We deployed the Intel HiBench suite on EC2 with the EMR (Elastic MapReduce) service and then launched four Spark benchmarks including three machine learning applications:*Bayes Classifier (Bayes)*, *Random Forest (RF)*, *Support Vector Machine (SVM)*, and one graph application (*NWeight*). The hardware configuration is cloned from the previously stated MapReduce experiment. All the instances are preconfigured with sufficient storage capacity (100GB per-node) and bandwidth ($\geq$ 450 Mbps). Executor reuse and logging are enabled for performance metrics collection. All the input data sets are generated by the respective data generators provided by this benchmark suite. We then extract the execution latencies for all stages from the application's Log file to derive the speedup. This is done by tracing the timestamps for each stage in the Spark Log files, which are available in the *JSON* format.

To identify the scaling properties using IPSO, we are interested in the speedups projected onto two different dimensions, i.e., the fixed-time (with N/m fixed) and fixed-size (with N fixed) dimensions, while scaling $n = m$ as depicted in Fig. 2.9 and Fig. 2.10 respectively. Along with the data points, we also plotted the projected curves of the matched two-dimensional surfaces as functions of N and m based on nonlinear regression for the ease of identification of trending.

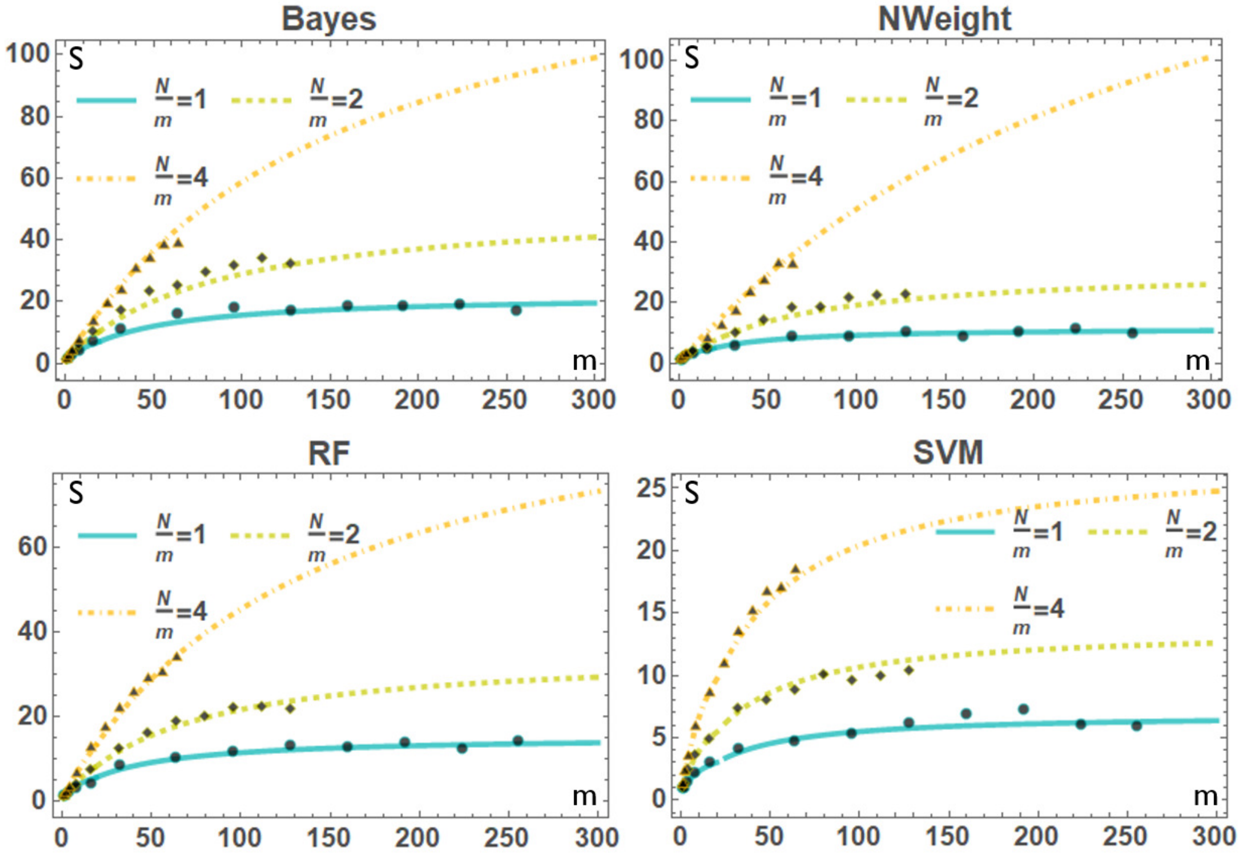


Figure 2.9: Fixed-time dimension

First, one notes that for fixed-time workloads, as depicted in Fig. 2.9, the larger the per executor load level, $\frac{N}{m}$, the higher the speedup is. In other words, the speedup curve at $\frac{N}{m} = 4$ is higher than that at 2, which in turn, is higher than that at 1. The detailed analysis of the code paths indicates that this is due to the increased percentage of the scale-out-induced workload as the number of tasks per executor decreases. More specifically, the scheduling and deserialization time (i.e., the communication cost) of the first wave of tasks outweigh the following waves. Hence, further increasing per-node workload level effectively reduces the scale-out induced workload per task. This however, by no means suggests that reducing the parallel degree always improves speedup performance. In fact, our study shows that the speedup at $\frac{N}{m} = 8$ is lower than that at $\frac{N}{m} = 4$. This is because the per-node workload level is constrained by the available resource at the node level. For example, insufficient RAM may cause the persistent RDDs to be spilled to the local disk, or even trigger increased task failure rate, leading to the rollback to the previous stage and hence poor performance. In other words, the optimal scale-out level, or parallel degree m is determined by both the workload size and the resource availability at individual executors.

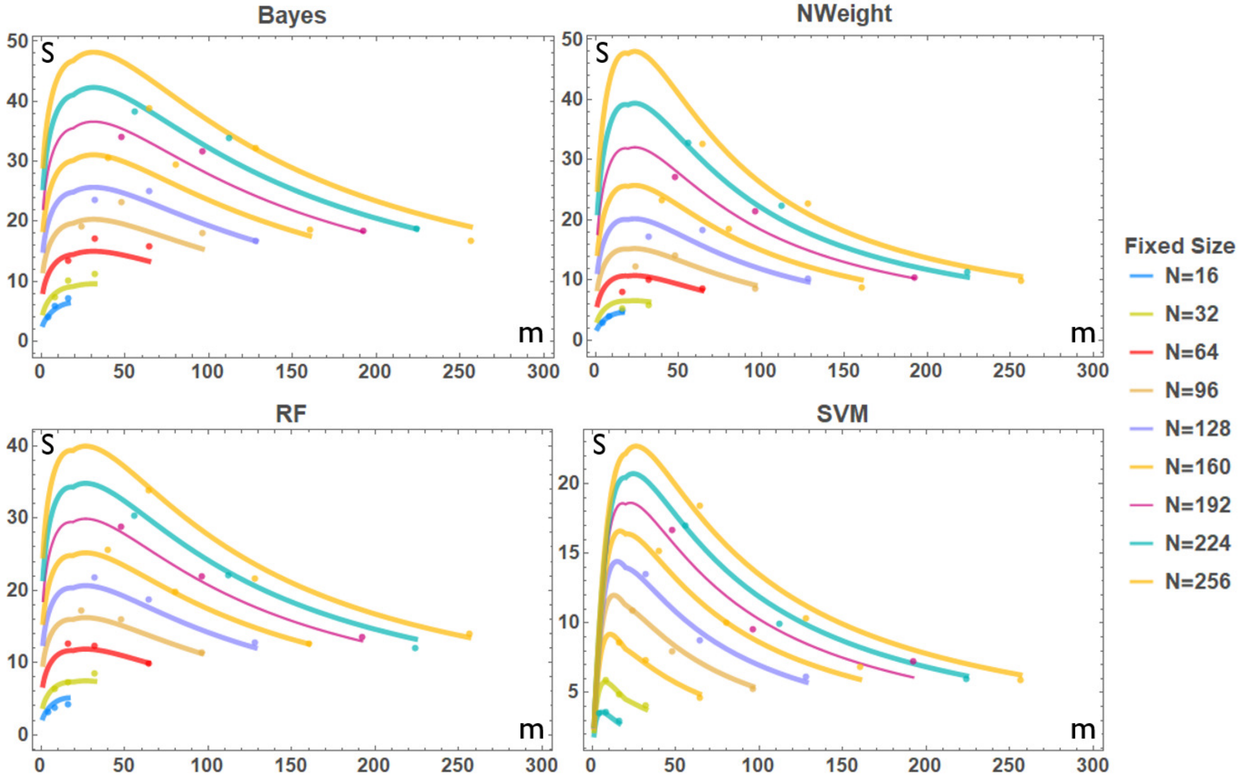


Figure 2.10: Speedups in fixed-size dimension

The scale-out-induced workload is also responsible for the degradation of the speedups from the ideal Gustafson-like I_t to at best II_t for large $\frac{N}{m}$ and III_t for smaller $\frac{N}{m}$. All four cases share similar scaling properties.

Second, for the fixed-size workload dimension, as depicted in Fig. 2.10, as the problem size N becomes large, the speedups for all four cases peak and then fall as n increases, agreeing with the scaling behavior of IV_s, the pathological one. This is in stark contrast with that predicted by Amdahl's law, or III_s. This is due to the strong scale-out induced overhead, as discussed above. Again, all four cases share similar scaling behaviors.

In summary, the above case studies clearly demonstrate that IPSO can serve as a diagnostic tool for the scaling analysis of data-intensive applications. Specifically, the following diagnostic procedure is recommended:

1. Determine the use case scenario, i.e., the fixed-time or fixed-size workload;

2. For given workload type, measure the speedup as the scale-out degree increases;

3. Plot the speedup data points versus scale-out degree, maybe together with a matched curve from nonlinear regression as a guide;

4. Compare the trending of the measured speedup with either Fig. 2.2 or Fig. 2.3, depending on the workload type to identify closely matched scaling types;

5. If the matched scaling type is I_t (I_s), II_t (II_s), or IV_t (IV_s), the root causes of the scaling behaviors are readily identified and understood based on the analysis in Section IV. If, however, the matched type is III_t (III_s), go to the next step to further identify which sub-type, $III_{t,1}$ ($III_{s,1}$) or $III_{t,2}$ ($III_{s,2}$) it belongs to;

6. Carry out a detailed analysis and measurement of the scaling parameters, δ and γ, to pin down the exact sub-type it belongs to, which however, may need to be done case

by case.

Finally, the source code and executables for all the case studies are publicly available at our code repository [3].

2.6 Conclusions and Future Work

In this paper, we proposed a new scaling model for scale-out, data-intensive applications, called In-Proportion and Scale-Out-induced scaling model (IPSO). IPSO sets itself apart from the existing scaling models in two important aspects. First, it takes the scaling of the serial portion of a workload into account, referred to as *in-proportion scaling*. Second, it accounts for the possible overheads induced by the scale-out, resulting in what we call *scale-out-induced scaling*. Both in-proportion and scale-out-induced scalings were observed in scale-out applications, but had not been formally defined until this work. MapReduce-and-Spark-based application case studies demonstrate that IPSO can serve as a powerful diagnostic tool for the scaling analysis of data-intensive applications.

As our future work, we plan to develop measurement-based resource provisioning algorithms for data-intensive workloads based on the prediction ideas behind IPSO. The key is to find a solution as to how to quickly estimate the two scaling parameters, δ and γ. The research in this direction is underway.

A Unified Scaling Model in the Era of Big Data Analytics

Li, Zhongwei, Feng Duan, and Hao Che. “A unified scaling model in the era of big data analytics.” In Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications, pp. 67-77. ACM, 2019.

Chapter 3

USBA Model

3.1 Introduction

Big data analytics are scale-out by design with a large number of data blocks to be processed in parallel. For such applications, a job may involve multiple stages of parallel task processing and task result exchanges for tasks mapped to up to tens of thousands of commodity servers in which the data blocks reside. Notable examples are data mining, machine learning, and graph processing based on, e.g., MapReduce [18], Spark [64], and/or Dryad [33] programming frameworks. As big data analytics have emerged as the predominant datacenter workloads and consumed a significant portion of datacenter resources, it is of paramount importance to gain a sound understanding of the scaling properties of such workloads, e.g., for job resource provisioning purpose.

To this end, one must establish good system and workload models for big data analytics workloads and then, on the basis of these models, predict the scaling behaviors for such workloads. Unfortunately, to date, no systematic approach of this kind has been developed. Consequently, the most widely cited scaling law for big data analytics, to date, is still the traditional, well-known Amdahl's law [5], which along with a seemingly competing, well-known scaling law – Gustafson's law [27], were discovered well before the era of big data analytics. Since both the system and workload models underlying these scaling laws are too simplistic, applying these laws to the scaling analysis of big data analytics workloads often brings more confusion than clarity.

First, the underlying system model shared by Amdahl's and Gustafson's laws can be viewed as a two-stage system with a serial and a parallel execution stage. While the serial execution stage has one processing unit in it, handling the serial portion of the workload, the parallel execution stage involves a number of processing units processing the parallelizable portion of the workload in parallel. The workload models underlying both Amdahl's and Gustafson's laws assume that the size of the serial portion of the workload stays unchanged as the system scales out, which is justified, according to [27], as the serial portion of the workload normally corresponds to the program initialization phase that does not change as the problem size increases. This is in stark contrast to the system and workload structures for big data analytics, which in addition to the initialization phase, normally involves from a few to several thousands of parallel execution stages, which must be executed sequentially. Moreover, as the problem size increases, in general, the number of parallel execution stages may grow, effectively adding more parallel and serial workloads simultaneously.

Second, the workload models for the parallelizable portion of the workload underlying both Amdahl's and Gustafson's laws are too restrictive, i.e., the fixed-size workload model for Amdahl's law and the fixed-time workload model for Gustafson's law, for which the size of the parallelizable portion of the workload stays unchanged and scales linearly, respectively, as the system scales out. In contrast, big data analytics workloads may stay unchanged, scale linearly or sublinearly, depending on the resource availability with respect to the workload size (see Section 2 for more details).

In lieu of the lack of and the importance of gaining a sound and fundamental understanding of the scaling properties and to facilitate more effective job resource provisioning for big data analytics workloads, in this paper, we put forward a Unified Scaling model for Big data Analytics (USBA) that renders the traditional scaling models its special cases. It makes the following major contributions. First, USBA is a unified scaling model, from which the well-known Amdahl's and Gustafson's laws, and other useful performance bounds can be derived as special cases. Second, USBA is a general scaling model that allows for flexible multi-stage system and workload scalings; variability of task response times; and barrier synchronization. Hence it can be applied to the scaling analysis of real-world big data analytics workloads, in the presence of a wide range of system and workload scaling behaviors and variability of task response times, e.g., task stragglers. Third, in USBA, a regression model is developed, that can match the measured performance data and predict the performance scaling within 20% errors for all four benchmarks studied [1]. Finally, the ability to work in a large workload design space allows USBA to capture the true scaling properties of big data analytics workloads that largely deviate from those predicted by the Amdahl's and Gustafson's laws.

The remainder of the paper is organized as follows. Section 3.2 provides the background information to motivate the current work. Section 3.3 introduces USBA. Section 3.4 presents the results for the scaling analysis of USBA, and its application to several Spark henchmarks. Section 3.5 reviews the related work. Finally, Section 3.6 concludes the paper.

[1] The prediction errors at this level is considered small for resource provisioning [60].

3.2 Background and Motivations

In this section, we first analyze in detail the Amdahl's and Gustafson's laws and the corresponding system and workload models. Then we identify, from both system and workload modeling points of view, the fallacies of applying these laws to characterize the scaling properties of big data analytics workloads, which motivate the current work.

To capture the workload models underlying the existing well-known scaling laws and to be consistent with the USBA scaling model to be defined later, we view the system model shared by Amdahl's and Gustafson's laws as depicted in Figure 3.1. It is composed of a single processing unit and a set of parallel processing units arranged in a two-stage pipeline, executing the serial and parallelizable portions of the workload, respectively. In this model, each black dot represents a base task or workload unit, which cannot be further divided and parallelized. There is only one base task at the serial stage, representing the serial portion of the workload. This discretization of the workload leads to discretized versions of the two laws, as we shall see shortly.

Now let τ_s and τ_p denote the execution times for a base task in the serial and parallel stages, respectively. In the parallel processing stage, the total number of base tasks is $N_p = m_p \times n_p$, called the problem size, where m_p is the number of parallel processing units (also called executors) or degree of parallelism (DOP); and n_p is the number of base tasks mapped to and processed sequentially by each executor in the parallel stage, referred to as *per-node workload level* hereafter.

Now, the speedup, $S(N_p, m_p)$, for the above system model can be written as the following,

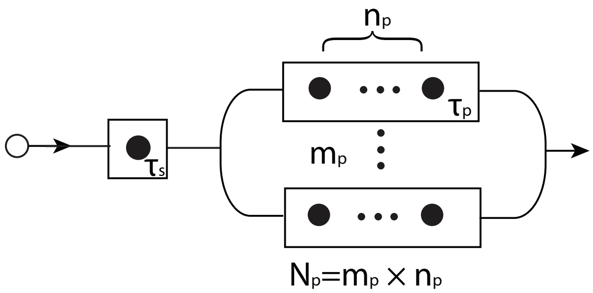


Figure 3.1: System Model for Amdahl's and Gustafson's laws. Each black dot represents a base task and each white box represents an executor. It has two stages, i.e., a serial and a parallel execution stage.

$$S(N_p, m_p) = \frac{\tau_s + N_p \tau_p}{\tau_s + n_p \tau_p} \tag{3.1}$$

where the numerator is the job response time when all the tasks are executed sequentially using a single executor and the denominator is the job execution time based on the above two-stage execution model. By writing down this formula, we implicitly assume that τ_s and τ_p stay the same for both sequential execution model and parallel execution model, which may not hold true in general, as we shall see. Now, both Amdahl's and Gustafson's laws can be easily derived from this expression, by taking into account of their respective workload models.

For fixed-sized workload, the problem size, N_p is a constant, we have the following discretized Amdahl's law,

$$S = \frac{1}{\frac{\eta}{m_p} + (1 - \eta)} \tag{3.2}$$

where,

$$\eta = \frac{N_p \tau_p}{N_p \tau_p + \tau_s} \tag{3.3}$$

is a constant. It simply states that for the fixed-size workload, the speedup is upper bounded by $\frac{1}{1-\eta}$, regardless of how many executors, m_p, are in use, where $m_p \leq N_p$. This is a discretized version of the original Amdahl's law. While the underlying parallelizable portion of the workload in the original Amdahl's law is infinitely divisible, the current one can only be divided into up to N_p base tasks, which are processed by up to $m_p = N_p$ parallel executors.

For fixed-time workload, i.e., n_p is a constant, we arrive at the following Gustafson's law,

$$S = m_p \epsilon + (1 - \epsilon) \tag{3.4}$$

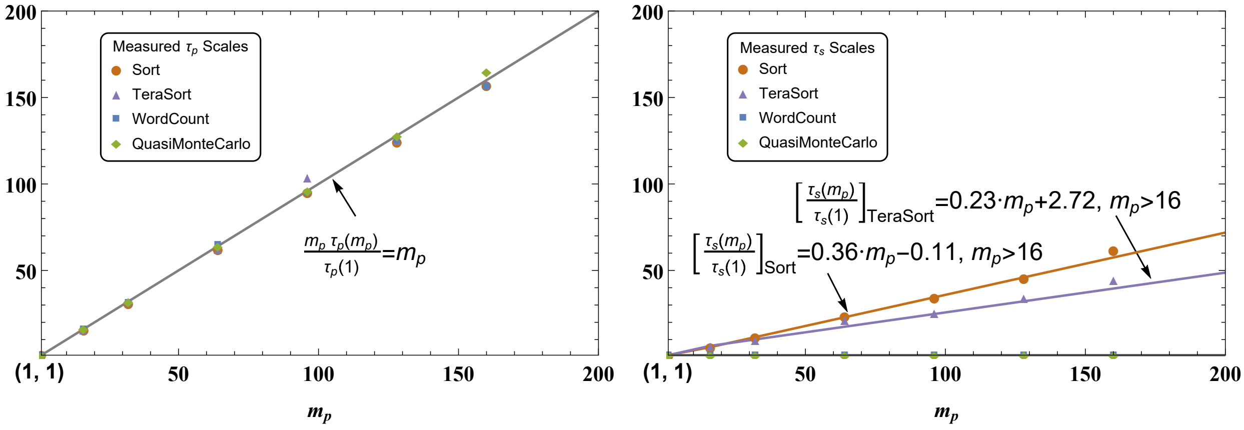
where,

$$\epsilon = \frac{n_p \tau_p}{n_p \tau_p + \tau_s} \tag{3.5}$$

is a constant. It simply states that for the fixed-time workload, the speedup grows unbounded, as both m_p and N_p, increase at fixed $n_p = \frac{N_p}{m_p}$. Again, it is a discretized version of the original Gustafson's law, as the parallelizable portion of the workload mapped to any given executor, is measured in terms of an integer number of base tasks, i.e., the per-node workload level n_p. Traditionally, the two scaling laws are considered as two mutually exclusive views of scaling for parallel computing. Now, we identify the limitations of the system and workload models underlying the above scaling laws.

First, we observe that in the context of big data analytics workloads, the fixed-size and fixed-time workload models represent only two extreme scenarios of the same problem, i.e., resource-abundant and resource-constrained scenarios, respectively. By resource-abundant, we mean that the parallelizable portion of the workload can be processed in its entirety by just one executor, i.e., $m_p = 1$ and $n_p = N_p$. In this case, one is interested in characterizing the scaling properties when the fraction of the parallelizable workload on each executor, $n_p = \frac{N_p}{m_p}$, decreases as the number of executors, m_p, increases at fixed N_p, leading to Amdahl's law in Eq. (3.2). By resource-constrained, we mean that each executor can only handle a fraction of the total parallelizable portion of the workload, $n_p = \frac{N_p}{m_p}$, e.g., the case when the memory allocated to each executor is fully occupied by the data block assigned to it. In this scenario, the workload grows with the number of allocated executors, m_p, resulting in Gustafson's law (Eq. (3.4)). Clearly, both scenarios may occur in practice, especially in a consolidated datacenter environment, where the resource availability may vary significantly from peak hours to off peak hours and the workload size may change dramatically from one application to another. Moreover, in general, as the system scales out, a big data analytics workload may scale in terms of both N_p and m_p, not following any of these two workload models. Hence, from the job resource provisioning point of view, one would be interested in finding the maximum speedup trajectory as a function of N_p and m_p. Clearly, the system model with workload discretization given in Figure 3.1 allows all possible trajectories to be characterized, not limited to the fixed-size or fixed-time workloads and hence, is a good model to allow a large workload design space to be exploited.

Second, we note that the base tasks for both parallel and serial portions of the workload underlying both Amdahl's and Gustafson's laws are fixed, i.e., τ_p and τ_s are constants and does not scale with m_p or N_p. For big data analytics workloads, however, this is not the case in general. As we shall see in Section 3.4, both may scale with N_p and/or m_p for Spark-based multi-stage workloads.



(a) Parallel Scaling

(b) Sequential Scaling

Figure 3.2: The scaling of serial and parallel portion of MapReduce workloads.

Even for some simple MapReduce-based applications, strong scalings for the serial portions of the workloads are found. For example, we do the experiments on Amazon EC2 cloud with EMR (elastic MapReduce) for four representative micro benchmarks from HiBench suite [31], including WordCount, Sort, TeraSort, and Quasi Monte Carlo (QMC) Pi program. We purposely configure only a single reducer in the Reduce phase so that the initialization and reducer phases together can mimic the serial stage in Figure 3.1. We consider the fixed-time workload by increasing the number of executors, m_p, in the Map phase with a 128 MB data block for every executor added. Figure 3.2(a) and (b) depict the scaling of $r_p(m_p) = \frac{\tau_p(m_p)}{\tau_p(1)}$ and $r_s(m_p) = \frac{\tau_s(m_p)}{\tau_s(1)}$, assuming that $\tau_p = \tau_p(m_p)$ and $\tau_s = \tau_s(m_p)$. As one can see, while none of the four exhibit visible parallel scaling, both Sort and TeraSort see strong linear serial scalings with m_p. As a result, while WordCount and QMC Pi follow Gustafson's law and grow unbounded, Sort and TeraSort are upper bounded, as shown in Figure 3.3.

The above examples clearly indicate that a good scaling model for big data analytics workloads should take the possible scalings for both parallel and serial portion of the workload into account.

Third, as explained in the introduction section, big data analytics may involve many parallel execution stages in addition to the initialization stage. Also, as the problem size increases, the number of parallel execution stages may grow. Clearly, the system model in Figure 3.1 fails to capture these features.

The above fallacies in applying the traditional scaling laws to big data analytics workloads motivate the work presented in this paper.

3.3 USBA

In this section, we introduce the system and workload models, the speedup, and the regression model, separately.

3.3.1 System Model with Discretized Workload

Consider a system model with discretized workload, depicted in Figure 3.4. It generalizes the two-stage system model in Figure 3.1 to a $(L+1)$-*stage* system model with a serial initialization stage and L parallel task execution stages with (the dotted vertical lines) or without synchronization barriers. At any given stage, k, there are m_k executors working in

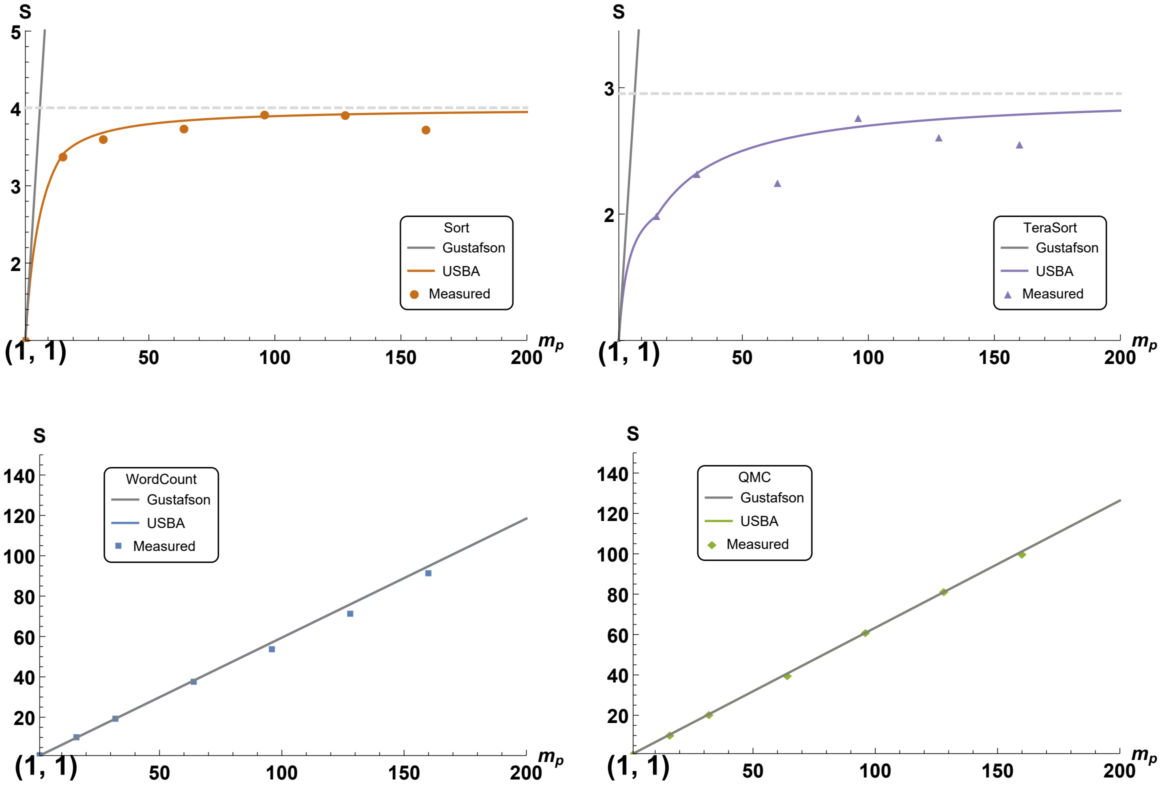


Figure 3.3: The speedups for four MapReduce benchmarks with fixed-time workloads. As shown, when taking serial scaling effects into account, big data analytics workloads may no longer follow Gustafson's law, i.e., the Sort and TeraSort cases.

parallel, each executing $n_k = \frac{N_k}{m_k}$ base tasks sequentially, where N_k is the total workload or total number of base tasks at stage k.

Now, define $t_k(N_k, m_k; i, j)$ to be the task execution time for the base task at position j in the ith executor at stage k, which in general is a random variable. As our experiment shows, it can be a strong function of the workload size, N_k, as well as parallel degree, m_k, at stage k. To limit the exposure, we further assume that all the base tasks at any given stage k have the same mean task execution time, i.e., $\mathbb{E}[t_k(N_k, m_k; i, j)] = \mathbb{E}[t_k(N_k, m_k)]$, for $i = 1, ..., m_k$ and $j = 1, ..., n_k$.

First, we note that in the above model, the serial initialization stage $k = 0$ is modeled separately from the parallel execution stages, even though it could be treated as special case of a parallel execution stage with a single base task. As we shall see later, this is because the initialization stage exhibits totally different scaling properties from the parallel execution

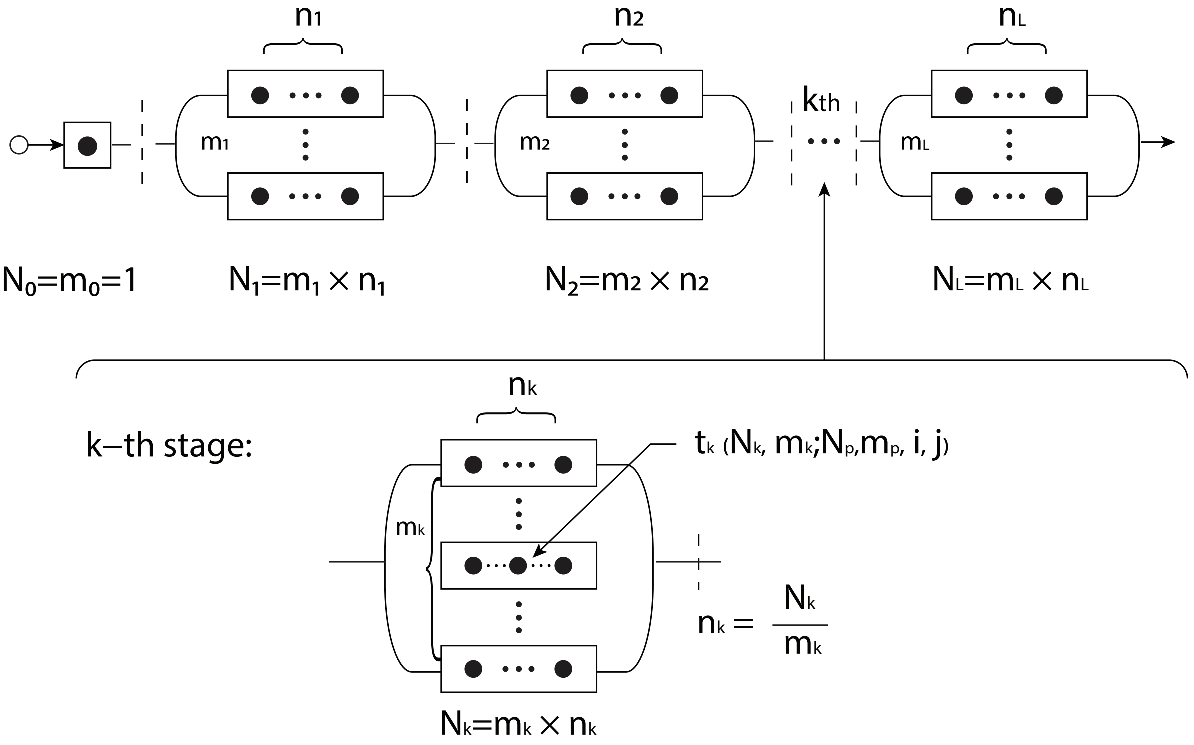


Figure 3.4: A system model with discretized workload. Stages are divided by data transformations that potentially involve large sequential overhead caused by the barrier, normally known as "shuffle" boundaries. Each white circle represents a driver process in that stage which takes input and dispatches tasks to its workers. The workers are usually running VM instances on core/task nodes of the cluster, depicted as white squire boxes in the figure. And each black circle represents a task being assigned to the worker. Also, when a multi-stage workload contains multiple chains of stages, the longest chain of stages with dependencies shall drive its performance in terms of execution latency.

stages and hence, needs to be modeled and predicted separately from the rest of the stages.

Second, we note that the above model inherits the nice features from the model in Figure 3.1. In particular, it can accommodate various workload models, including the fixed-size workload (i.e., increasing m_k at fixed N_k for $k = 1, ..., L$), fixed-time workload (i.e., increasing N_k at fixed $n_k = \frac{N_k}{m_k}$ for $k = 1, ..., L$), or any combinations of the two in a $2L$-multidimensional space defined by $N_1, m_1, ..., N_L, m_L$. Moreover, it degenerates to the model in Figure 3.1 at $L = 1$, meaning that Amdahl's and Gustafson's laws can be derived from the above model as special cases.

Finally, we note that the $2L$ multidimensional space of the above model can be reduced to a two dimensional space, greatly simplifying the scaling analysis of the model. As we shall see later, for Spark-based workload, the workload size in terms of N_k and m_k at parallel execution stage k, for $k = 1, ..., L$, are strongly correlated with the problem size, N_p, defined as the initial number of parallel base tasks that a job spawns, as well as the degree of parallelism, m_p, i.e., the number of executors configured, where in general, $N_1 = N_p$ and $m_1 = m_p$. Moreover, the number of stages, L, may also be a function of N_p and m_p. As we shall see, this observation lends us an effective means to characterize and predict the scaling properties of a big data analytics workload, using only two parameters, N_p and m_p.

3.3.2 Speedup

The speedup for a given problem size must be measured against a baseline solution. Ideally, a baseline solution should follow a strictly sequential execution model, e.g., the sequential execution model in the case of Amdahl's or Gustafson's law. Unfortunately, for big data analytics, a strictly sequential execution model may not be implementable, as the most probable reason for scale-out processing of a big data workload is that both time and space complexities to do so sequentially are too high.

An alternative solution, e.g., proposed in [58], is to replace the baseline sequential model with a baseline scale-out model at the lowest possible DOP (degree of parallelism) that is implementable. For example, in the context of our USBA model, the speedup, S_r, thus defined can be written as,

$$S_r(L, \{N_k, m_k\}) = \frac{\mathbb{E}[t_0(N_1, m_1^o)] + \mathbb{E}[T(L, \{N_k, m_k^o\})]}{\mathbb{E}[t_0(N_1, m_1)] + \mathbb{E}[T(L, \{N_k, m_k\})]} \tag{3.6}$$

where $t_0(N_1, m_1)$ is the task response time in the initialization stage, which is a function of the problem size; $T(L, \{N_k, m_k\})$ is the job response time with respect to the parallel execution stages; the numerator is the average job response time for the baseline scale-out configuration at the lowest DOP in terms of m_k^o's and the denominator is the average job response time for the desired scale-out configuration, where $m_k > m_k^o$ for $k = 1, ..., L$.

Although offering a measurable speedup for a scale-out configuration versus a baseline one, the above definition suffers from a major drawback. As the problem size increases, the DOP for the baseline, i.e., m_k^o's, must also increase to be implementable. Hence, the speedup becomes a relative measure depending on the range of problem sizes. This makes it difficult to identify the true performance bounds as problem size becomes large, as there is no fixed point of reference.

To address the above issue, USBA adopts the following ideal, easily implementable sequential execution model as the baseline. This baseline model simply uses the mean job response time measured at the minimum problem size, i.e., $N_p = 1$ and $m_p = 1$ (i.e., without scale-out effects), multiplied by N_p as the mean job response time, $\mathbb{E}[T_o(N_p)]$, at any given problem size in terms of N_p, independent of m_p. More specifically, assume that $L = L(N_p, m_p)$ in general. At the smallest problem size, i.e., $N_p = 1$, $m_p = 1$ and $L = L(1,1)$, define τ_k to be the average task execution times, i.e., $\tau_0 = \mathbb{E}[t_0(1,1)]$ and $\tau_k = \mathbb{E}[t_k(1,1)]$, for $k = 1, ..., L(1,1)$. Then, we have,

$$\mathbb{E}[T_o(N_p)] = N_p \times T(L(1,1), \{1,1\}) = N_p \sum_{k=1}^{L(1,1)} \tau_k \tag{3.7}$$

Clearly, this ideal sequential model is free from the scale-out effects, i.e., independent of m_p, and hence, is truly a pure sequential one.

With the above baseline sequential execution model, now we can define speedup, $S(L, \{N_k, m_k\})$, as follows,

$$S(L, \{N_k, m_k\}) = \frac{\mathbb{E}[t_0(1,1)] + \mathbb{E}[T_o(N_p)]}{\mathbb{E}[t_0(N_1, m_1)] + \mathbb{E}[T(L, \{N_k, m_k\})]} \tag{3.8}$$

Now, with reference to the model in Figure 3.4, $\mathbb{E}[T(L, \{N_k, m_k\})]$ can be written as follows,

$$\mathbb{E}[T(L, \{N_k, m_k\})] = \sum_{k=1}^{L} \mathbb{E}[R(k, N_k, m_k)] \tag{3.9}$$

where $R(k, N_k, m_k)$ is the job response time with respect to stage k and is given by,

$$R(k, N_k, m_k) = \begin{cases} \max_{j\in[1,m_k]}\{\sum_{i=1}^{N_k/m_k} t_k(N_k, m_k; i, j)\}, \\ \qquad\qquad\qquad\qquad \text{with barrier,} \\ \sum_{i=1}^{N_k/m_k} t_k(N_k, m_k; i, 1), \\ \qquad\qquad\qquad\qquad \text{without barrier.} \end{cases} \tag{3.10}$$

This expression accounts for the case with or without synchronization barrier. For the case with barrier, $R(k, N_k, m_k)$ is determined by the slowest task at stage k. This definition allows long-tail task execution effects, e.g., due to stragglers [67], to be captured. For the case without barrier, since the total mean response times for the n_k base tasks at different parallel executors are the same, without loss of generality, $R(k, N_k, m_k)$ is expressed as the response time with respect to the first executor.

Clearly, with the above definition, the speedups for all the execution configurations in terms of m_k's can be compared, as they are measured against the same baseline. Moreover, there is an added benefit of this definition. Namely, the speedup in Eq. 3.6 can be derived from the current speedup as follows,

$$S_r(L, \{N_k, m_k\}) = \frac{S(L, \{N_k, m_k\})}{S(L, \{N_k, m_k^o\})} \tag{3.11}$$

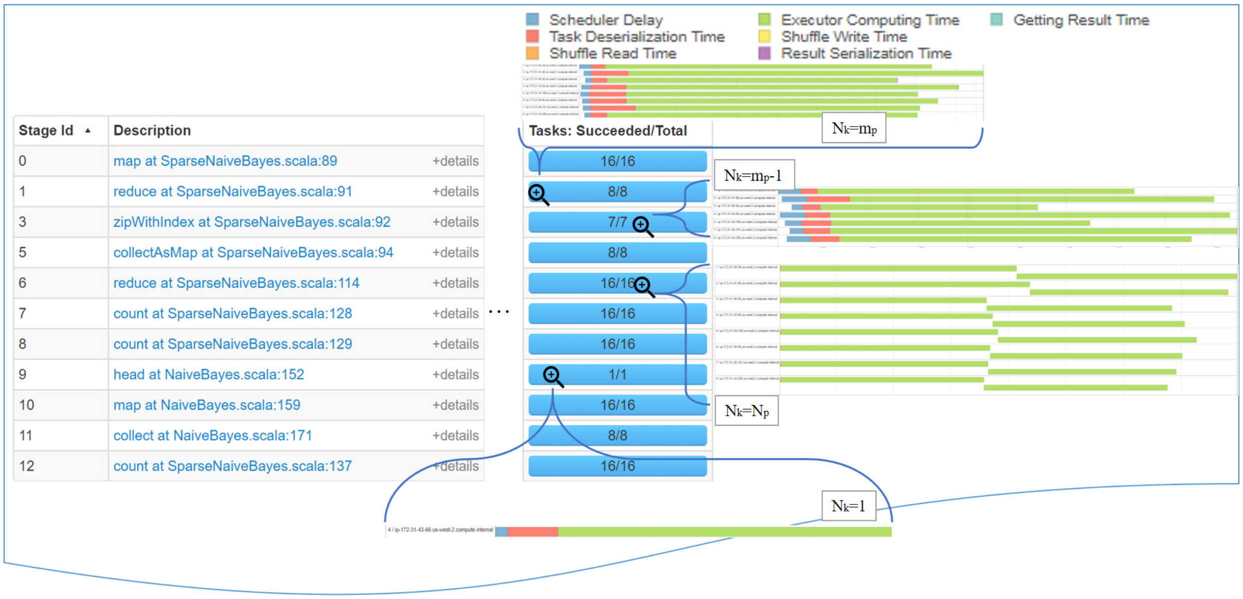
Finally, we note that USBA is a unified scaling model that renders Amdahl's and Gustafson's scaling models its special cases, i.e., a deterministic version of USBA with $L = 1$ and fixed-size workload for the Amdahl's case and fixed-time workload for the Gustafson's case.

3.3.3 Regression Model

In this paper, we focus on Spark-based big data analytics workloads. Spark is well known for its “In-Memory Computing” and outperform traditional MapReduce framework (e.g. Hadoop MapReduce) in processing large amount of data. As a result, it has been widely adopted in business applications in recent years. A Spark-based workload may contain up to thousands of processing stages, constructed by various parallel or sequential operations on RDDs (i.e., Resilient Distributed Datasets [66]) and the stages are bounded by data shuffle dependencies. These shuffle boundaries introduce barriers where a stage must wait for its previous (parent) stage to finish before it fetches the intermediate results. Each stage is associated with a certain number of tasks, which are assigned to a number of executors (an executor is a process on a worker node for computation and data storage) by the driver program. Therefore, Spark applications are by design multi-stage workloads that well match with the USBA scaling model.

In what follows, we use a machine learning benchmark from Intel Hibench Suite, i.e., Bayes Classifier (*Bayes*), to guide the discussion. The workload is a simple multi-class classification algorithm implemented in spark.mllib and uses automatically generated documents provided by the Hibench Suite as data input.

Figure 3.5 shows a workflow of *Bayes* as a serial executions of stages, which is typical for most Spark applications. In this example, $N_p = 16$, $m_p = 8$ and $L = 13$ with a stage ID assigned to each stage, as shown in the leftmost column. We first note that the number of *Succeeded/Total* tasks in column three (i.e., N_k) at each stage is closely related to either N_p or m_p. More specifically, it takes value of $N_p = 16$ (in these stages $m_k = 8$), $m_p = 8$



Stage Id ▲	Description		Tasks: Succeeded/Total
0	map at SparseNaiveBayes.scala:89	+details	16/16
1	reduce at SparseNaiveBayes.scala:91	+details	8/8
3	zipWithIndex at SparseNaiveBayes.scala:92	+details	7/7
5	collectAsMap at SparseNaiveBayes.scala:94	+details	8/8
6	reduce at SparseNaiveBayes.scala:114	+details	16/16
7	count at SparseNaiveBayes.scala:128	+details	16/16
8	count at SparseNaiveBayes.scala:129	+details	16/16
9	head at NaiveBayes.scala:152	+details	1/1
10	map at NaiveBayes.scala:159	+details	16/16
11	collect at NaiveBayes.scala:171	+details	8/8
12	count at SparseNaiveBayes.scala:137	+details	16/16

Figure 3.5: Workflow in stages for *Bayes*. The number of tasks (N_k) and executors (m_k) within each stage are closely related to N_p and m_p.

$(m_k = 8)$, $(m_p - 1) = 7$ $(m_k = 7)$, or 1 $(m_k = 1)$, depending on the actual operators taken at that stage, as listed in the *Description* column. Our experiments also indicate that L changes with N_p and m_p as well. The existence of N_k, m_k, and L dependencies on N_p and m_p is the fundamental premise underlying the following regression models.

We define the following two scaling factors,

$$p\left(N_p, m_p\right) = \frac{\mathbb{E}[T(L, \{N_k, m_k\})]}{\sum_{k=1}^{L(1,1)} \tau_k} \tag{3.12}$$

$$q\left(N_p, m_p\right) = \frac{\mathbb{E}[t_0(N_p, m_p)]}{\tau_0} \tag{3.13}$$

called *p-function* and *q-function*, respectively. In general, $p(N_p, m_p) \geq 1$ and $q(N_p, m_p) \geq 1$. Note that the above variable deduction is made possible because L, N_k and m_k (for $k = 1, ..., L$) are implicit functions of N_p and m_p. As a result, $S(L, \{N_k, m_k\}) = S(N_p, m_p)$

and Eq. (3.8) can be rewritten as follows,

$$S(N_p, m_p) = \frac{\tau_0 + N_p \sum_{k=1}^{L(1,1)} \tau_k}{\tau_0 \times q(N_p, m_p) + (\sum_{k=1}^{L(1,1)} \tau_k) \times p(N_p, m_p)} \tag{3.14}$$

since τ_k (for $k = 0, 1, ..., L(1,1)$) are easily measurable parameters at the smallest problem size $N_p = 1$ without scaling out (i.e., $m_p = 1$), the speedup $S(N_p, m_p)$ is completely determined by the *p-function* and *q-function.* In other words, if one can pin down these functions, e.g., by regression, based on measured data at small N_p's and m_p's, the scaling behaviors in terms of speedup can then be predicted at larger N_p and m_p. To this end, we propose the following solution.

Without using any structural information from the workflow, we identify the most plausible function format for *p-function* by searching in a large function space using the 3D ($\{S(N_p, m_p), N_p, m_p\ \}$) measured data as input. This approach, if successful, can be generally applied to any applications and programming frameworks that match the USBA scaling model, not limited to Spark-based data mining and graph applications under study. To this end, we resort to an open source online curve (and surface) fitting tool [44]. Using our 3D data set as input, it compares thousands of templates selected from a wide range of families (including Polynomial, Logarithmic, Exponential, Miscellaneous, Power, etc.) in parallel. Taking the results from the top of the list, we then examine the relative magnitudes of the involved parameters to further trim the function to a meaningful and simple form:

$$p(N_p, m_p) = (1 - \beta - \gamma) + \alpha \cdot log(\frac{N_p}{m_p}) + \beta \cdot m_p + \gamma \cdot N_p \tag{3.15}$$

Although Eq. (3.15) is derived without taking the underlying workflow structure into account, it is still plausible to interpret the meanings for each involved terms in this expression. The term, $(\alpha \cdot log(\frac{N_p}{m_p}))$, indicates that the effective computing time for the workload mapped to each executor increases fast at small load level, n_p, but then increases sublinearly fast as n_p becomes large. This can be understood by the fact that the initial base task execution incurs much more overhead than the following base tasks as is evident in the example given in Figure 3.8. The term, $\beta \cdot m_p$, is clearly the overheads induced by the scaling out, e.g., the communication overheads. The term, $\gamma \cdot N_p$, which is a very small negative term, is probably an artifact induced by the pseudo random nature of the synthetic data set generated by the HiBench tool itself [31].

The above expression bears some resemblance to the response time model derived from the workflow structures underlying some Spark-based applications (Equation 1 in [60]). A major difference lies in the term for per executor workload computing time. In [60], this term is given as a linear function of n_k, whereas ours is a sublinear one in log form. Comparison study of the two solutions with application to the four Spark benchmarks (not shown here) indicates that our solution outperforms the other one by a small margin. This suggests that a workflow-structure-oblivious solution like ours can perform equally well as a more elaborate workflow-structure-aware solution.

With regard to *q-function*, we find from the raw data for all four Spark benchmarks that it is only a function of m_p, not N_p and it is a strong piecewise function, which can be easily modeled using the following expression,

$$q(m_p) = \begin{cases} 1 - \epsilon + \epsilon \cdot m_p & m_p \leq \theta \\ 1 - \epsilon + \epsilon \cdot \theta & o.w. \end{cases} \tag{3.16}$$

It means that the launch overhead increases with m_p initially and then reach a threshold θ, beyond which it becomes a constant. Note that in [60], this term is assumed to be a constant.

In Table 3.1, we list the parameter values for all the parameters in *p-function* and *q-function*, together with their respective significances, i.e., the *p-values*, for both performance modeling and prediction, estimated by Wolfram Mathematica [61]. Generally speaking, *p-value* ≤ 0.05 implies the estimated value is significant. The *p-values* for all the parameters except γ's for some benchmarks indicate that they are significant and cannot be omitted. The estimated small γ values and their relatively large *p-values* also suggestions that the impact of the artifacts induced by the pseudo randomness of the synthetic data is minor.

Finally, we need to point out that there are other ways of generating the regression models that we tested. For instance, RSM (Response Surface Methodology) [22] is a widely adopted 3D curve fitting technique which is particularly useful in searching for optimal values. Unfortunately, even though the model is systematic and easy to follow, it fails at prediction with large percentage errors in our use cases.

3.4 Application of USBA

This section applies USBA to four selected Spark benchmarks from Intel HiBench Suite. We explain the experiment setup on the EC2 cloud; the measurement procedure; and the results separately.

Table 3.1: Parameter estimation for the selected Spark benchmarks' *p-function* and *q-function*.

(a) Performance Modeling

	α		β		γ		ϵ		θ	
	est.	p-value	est.	p-value	est.	p-value	est.	p-value	est.	p-value
Bayes	1.58	3.15E-11	0.051	4.90E-19	-0.006	0.008	0.044	1.58E-27	19.09	1.84E-27
RF	1.938	1.19E-13	0.074	2.89E-24	-0.009	0.000116	0.04389	2.29E-34	18.983	2.71E-34
SVM	0.807	0.006402	0.14	1.28E-33	-	-	0.037	1.50E-32	19.905	1.62E-32
NWeight	2.505	8.39E-09	0.106	6.82E-20	-0.022	1.05E-05	0.04	1.82E-29	19.516	1.83E-29

(b) Prediction ($N_p = 64$)

	α		β		γ		ϵ		θ	
	est.	p-value	est.	p-value	est.	p-value	est.	p-value	est.	p-value
Bayes	1.555	4.22E-11	0.031	0.000443	-1.91E-06	0.999713	0.050	2.41E-08	15.367	2.40E-08
RF	2.194	6.61E-09	0.085	1.77E-05	-0.021	0.067175	0.049	7.08E-10	15.612	6.93E-10
SVM	0.886	1.47E-05	0.137	2.08E-13	-	-	0.037	4.54E-16	19.783	2.08E-14
NWeight	2.067	1.09E-05	0.09	0.00136	-1.09E-07	0.999995	0.040	1.81E-14	19.044	6.69E-13

Table 3.2: Measurements of terms in numerator and denominator of Eq. (3.14) from Spark Log files.

Term	**Log File Identifier**	**Configuration**
τ_0	*stage_start_time* - *app_launch_time* (first stage)	$N_p = m_p = 1$
τ_k $(k \geq 1)$	*stage_end_time* - *stage_start_time*	$N_p = m_p = 1$
$q(m_p) \times \tau_0$	*stage_start_time* - *app_launch_time* (first stage)	$N_p \geq m_p > 1$
$p(N_p, m_p) \times (\sum_{k=1}^{L(1,1)} \tau_k)$	*Accumulate* $\{$*stage_end_time* - *stage_start_time*$\}$	$N_p \geq m_p > 1$

3.4.1 Experiment Setup

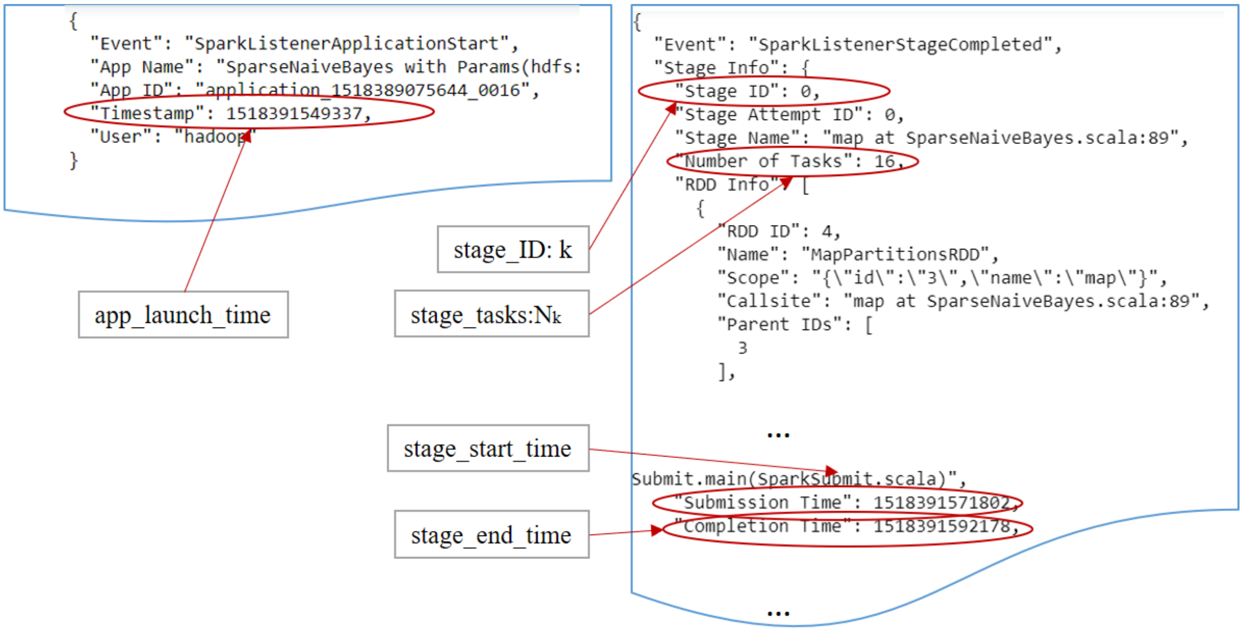
We deployed the Intel HiBench suite on EC2 with the EMR (Elastic MapReduce) service. The selected benchmarks includes three machine learning applications:*Bayes Classifier (Bayes)*, *Random Forest(RF)*, *Support Vector Machine(SVM)*, and a graph application (*NWeight*). On the hardware side, a heterogeneous cluster, composed of one master node of m4.2xlarge instance type (8 vCPU, 32GB memory) and multiple core nodes of m4.large instance type (2 vCPU, 8GB memory), is set up. All the instances are pre-configured with sufficient storage capacity (100GB per-node) and bandwidth ($\geq$ 450 Mbps). On the software side, dynamic resource allocation (i.e., performance tuning option) is turned off to prevent possible unpredictable execution behaviors, as executor reuse and logging is enabled for performance metrics collection. All the input data sets are generated by the respective data generators provided by this benchmark suite. The number of executors (m_p) are chosen as 1, 1/2, 1/4 and 1/8 in ratio to N_p, or equivalently, $n_k = 1, 2, 4$ and 8.

3.4.2 Measurement

The next step is to extract the execution latencies for all stages from the application's Log file to derive the speedup in Eq. 3.14. Table 3.2 summarizes the terms in Eq. 3.14 that need to be measured. This done by tracing the beginning and ending timestamps for each execution stage in the Spark Log files, which are available in the format of *JSON* entries. Figure 3.6 illustrates these highlighted JSON entries and the ways to extract each timestamp.

3.4.3 Results and Analysis

Using the regression model introduced in Section 3.4.2, each benchmark's speedup can be explicitly evaluated as a function of the problem size (N_p) and parallel degree (m_p). Partial results are given in Table 3.3. As one can see, the the regression-model-based speedups are within 15% errors of the measured ones for the left three data mining benchmarks. For the graph benchmark on the right, they are within 21% errors. This suggests that the subtle differences of the underlying workflow structures for different types of applications may play an important role in modeling data analytics applications.



```
{
  "Event": "SparkListenerApplicationStart",
  "App Name": "SparseNaiveBayes with Params(hdfs:
  "App ID": "application_1518389075644_0016",
  "Timestamp": 1518391549337,
  "User": "hadoop"
}
```

```
{
  "Event": "SparkListenerStageCompleted",
  "Stage Info": {
    "Stage ID": 0,
    "Stage Attempt ID": 0,
    "Stage Name": "map at SparseNaiveBayes.scala:89",
    "Number of Tasks": 16,
    "RDD Info": [
      {
        "RDD ID": 4,
        "Name": "MapPartitionsRDD",
        "Scope": "{\"id\":\"3\",\"name\":\"map\"}",
        "Callsite": "map at SparseNaiveBayes.scala:89",
        "Parent IDs": [
          3
        ],

            ...
Submit.main(SparkSubmit.scala)",
    "Submission Time": 1518391571802,
    "Completion Time": 1518391592178,

            ...
```

Figure 3.6: Log analyzer of Spark applications. As marked in the figure, for the k_{th} stage, it is tagged by the *stage_tasks:*N_k. The stage execution latency is calculated by *stage_end_time* - *stage_start_time*, and the application launch overhead is estimated by the *stage_start_time* of the first launched Spark stage subtracted by *app_launch_time*. To distinguish the cost, we mark this phase of execution as the first stage and the rest regular stages afterwards.

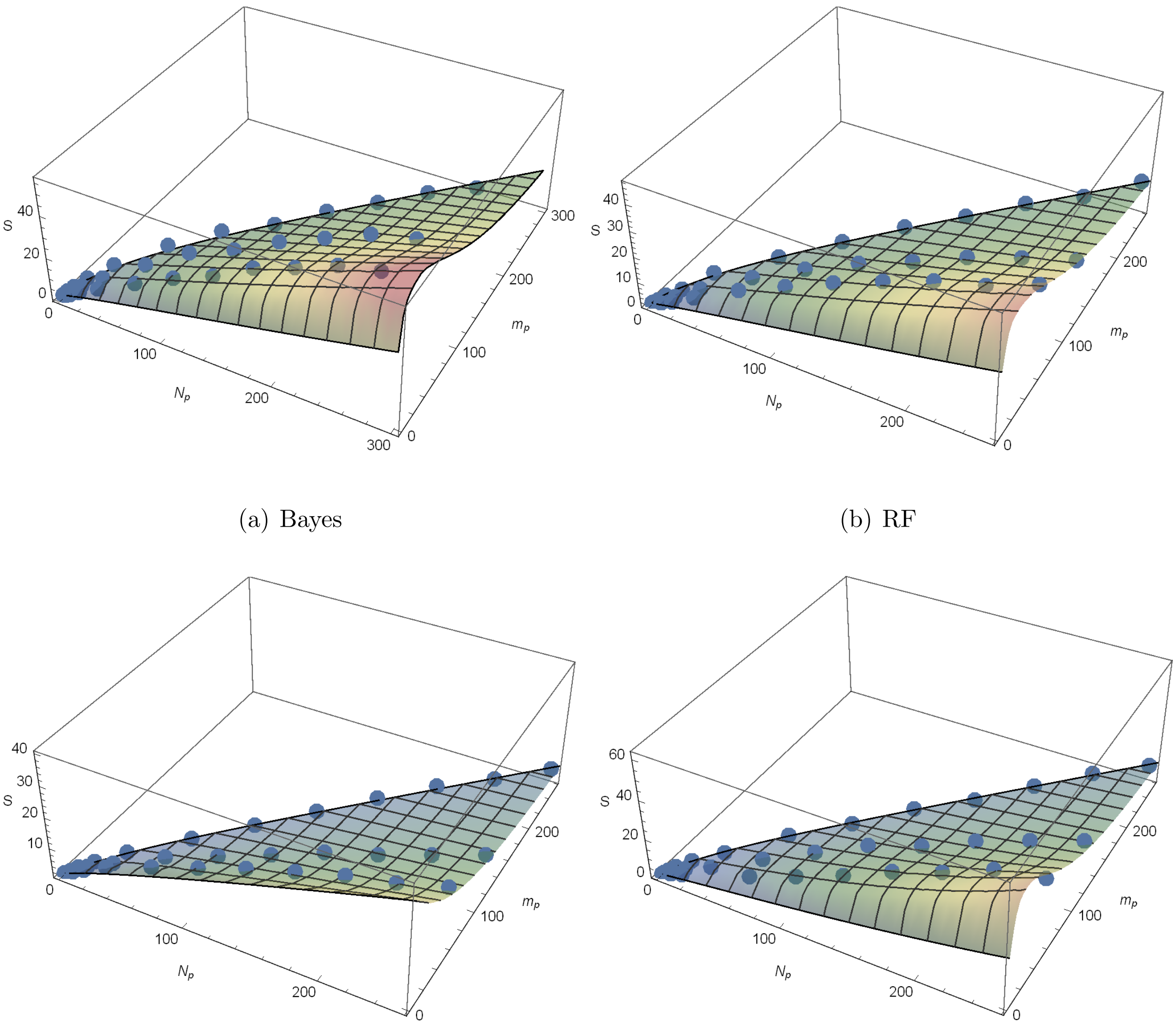
Now Figure 3.7 visualizes the complete results in the 3-D plots. As one can see, the model-based speedups closely follow the measured ones. Note that the contour in each plot has a peaked trajectory along which the speedups are maximized for the corresponding problem size N_p and degree of parallelism m_p. From job resource provisioning point of view, a job resource provisioning algorithm can be developed to identify the optimal point on this trajectory, based on the resource availability and/or monetary budget, which however, is out of scope of this paper.

Viewing clearly from this solution space, we are particularly interested in the scaling properties of the workload projected onto m_p axis and $n_p = \frac{N_p}{m_p}$ axis, corresponding to the fixed-time workload underlying Gustafson's law and fixed-size workload underlying Amdahl's law. For the first scenario (i.e. fixed-time), it's easily to identify from the figure that at given number of worker nodes or degree of parallelism (DOP), m_p, the higher the per-node workload level, $n_p = \frac{N_p}{m_p}$, the better the performance is. In other words, the speedup curve at $n_p = 4$ is higher than that at $n_p = 2$, which in turn, is higher than that at $n_p = 1$. This is due to the increased percentage of scale-out workload as the number of base tasks per executor decreases. This is evident from Figure 3.8, in which stage 8 with per-node workload level of 2 ($\frac{N_p}{m_p} = 2$) is given. It clearly shows that the scheduling and deserialization time (i.e., the communication cost) of the first round of tasks outweigh the second one. Hence, further increasing per-node workload level effectively reduces the scale-out induced workload per base task. On the other hand, for the latter scenario (i.e. fixed-size), one can find out that except for SVM, all the other applications have their speedups peaks and then fall as DOP increases. For SVM however, the speedup decreases monotonically (for $m_p \geq 3$). This, again, is in stark contrast with that predicted by Amdahl's law, i.e., the speedup monotonically increases with a fixed upper bound as given in Eq. (3.2). This is due to the scale-out induced overhead, as discussed above. Again, the optimal configuration should be one with the per-node workload level matching the node-level resource availability.

3.4.4 Prediction

While modeling is important to understand the big data application's scaling behavior, speedup prediction on the other hand is useful for the system designers to achieve desired performance while avoid wasting time and energy on over-allocating most valuable parallel computing resources.

Typically, using regression analysis or machine learning for prediction, a proper selection of the training data set is at the core of achieving the desired precision goal. In this study, we train our regression models introduced in Section 3.3.3 for the selected benchmarks at roughly 10% of total cost in terms of EC2 instance hours. The samples selected in the training set needs to cover all the scaling factors and hence, a certain threshold of the sample set size must be reached. Specifically, we use the data set for the problem size no bigger than $N_p = 64$ as the training set. This is because we find that the fourth term in Eq. (3.15) —$(\gamma \cdot N_p)$ can only be captured when the problem size exceeds 32, thus the data samples for $N_p = 64$ have to be included in the training sets for *Bayes, RF* and *NWeight.* But for *SVM*, the training set that covers only up to $N_p = 16$ is sufficient for accurate prediction. This is because for *SVM*, the fourth term in *p-function* is insignificant, as is evident from Table 3.1. This phenomenon can be possibly explained by the characteristics of synthetic input data sets. The contents of these pseudo-randomly generated data blocks inevitably repeat themselves



(a) Bayes

(b) RF

(c) SVM

(d) NWeight

Figure 3.7: The 3D unified speedup solution spaces of all four Spark benchmarks. The highlighted surfaces represent the speedup solutions while the blue dots represents the experiment data collected (taken as the average values of multiple observations). The gradually-changed color planes from cold to warm indicate the "backbone" areas where the local performance peaks can be achieved.

(in cycles) as the problem size reaches certain level. This may cause the content-sensitive analytics programs to exhibit different scaling characteristics when the problem size reaches

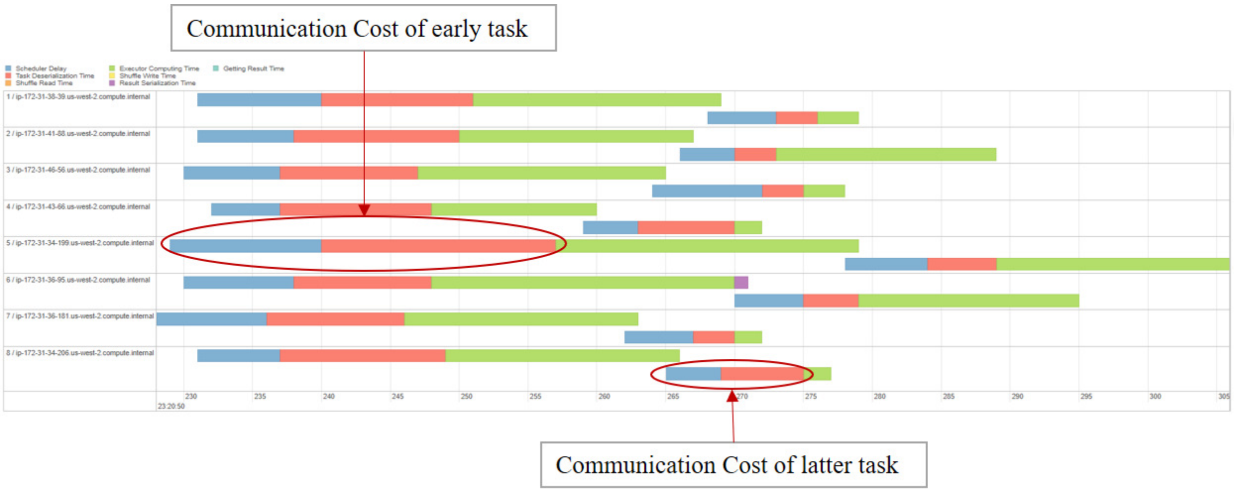
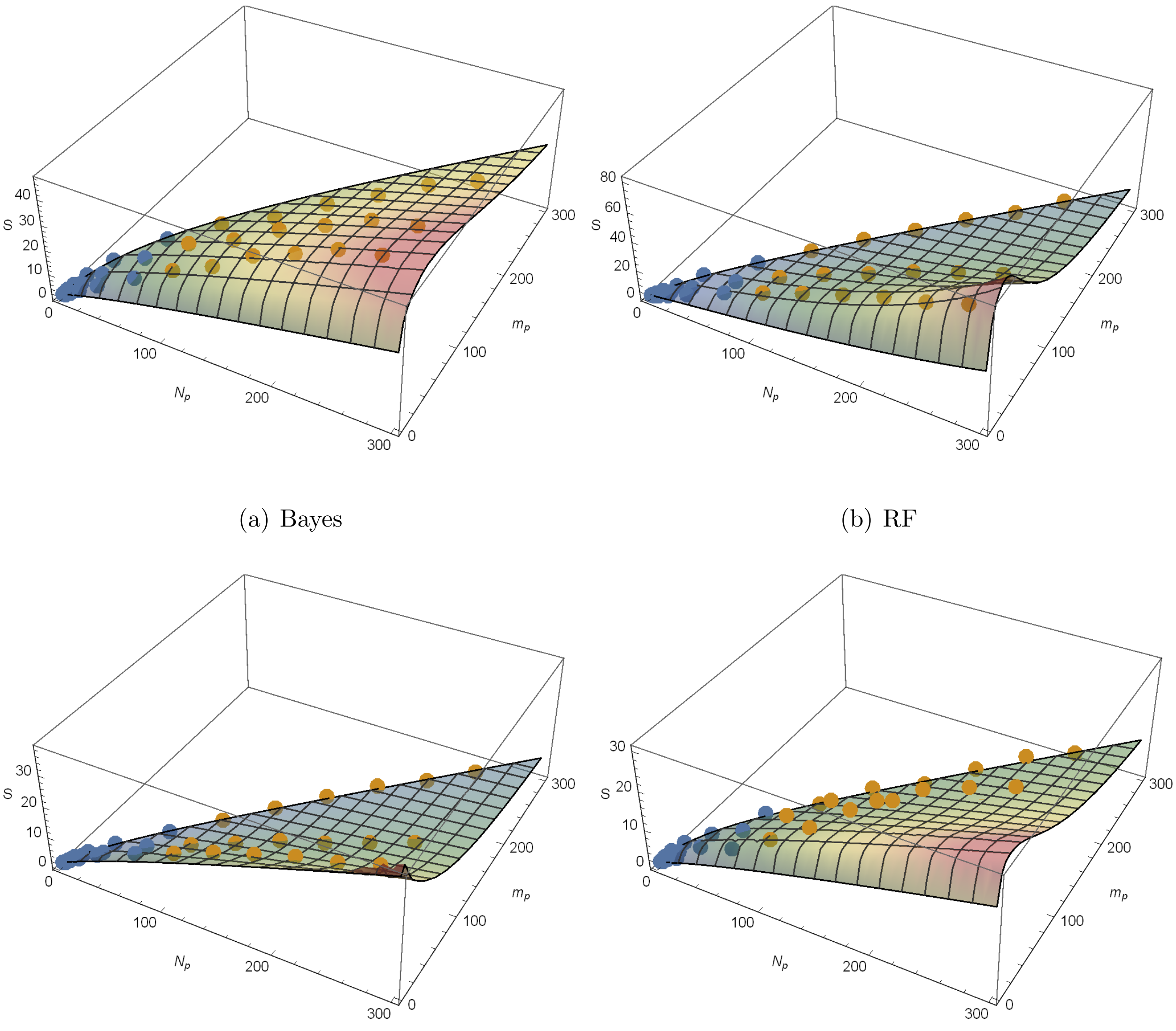


Figure 3.8: Communication and computing costs for a typical Spark applications, the parallel degree is set as 2 for this workload.

certain threshold, which cannot be predicted with the training samples from the problem size below this threshold.

Figure 3.9 demonstrates that except for a few outliers, most experimental data points are closely scattered around the prediction surface. In fact, the average prediction errors against the measured ones are constantly within 6% to 12% for the Spark machine learning benchmarks(*Bayes*, *RF* and *SVM*) and 19% for the graph benchmark —*NWeight*. The performance difference indicates the workload for *NWeight* may be more content sensitive than the rest machine learning benchmarks, and thus requires a wider training set coverage. Nevertheless, at this error range, the prediction model is sufficient to capture the speedup trending at large problem sizes. In fact, the performance of our prediction is on par with that of the workflow-structure-aware prediction model given in [60], which reports 20% prediction errors for big data analytics workloads.

In summary, USBA can be applied to the evaluation of the scaling properties for today's big data applications. It can also be used to facilitate the development of efficient job resource provisioning algorithms.



(a) Bayes

(b) RF

(c) SVM

(d) NWeight

Figure 3.9: Speedup predictions for selected Spark benchmarks. The surface plots depicts the unified speedups calculated by prediction models derived from training samples ($N_p \leq 64$), where these data collections are colored in blue. The rest samples (points colored in orange) are shown as the prediction targets.

3.5 Related Work

In this section, we first review the related work in terms of scaling modeling and then relevant job resource provisioning mechanisms.

As we have seen in the previous sections, big data analytics workloads cannot be adequately characterized by the existing scaling laws. However, to the best of our knowledge, to date, no scaling model is available that can adequately characterized such workloads. As a result, Amdahl's law is still the most widely cited law for big data analytics, e.g., [50], [58].

The lack of a scaling model does not mean that researchers are unaware of the inadequacy of the existing scaling models. In fact, as scale-out applications has emerged as the dominant applications in datacenter, rich scaling properties of such applications reveal themselves, which are well noted. Here are some examples. It was found [14] that for a fixed-size iterative computing and broadcast scale-out workload, the job stops scaling at about $n = 60$, beyond which the speedup decreases due to linear increase of the broadcast overhead, where n is the number of computing nodes for parallel processing. TCP-incast overhead was found to be responsible for the speedup reduction for many big data analytics applications [13]. Centralized job schedulers used in some popular programming frameworks, such as Hadoop and Spark, were found to pose performance bottlenecks for job scaling, due to a quadratic increase of the task scheduling rate as n increases [45]. The scaling analysis of data mining applications [39] reveals that the reduction operations in each merging phase are induced by external scaling, resulting in much lower speedup than that predicted by Amdahl's law. As we demonstrated in Section 2, even for some simple MapReduce-based applications, including Sort and TeraSort (Hadoop), their scaling properties cannot be captured by the existing scaling laws.

Some results on the improvement of the existing scaling laws are available in the context of using multicore processors for parallel computing, e.g., considering multicore hardware scaling [29], multithreading and multicore scaling [9], and impact of critical section [9, 23]. However, the system and workload models underlying these scaling models still follow the ones for the traditional scaling laws, and hence, these scaling models are not suitable for the big data analytics workloads.

Job resource provisioning solutions can be generally classified into two categories, i.e., blackbox and graybox ones. The blackbox solutions generally rely on job profiling data (e.g., [49])as input to some machine learning or datamining algorithms to predict the resource demand of a target job to be scheduled, e.g. [4]. Although applicable to a wide range of applications, a blackbox solution generally works for recurring jobs only. The graybox solution makes use of the system and workload information to a certain degree so that some modeling can be done to narrow down the design space to allow lower cost of profiling/sampling for the prediction, e.g., [60], [6]. No white-box solution is available due to scale and complexity involved in a job workflow. Although without considering detailed resources, among other things, USBA by itself serve as a job resource provisioning solution, its system, workload, and prediction models have the potential to lead to an effective graybox job resource provisioning solution. In particular, the ability to capture the performance of different workload scalings from the resource-abundant, all the way to resource-constrained scenarios makes the job resource provisioning in the largest design space of practical interests possible.

3.6 Conclusion

In this paper, we proposed a new scaling model for scale-out datacenter applications, called Unified Scaling Model for Big data Analytics (USBA). USBA is a unified scaling model that applies to the modern big data analytics based on multi-stage iterative parallel programming frameworks and takes the traditional two-stage serial-and-parallel execution model as its special case. USBA also provides a regression model for the matching and prediction of the parallel speedup for big data analytics workloads. The case studies based on four Spark-based data mining and graph benchmarks demonstrated that USBA can capture the novel scaling properties of big data analytics workloads, which cannot be predicted by the traditional scaling laws, including Amdahl's and Gustafson's laws. These case studies also demonstrated the promising prospects of USBA to facilitate effective job resource provisioning for big data analytics workloads in datacenter environments.

Table 3.3: Measured (partial) USBA speedups for Spark benchmarks in comparison with model estimations as problem size scaling from 1 to 256.

		Bayes			RF			SVM			NWeight		
N	m	exp.	USBA	RE	exp.	USBA	RE	exp.	USBA	RE	exp.	USBA	RE
1	1	1.0	1.0	0%	1.0	1.0	0%	1.0	1.0	2%	1.0	1.0	1%
8	2	2.2	2.3	4%	1.8	1.9	5%	2.3	2.2	3%	2.1	1.7	20%
8	4	3.3	2.9	12%	2.2	2.5	14%	2.4	2.2	11%	2.7	2.3	16%
8	8	4.2	4.0	4%	2.9	3.3	14%	2.2	2.0	7%	3.3	3.3	2%
64	32	17.0	15.3	10%	12.3	12.0	3%	7.3	7.0	4%	10.0	11.0	10%
64	64	15.8	13.4	15%	10.0	9.9	1%	4.6	4.9	5%	8.7	8.5	3%
96	48	23.3	19.9	15%	16.0	15.2	5%	8.0	8.3	4%	14.2	13.9	2%
96	96	18.1	15.4	15%	11.4	11.2	2%	5.3	5.4	3%	8.6	9.3	9%
128	32	23.6	26.2	11%	21.8	21.0	4%	13.5	13.1	3%	17.2	20.7	21%
128	64	25.1	23.4	7%	18.7	17.7	5%	8.8	9.2	5%	18.3	16.1	12%
128	128	16.7	16.7	0%	12.8	12.0	6%	6.1	5.7	6%	10.3	9.8	4%
192	48	34.0	35.9	6%	28.9	28.4	2%	16.7	15.8	5%	27.2	29.3	8%
192	96	31.6	28.5	10%	22.0	21.2	3%	9.6	10.4	9%	21.5	19.2	11%
192	192	18.4	18.2	1%	13.5	13.0	4%	7.2	6.1	15%	10.5	10.4	1%
256	64	38.8	44.2	14%	33.9	34.7	2%	18.5	17.7	4%	32.6	37.0	14%
256	128	32.1	32.0	0%	21.6	23.6	9%	10.4	11.1	7%	22.7	21.3	6%
256	256	16.8	19.1	14%	14.0	13.5	4%	5.9	6.3	7%	10.0	10.7	8%

Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler

Wang, Zhijun, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang.“Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler.” In Proceedings of the ACM Symposium on Cloud Computing, pp. 246-258. ACM, 2019.

Chapter 4

Pigeon

4.1 Introduction

Workload heterogeneity has been a long-standing challenge in datacenter scheduling. Jobs that differ in execution time and fanout degree have distinct requirements for scheduling. Short jobs have stringent latency requirements and are sensitive to scheduling delays; long jobs, which usually have a large fanout and high resource demands, require high-quality scheduling, e.g., improving load balance, but can tolerate some scheduling delays. While short jobs are usually user-facing applications [8,35] and important to user-perceived quality-of-service, long jobs help improve datacenter resource utilization. Therefore, it is common practice to collocate short and long jobs in datacenter management, but meeting the diverse needs of heterogeneous jobs remains a critical challenge.

Early datacenter job schedulers, e.g., Jockey [24], Quincy [34], Tetrished [59], Delay Scheduling [65], Firmament [26] and Yarn [41] are centralized by design. Centralized schedulers rely on a global view of resource availability to make scheduling decisions. As systems scale, handling a large number of jobs and collecting runtime status from a large number of nodes inevitably become a bottleneck and incur a significant scheduling delay for each job.

This is particularly problematic for short jobs with tight deadlines.

To address the scalability issue, recent research, such as Sparrow [43] and Peacock [38], employs multiple schedulers to dispatch tasks in an independent and distributed manner. Without requiring a global view of resources, distributed schedulers probe randomly selected nodes (usually twice as many as the number of tasks to be dispatched) and dispatch tasks onto the least loaded nodes. The probe based technique has been proved to greatly improve task queuing time compared to random placement [43]. However, each scheduler still needs to maintain a fairly large amount of probe related states and incurs non-negligible probe processing overheads.

Besides the above issues, the collocation of heterogeneous workloads presents unique challenges to the centralized and distributed schedulers. First, heterogeneous workloads require an effective mechanism to prioritize short jobs over long jobs. Distributed schedulers lack coordination among one another, thereby unable to enforce global service differentiation among jobs. While centralized schedulers can employ priority queues to differentiate task scheduling for different types of jobs, they are usually work conserving – low priority, long jobs can utilize the entire cluster to avoid wasting cluster resources. However, by doing so, a burst of long jobs can inflict the so called head-of-line blocking to short jobs that arrive immediately after the burst. Even in the presence of centralized priority queues, tasks from short jobs need to wait for the tasks of long jobs that have already been dispatched onto workers. Recent work BigC [10] and Karios [20] propose to suspend long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks. Second, high resource utilization in datacenters that embrace workload consolidation makes randomized load balancing less effective. For heterogeneous workloads that contain tasks of various sizes, it is difficult to identify less loaded nodes. It has also been reported that randomized load balancing is inefficient and requires multiple rounds of probing to locate idle or less busy nodes if most nodes are highly loaded [56].

Hybrid approaches, such as Mercury [37], Hawk [21] and Eagle [19], combine centralized and distributed schedulers, with former handling long jobs and the latter short jobs. However, long and short jobs are scheduled independently. This makes it difficult to mitigate the negative impact of long jobs on the performance of short jobs. For example, Eagle [19] employs two techniques to entirely eliminate the head-of-line blocking, i.e., multiple rounds of probing for short-job task placement and a reserved worker pool for short jobs. However, as the cluster load becomes high, most of the short jobs are driven by long jobs to the reserved pool [20], resulting in rapid performance deterioration for short jobs. Our simulations based on the Yahoo trace [12] show that the performance of short jobs drastically degrades, by as many as 70 times at high load compared with the non-resource-constrained case (see Section 4 for details).

In this paper, we demonstrate that a hierarchical scheduler that employs a divide-and-conquer approach in task scheduling can effectively overcome the shortcomings of centralized, distributed and hybrid schedulers, and ensure low latency for short jobs while maintaining high resource utilization without significantly sacrificing the performance of long jobs. Specially, we propose *Pigeon*, a two-layer, hierarchical scheduler for heterogeneous jobs. Pigeon divides workers in a cluster into groups and delegates task scheduling in each group to a group master. Upon job submission, Pigeon assigns the tasks of an incoming job to the masters as evenly as possible. The dispatching of tasks onto masters is intended to be simple and does not consider the type of tasks. The master in each group implements more sophisticated scheduling by maintaining two weighted fair queues, one for tasks from short jobs and the other for tasks from long jobs, respectively, and partitioning workers in each group into high and low priority workers. Tasks of short jobs can run on any workers while tasks of long jobs can only run on low priority workers. Tasks are only dispatched when there are idle workers from a group and are otherwise queued at a respective priority queue according to their types.

Pigeon is a hierarchical solution purposely designed for effective task distribution to combat heterogeneity. Pigeon's two-layer design is specially useful for heterogeneous jobs. First, it effectively mitigates head-of-line blocking of short jobs. The simple job-oblivious task dispatching among masters prevents a burst of tasks from monopolizing all workers and provides a certain level of isolation between jobs. Unlike in a centralized scheduler, where tasks of the same type (e.g., short jobs) are usually served in FIFO order, tasks of different jobs in Pigeon are evenly distributed among masters, allowing tasks that arrive late to start to execute even before some tasks of an earlier job start to execute (see Section 4.1 for details). Second, the two-layer design preserves good scalability of distributed schedulers but avoids the pitfalls of randomized load balancing. The size- and type-oblivious task dispatching among masters provide sufficient randomness for effective load balancing without global knowledge and the weighted fair queuing based scheduling within a group is deterministic, ensuring that idle workers are rapidly located to serve latency-sensitive jobs without starving the long jobs.

We perform an evaluation of Pigeon through theoretical analysis, simulation, and a prototype implementation on the Amazon EC2 cloud. Analysis results show that Pigeon can greatly increase the job-zero-queuing probability compared to Sparrow, a representative distributed scheduler, for workloads that only contain short jobs. Trace-driven simulations based on the Yahoo, Cloudera and Google traces demonstrate that Pigeon outperforms Eagle, a state-of-the-art hybrid scheduler, on short job performance by as many as tens of times in a highly loaded cluster. Experimental results on the Amazon EC2 further confirm the effectiveness of Pigeon.

4.2 Pigeon Scheduler

This section presents Pigeon. We first give an overview of Pigeon and introduce its task placement scheme, and then discuss how it handles tasks at the master level.

4.2.1 System model

We consider a datacenter cluster composed of a large number of workers, each of which can be an independent processing unit, such as a CPU core. The workers can run in parallel to execute different tasks. A key idea in Pigeon is to divide workers into groups. Each group is managed by a master which centrally controls all the tasks handled by the group and places tasks among the workers in the group. Distributed job schedulers directly distribute the tasks belonging to a job to the masters. After a master receives a task, it either directly sends the task to an idle worker to be processed immediately or puts it in the corresponding task queue if there is no idle worker in the group at the time. Figure 4.1 gives a system overview of Pigeon. The system is composed of multiple distributed job schedulers, masters, and workers. All job schedulers work independently and do not exchange any task placement

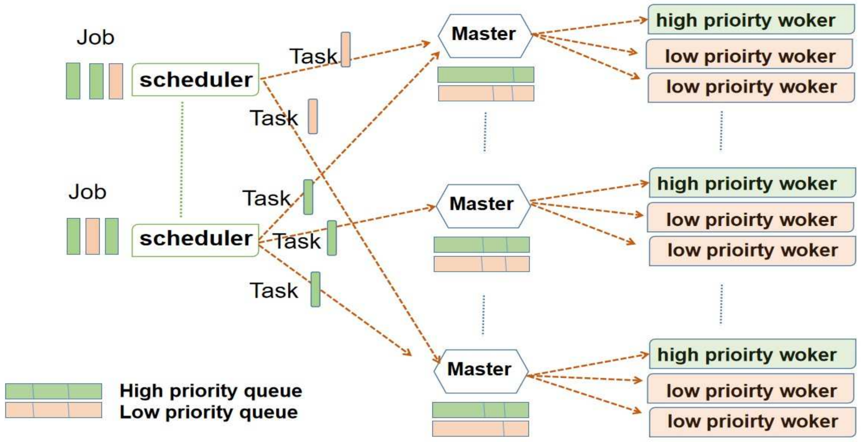


Figure 4.1: Overview of Pigeon.

information among themselves.

A master works at the task level and is mostly job oblivious except for its awareness of whether a task is a low or high priority one, based on whether the task is from a short or a long job. It maintains two weighted fair task queues, where the high priority and low priority queues store tasks belonging to short and long jobs, respectively. The classification of a job as a short or long job is handled by schedulers, based on the type of application the job belongs to. For example, user-facing applications, such as web searching and social networking, that generally have short task execution times and require stringent tail latency guarantee, can be classified as short jobs. On the other hand, background batch applications, such as data backup, that usually have long task execution times and call for loose mean response time assurance, can be classified as long jobs. In Pigeon, a small number of workers in the group, called high priority workers, are reserved exclusively for serving high priority tasks. The other workers, called low priority workers, can serve both low and high priority tasks. Since all the workers in a group are shared among tasks from short jobs in a work-conserving manner, while all the low priority workers are shared by tasks from long jobs, Pigeon can greatly improve resource efficiency, achieving high multiplexing gain, compared with the existing job schedulers that distribute tasks directly to individual workers.

A master can run in a worker who needs to be be allocated enough computation resource to effectively handle group status report and task placement functions. As we shall discuss in more detail later, in Pigeon a master needs to handle about one incoming task per second on average, which is modest from computation resource demand point of view.

4.2.2 Task Scheduling

Assume that a system has N_s schedulers and N_g groups (i.e., N_g masters). Each group has N_w workers in it. For a job with F tasks (i.e., fanout degree, F), the scheduler that handles the job will distribute the tasks as follows. It sends $S = \lfloor F/N_g \rfloor$ task(s) to each master (here $\lfloor x \rfloor$ represents the floor of x, i.e., the integer part of x) and the remaining $r = F\%N_g$ to r randomly selected masters. Since the number of workers in each group is much larger than one (i.e., in the range of 50 to 100), according to the law of large numbers, the workloads distributed to different groups are expected to be much more balanced than those distributed directly to individual workers. This helps synchronize the task processing for tasks belonging to the same job and hence, reduce the job completion time, with respect to the existing job scheduling solutions.

Two task queues of different scheduling priorities are set in each master to store the corresponding types of tasks[1], i.e., tasks belonging to short and long jobs. More specifically,

[1] Pigeon can be easily extended to support more than two job types by allocating as many priority queues as the number of job types with weighted fair queuing.

the two queues are scheduled based on weighted fair queuing with a single integer weight to ensure that tasks from the high priority queue are served with higher priority than those from the low priority queue, without starving the low priority tasks. The queue scheduler ensures that out of every W tasks to be served, at least one comes from the low priority task queue if it is not empty. The queue scheduler degenerates to strict priority queuing, when W is set to infinity. In this case, the low priority tasks can be served only when the high priority task queue is empty.

A master maintains two idle worker lists, i.e., the high and low priority idle worker lists that record all high and low priority workers that are currently idle, respectively. A task sent to a master must include the priority of the task. When a master receives a high priority task, it first checks whether the low priority idle worker list is empty or not. If the list is not empty, an idle worker from the list is removed and assigned to handle the task. Otherwise, the master checks whether the high priority idle worker list is empty or not. If it is not empty, a worker is removed from the list and assigned to handle the task. If both idle worker lists are empty, the high priority task is put into the high priority task queue. When a master gets a low priority task, it only checks the low priority idle worker list. If the list is not empty, a worker is removed from the list to serve the task. Otherwise, the task is put into the low priority task queue. Whenever a worker is selected to handle a task, the master sends the task to the worker, together with the scheduler identifier (ID) for the scheduler from which the task is received. If a master receives multiple tasks from a job at a time, it handles these tasks one by one consecutively following the same procedure.

We note that both reserving a given portion of workers in a group for high priority tasks and setting W to be a finite integer help to avoid head-of-line-blocking of short jobs and starvation of long jobs, respectively. The exact values of these two parameters must be properly selected in practice. For all our real-world-trace-driven case studies (see Section 4), we found that no more than 10% of workers need to be reserved to achieve high short job performance, lower than that of Eagle, a state-of-the-art hybrid scheduler. In the meantime, W can be simply set to infinity to achieve the highest short job performance without significantly impacting the long job performance. This is because the trace statistics show that the short job execution time is less than 20% of the overall job execution time and hence, long jobs have little chance to be starved by short jobs.

When a worker completes a task, it sends the reports/results directly to the corresponding scheduler and meanwhile, sends an idle notification message to its master. This may further trigger a task in one of the two queues to be sent to the worker or the worker to be added to the high priority worker list if it is reserved for high priority jobs, otherwise, to the low priority worker list.

4.3 Performance Modeling and Analysis

To gain insights on the Pigeon performance, in this section, we conduct simple performance modeling and analyses for Pigeon, compared with the analysis of a performance model for Sparrow [43]. To be mathematically tractable, we consider only one class of jobs and assume that the job fanout degree (i.e., the number of tasks in a job) is less than the number of groups and the number of workers in Pigeon and Sparrow, respectively. Hence, only one task queue is used in each master. In this case, all the workers serve tasks from all jobs. We focus on short jobs, which are usually more latency sensitive and whose fanout degrees are smaller than long jobs. We assume that the job fanout degrees are no larger than the number of groups.

Consider a cluster with N_g groups and each group with N_w workers, with a total of $N_c = N_g N_w$ workers in the cluster. Assume that jobs arrive following a Poisson arrival process with average arrival rate λ. All the jobs have fanout degree, F, where $F \leq N_g$, and the task execution time follows an exponential distribution with average execution time, T_e.

With the above model, each master can then be approximately modeled as running a single M/M/N_w task queue [15] with average task arrival rate $\lambda_t = \lambda F/N_g$. The worker utilization is $\rho = \lambda_t T_e/N_w$. Given that $F \leq N_g$, the probability, $P_{task}(0)$, that a task experiences zero queuing time in a group is then given as follows [15],

$$P_{task}(0) = 1 - \frac{1}{1 + (1-\rho)(\frac{N_w!}{(N_w\rho)^{N_w}}) \sum_{k=0}^{N_w-1} \frac{(N_w\rho)^k}{k!}}, \tag{4.1}$$

and the average queuing time T_q for a task in a master is

$$T_q = \frac{1 - P_{task}(0)}{N_w/T_e - \lambda_t}. \tag{4.2}$$

In this paper, a job is considered to have zero queuing time if the job completion time (not including the communication time) is equal to its longest task execution time. For example, assume that a job has 2 tasks with execution time 10s and 100s, respectively. If the job completion time is 100s, it experiences no queuing delay, even though its task with 10s execution time may have queued for some time, e.g., 50s.

Now we first consider the case that all the tasks in a job have the same execution time. Then the job-zero-queuing probability in Pigeon, $P^{PI}_{job}(0)$, can be written as,

$$P^{PI}_{job}(0) = (P_{task}(0))^F. \tag{4.3}$$

In this case, the job-zero-queuing probability for Sparrow, $P_{job}^{SP}(0)$, using $2F$ probes per job, is derived in the original paper on Sparrow [43], as follows,

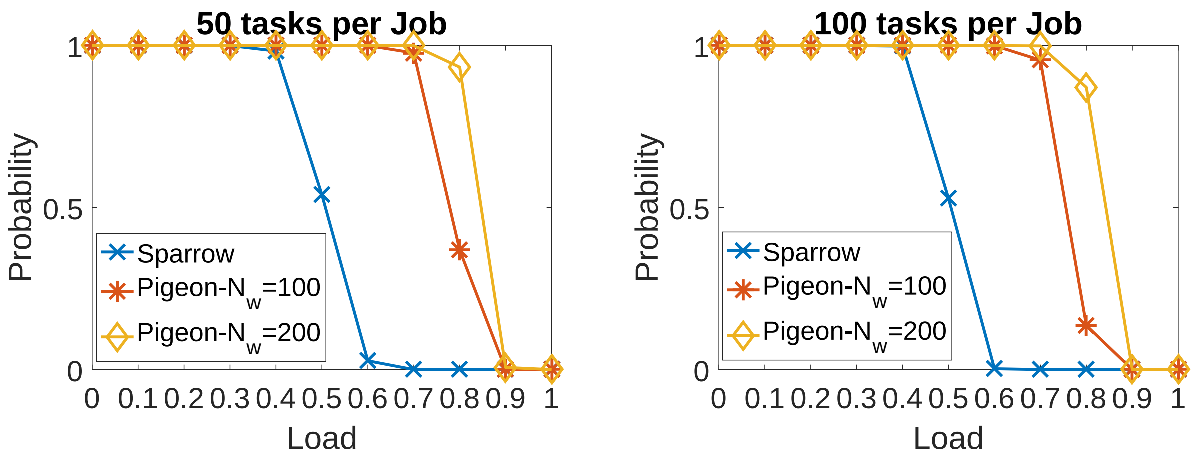
$$P_{job}^{SP}(0) = \sum_{i=F}^{2F} (1-\rho)^i \rho^{2F-i} C(2F, i), \tag{4.4}$$

where $C(2F, i)$ is the combination function.

Figure 4.3 depicts the analytical job-zero-queuing probability for Sparrow (i.e., Eqn.(4.4)) and Pigeon (i.e., Eqn.(4.3)) for two different group sizes, i.e., N_w=100 and 200 and two job fanout degrees, i.e., F = 50 and 100. As one can see, the job-zero-queuing probability for Sparrow starts to drop at load 0.4 and quickly drops to near zero at load 0.6, whereas for Pigeon, similar drops occur in a much higher load region, i.e., 0.6 to 0.8. It means that Pigeon can work at 20 - 40% higher load than Sparrow, while achieving similar job-zero-queuing performance as Sparrow, demonstrating the effectiveness of Pigeon for job scheduling, compared with Sparrow.

We also note that for Pigeon, when N_w increases from 100 to 200, the job-zero-queuing probability starts to drop at load 0.7, 0.1 higher than the former case. But it quickly approaches 0 as the load approaches 0.9, similar to the former case. This suggests that a larger group can improve performance in the medium load region (0.7 to 0.8), but not much in high load region (>0.9).

The above analyses assume that each task in a job has the same execution time. However, real trace analyses indicate that the task execution time can vary significantly from one task to another for a given job. To capture the performance impact of such variability, we consider



(a) Parallel Scaling

(b) Sequential Scaling

Figure 4.2: Job-zero-queuing probabilities for Sparrow and Pigeon with two different group sizes (N_w=100 and 200). All tasks in a job have the same execution time. (a) Job fanout F=50; (b) F=100.

the case where the task execution time for a task in a given job follows an exponential distribution.

We first calculate the average job queuing time, T_{job}. Since the job queuing time is defined as the queuing time of the slowest task of the job, we need to find the queuing time for the slowest task of the job. To this end, we observe from Figures 4.3(b)(b) and 4.4(b) that the average queuing time in Pigeon is much shorter than the average task execution time (i.e., T_e=100 ms) even at a high load (e.g., 90%). This suggests that whichever task has the largest execution time is likely to be the slowest one, regardless of its queuing time. This implies that the average queuing time for the slowest task can be simply approximated as the average queuing time for all tasks, i.e., $T_{job} \approx T_q$.

Now we calculate the job-zero-queuing probability. Consider two independent exponential distribution random variables (t_1 and t_2) with average value T_e, the joint probability density function $f(t_1, t_2) = \frac{1}{T_e^2} e^{-(t_1+t_2)/T_e}$. Then the probability of $t_1 - t_2 > T_q$ under condition $t_1 > t_2$ [53] is

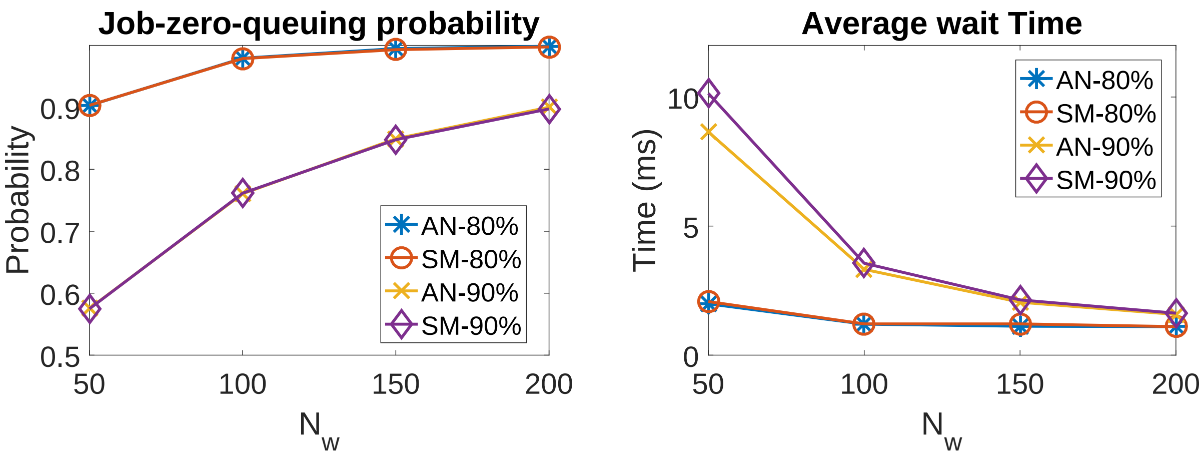
$$P(t_1 - t_2 > T_q | t_1 > t_2) = e^{-T_q/T_e}. \tag{4.5}$$

Let $A1$ and $A2$ be the tasks with the longest (t_1) and second longest (t_2) execution times in a job, respectively. Now the job-zero-queuing probability $P_{job}^{dt}(0)$ for a job with different task execution times can be approximately expressed as the probability of $A1$ with zero-queuing time (i.e., $P_{task}(0)$) while $t_1 - t_2 > T_q$, i.e., the execution time difference between the longest and the second longest task execution time is greater than the average task queuing time T_q, namely,

$$P_{job}^{dt}(0) \approx P_{task}(0) e^{-T_q/T_e}. \tag{4.6}$$

We verify the analytical approximations for T_{job} and P_{job}^{dt} by simulation. Assume that N_c=30,000, $F = 100$, and T_e=100 ms. Each task execution time follows an exponential distribution. The communication time is set at 0.5 ms between any two nodes. We note that with communication delay, the average job waiting time T_w is no longer equal to the average queuing time, but rather the average queuing time plus the communication time.

We study the Pigeon performance by changing N_w from 50 to 200 (the total number of workers N_c in the system is fixed). Figure 4.3(b) depicts the job-zero-queuing probability and the average job waiting time at two different high loads (i.e., 80% and 90%). We note



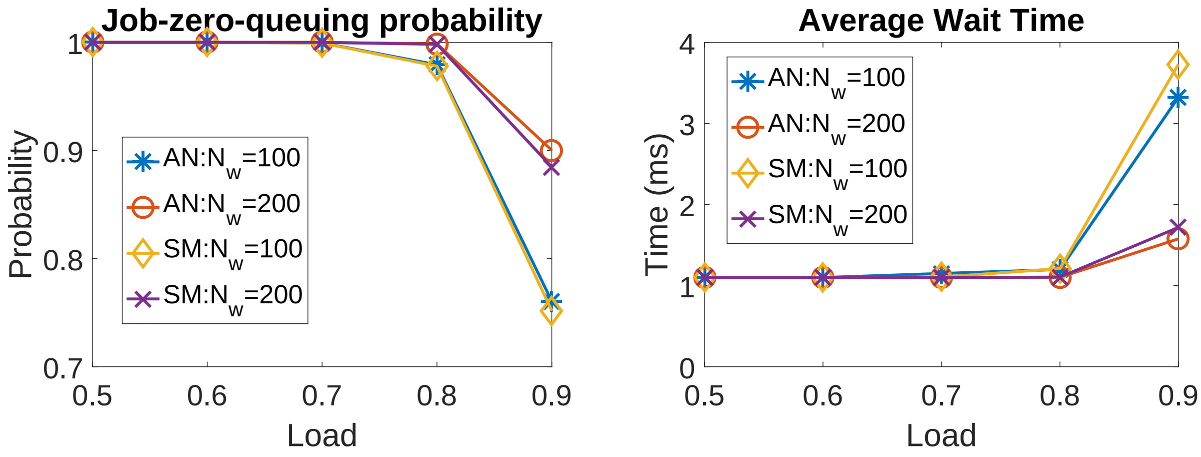
(a) Parallel Scaling

(b) Sequential Scaling

Figure 4.3: Analysis (denoted as AN) vs Simulation (denoted as SM) at different group sizes. (a) Job-zero-queuing Probabilities (b) Average job wait time.

that the simulation results (denoted as SM) closely match the analytical ones (denoted as AN), e.g., less than 1% for the job-zero-queuing probability for all N_w's tested. The largest difference is about 12% for the average waiting time at N_w=50 and the load of 90%. In this case, the simulated waiting time (also queuing time) is longer than the analytical one because the analytical results only consider the waiting time for the task with the longest execution time. As the job-zero-queuing probability is low (below 60%), the contribution of other tasks may not be neglected, resulting in larger errors.

The results verify that Eqns. (4.2) and (4.6) can be used to estimate the performance of Pigeon for handling jobs with fanout degrees less than the number of groups. The results indicate that the job-zero-queuing probability increases and the average waiting time decreases as the group size increases. It means that a larger group can provide better performance,



(a) Parallel Scaling

(b) Sequential Scaling

Figure 4.4: Analysis (denoted as AN) vs Simulation (denoted as SM) results with various cluster loads. (a) Job-zero-queuing probabilities (b) Average job wait time.

particularly from 50 to 100. The performance improves slower as the group size increases from 100 to 200, particularly for the average waiting time. Further increasing the group size is expected to offer marginal performance gain. This result provides some insight on how to set the right group size when a cluster handles jobs with small fanout degree (i.e., the number of tasks in a job is less than the number of groups in the cluster).

Now we study the performance of Pigeon by varying cluster loads. Two cases with N_w set at 100 and 200 are studied. The results are given in Figure 4.4. Again, the simulation results closely match the analytical ones. The results indicate that the job-zero-queuing probability is close to 1 even at load 80% and reduces to 0.7 at load 90%. This means that most jobs do not need to be queued even at high load, hence offering high probability of meeting the tightest job performance requirements at high load. We also note that the average waiting time is very small (less than 4%) compared to the average task execution time even at very

Trace	F_{max}	F_{min}	F_{avg}	T_e^{max} (s)	T_e^{min} (s)	T_e^{avg} (s)
Yahoo	5900	1	39.91	21259.9	1.54E-5	118.78
Cloudera	51834	1	272.93	97941.8	3.89E-5	162.19
Google	49960	1	35.32	774922	1E-6	661.74

Table 4.1: Trace statistics of job fanout and execution time

high load, e.g., 90%. These results clearly demonstrate the effectiveness of Pigeon for job scheduling.

The following two sections test the efficiency of Pigeon by large-scale simulation and on a small EC2 cloud cluster, respectively.

4.4 Simulation Testing

To test the scalability and efficiency, we use simulation to evaluate the performance of Pigeon against Eagle[2] in large clusters, using three real-world traces as input, i.e., Yahoo [12], Cloudera [11], and Google traces [48]. The open source simulation code of Eagle [19] is used and an event-driven simulator is developed for Pigeon.

Table 4.1 provides the statistics of these traces, including the maximum/minimum/average job fanout degrees (denoted as $F_{max}/F_{min}/\ F_{avg}$) and maximum/minimum/average task

[2] As Eagle outperforms Sparrow and Hawk [21], only Eagle is compared here.

execution time (denoted as T_e^{max}/T_e^{min} $/T_e^{avg}$). We see that the job fanout degree ranges from 1 to 51834; the execution time varies from microseconds to over 700K seconds; and the average task execution time ranges from 118.78 seconds to 661.74 seconds. Unlike the modeled workload in the previous section, these statistics indicate that the job size in terms of both fanout degree and task execution time vary significantly from job to job in practice. Such job heterogeneity makes it difficult to meet service requirements for individual applications, e.g., in terms of providing job completion time or throughput guarantee. For example, for a cluster with 10K workers and a long job with fanout degree of 50K, each worker needs to execute 5 tasks for the job on average. The placement of such a job evenly among all the workers in the cluster can take up all the cluster resources at once, causing head-of-line blocking to the upcoming short jobs. As aforementioned, to effectively deal with the job heterogeneity issue, both Pigeon and Eagle [19] reserve a subset of workers to be used by short jobs only, at the group-level and cluster-level, respectively. In what follows, we first discuss the parameter settings, in terms of the short-vs-long job thresholds, the reserved worker pool size, the communication delays, the group size, and the weight value for weighted fair queuing and then performance evaluation.

4.4.1 Parameter Settings

Short Jobs vs Long Jobs: As mentioned earlier, in practice, a scheduler may rely on whether a job belongs to a user-facing application to classify it as a short job or not. However, due to the lack of the application information for the three traces and to fairly compare against Eagle, for Pigeon, we simply use the same short job cutoff times, defined as the average task execution time of a job, as those used in Eagle, i.e., 90.5811, 272.783 and 1129.532 seconds for the Yahoo, Cloudera and Google traces, respectively.

Reserved Worker Pool Size: The actual number of workers reserved for tasks of short jobs has significant impact on job completion times for both short and long jobs. The more workers are reserved, the smaller the job completion time for short jobs but the larger the job completion time for long jobs. We study the performance using the three traces by varying the worker reservation ratio (due to page limitation, the results are not presented here). By taking into account of the performance for both short and long jobs, we decide to set the reservation ratios at 2%, 7% and 9% for the Yahoo, Cloudera and Google traces, respectively. For Eagle, against which Pigeon is to be compared in the following section, we set the reservation ratios for the three traces at the same values as those used in [19], i.e., 2%, 9% and 17%, respectively.

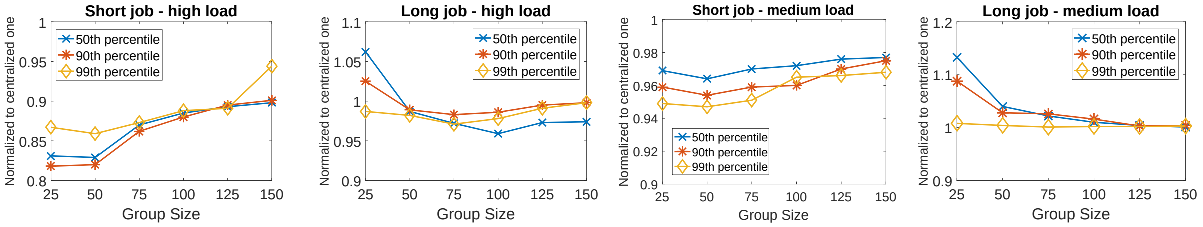
The reservation ratio that gives the best performance tradeoffs for Pigeon is between 2%-9% for the three traces. We also found that setting this ratio at 5% for all the traces leads to a maximal performance deviation from the best tradeoffs within 20% for both short and long jobs. Hence, to address the possible lack of the traces in practice, the ratio can be initially set at 5% and then adjusted as the trace workload runs long enough to estimate the best ratio.

Weight Value for Fair Queuing: The weight value W is an important parameter for Pigeon. A smaller (larger) W helps improve the performance of long (short) jobs at the cost of the other. We study the Pigeon performance by varying W from 5 to 100 and compared to that with strict priority queuing (i.e. W is set to infinity) (again, the results are not presented here due to page limitation). We find that the short job performance becomes very sensitive to W at high cluster loads when W gets below 20. For example, while the 99th-percentile short job completion time at $W = 20$ is within 140% of that at $W = \infty$, it increases to more than 300% at $W = 5$, at high cluster loads for all the three traces. Meanwhile, we find that the long job performance is insensitive to W in a wide range, e.g., only 2% difference from $W = 10$ to ∞ at all cluster loads for all the three traces. In other words, no long job starvation occurs even at $W = \infty$ for all the three traces. Hence, for all the cases studied, we set W in the range of 20 to ∞.

Communication Delays: The communication delays are set at 0.5 ms between any two nodes, i.e., a scheduler and a master, a master and a worker, or a worker and a scheduler.

Group Size: Without knowing the exact processing overhead per task scheduling at each master, we have not taken this overhead into account in both performance modeling in the previous section and the simulation in this section. As a result, intuitively, one would expect that the testing results in both previous and this section will be always in favor of larger group size, with the group size equal to the cluster size offering the highest performance (i.e., the case when Pigeon degenerates to a centralized scheduler). While this intuition is confirmed in the previous section based on the results from an ideal model, much to our surprise, it turns out to be false as confirmed by the simulation results in this section. More specifically, we can conclude that *Pigeon with the group size in a finite range actually outperforms its centralized counterpart, even when the centralized scheduler incurs negligible processing overhead.* The implication of this is significant. It means that one can no longer assume that so long as it is free from scalability concerns, centralized scheduling is always the best choice, as it has a global view of the cluster resource availability. In what follows, we first identify the range and the preferred group size, and then provide an explanation of why this seemingly counter-intuitive phenomenon can happen.

We compare the Pigeon performance at different group sizes, using the Cloudera trace as input for the simulation (similar results are obtained for the Yahoo and Google traces and hence are not given here). We consider the cluster size of 12K and 18K, corresponding to high (about 93%) and medium (about 62%) cluster loads, respectively. All other parameters pertaining to Pigeon are the same as those given above. The 50th, 90th and 99th percentiles of the short and long job completion times are used as performance metrics.



(a) Parallel Scaling (b) Sequential Scaling (c) Sequential Scaling (d) Sequential Scaling

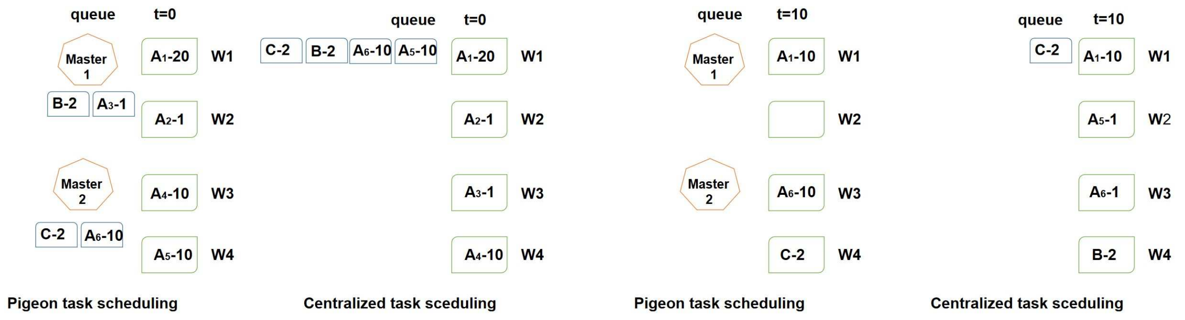
Figure 4.5: Pigeon performance at different group sizes (Pigeon is normalized to the centralized scheduler).

From the results depicted in Figure 4.5 (normalized to the centralized one), we can see that Pigeon performs better than its centralized counterpart for all the three performance metrics for short jobs, particularly at the high load (Figures 4.5 (a)). At high load, the short job performance gets better as the group size reduces from 150 to 50 and then becomes slightly worse as it further reduces to 25. The largest performance gains for short jobs are about 17%, 18% and 14% for 50th, 90th and 99th percentile job completion times at group size 50, compared to the centralized one, respectively. Similar results with smaller gains are observed at the medium cluster load (Figure 4.5 (c)). For long jobs at high load (Figure 4.5 (b)), the performance is better (worse) than the centralized one when group size is above (below) 50. The three percentiles of job completion times decrease when the group size reduces from 150 to 100, and then increase when the group size further reduces. In the medium load (Figure 4.5 (d)), the performance for long jobs is worse than that of the centralized one in the entire group size range studied. All the three percentiles of long job completion time decrease as the group size increases. The above results indicate that Pigeon is not only more scalable but also performs better than its centralized counterpart for handling heterogeneous jobs, particularly at high cluster loads.

Although the optimal group size may be workload dependent (such as the ratio between the number of short and long jobs, the task execution time distribution, etc.), based on the above observation, which agree with the observation made for the other two traces, and consistent with the analytical results for jobs at small fanout (i.e., the multiplexing gain is small when the group size is over 100), we can safely conclude that in general, the performance is insensitive to the group size in a wide range, i.e., between 50 and 100. Hence it suffices to set group size anywhere in between 50 and 100 and we set the group size at 100 for all the cases studied in this section.

Explanations for the Counterintuitive Phenomenon: A key observation we make is that this phenomenon may occur when both job fanout degree and task execution time vary in a wide range, which is the case in practice (see Table 4.1) but not for the model in the previous section (that explains why we did not observe this phenomenon there). The best way to see why this is true is by example.

Consider job scheduling for a single type of jobs and a cluster of 4 workers. At time 0, all the workers are idle and job A with 6 tasks (called as tasks A_1, ..., A_6) arrives, with task execution times of 20, 1, 1, 10, 10 and 10 units. Immediately following it are two other jobs B and C, each having 1 task with execution time of 2 units. We further assume that there is no processing overhead and the communication time can be neglected. Now we compare the performance of a Pigeon scheduler and its centralized counterpart.



(a) Parallel Scaling

(b) Sequential Scaling

Figure 4.6: Task scheduling example: (a) tasks at time 0; (b) tasks at time 10. X-t at a worker or a queue: X is the task name, and t is the remaining task execution time.

First consider a Pigeon scheduler, where 4 workers are divided into 2 groups with 2 workers each. Upon the arrival of jobs A, B, and C, in that order, the first 3 tasks from A (i.e., A_1, A_2 and A_3 with execution times 20, 1 and 1) are sent to group one and the other

3 tasks from A (i.e., A_4, A_5 and A_6 all with execution time 10) to group two. Then the task from job B is sent to group one and the task from job C to group two. At time 0, in group one, two tasks A_1 and A_2 with execution times 20 and 1 are served by workers 1 and 2, respectively; and in group two, workers 3 and 4 serve tasks A_4 and A_5, respectively. The tasks at time 0 in Pigeon are shown in Figure 4.6 (a). At time 1, worker 2 completes the task A_2 and immediately starts serving the task A_3. It finishes the task A_3 at time 2 and then serves the task from job B which is completed at time 4. Hence job B is finished at time 4. At time 10, workers 4 and 5 complete the tasks A_4 and A_5, and then serve the tasks A_6 and C. The tasks in Pigeon are now given in Figure 4.6 (b). The task from job C is finished at time 12, so job C is completed at time 12. As tasks A_1 and A_6 are finished at time 20, so job A finishes at time 20. The job completion times in Pigeon for the three jobs are 20, 4 and 12, for a total of 36 units.

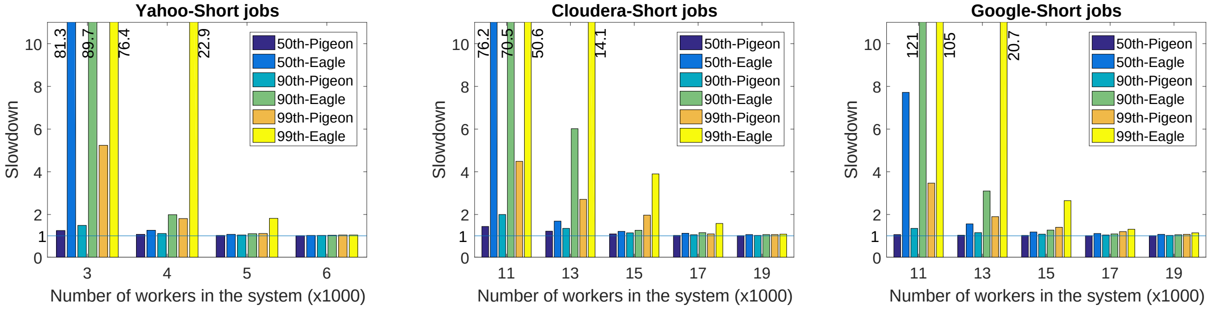
Now, consider a centralized scheduler. The first 4 tasks (i.e., A_1, A_2, A_3 and A_4) from job A are sent to workers 1-4 at time 0, respectively, as given in Figure 4.6 (a). At time 1, workers 2 and 3 complete their tasks and then serve the other two tasks (i.e., A_5 and A_6) from job A. At time 10, worker 4 finishes its task and then serves the task from job B which will be completed at time 12. So job B is completed at time 12. The tasks at time 10 is given in Figure 4.6 (b). At time 11, workers 2 and 3 complete their tasks, and then worker 2 serves the task from job C which is completed at time 13, and hence job C is completed at time 13. Task A_1 finishes at time 20, and hence job A finishes at time 20. So the job completion times for the three jobs are 20, 12 and 13, respectively, for a total of 45 units, 9 units or 25% more than the Pigeon scheduler!

From the above example, we see that for centralized scheduling, a job with a large fanout degree (i.e., job A) causes head-of-line blocking of the following jobs of the same type, even when their fanout degrees are low (i.e., jobs B and C). The head-of-line blocking caused by the same type of jobs may be alleviated by enabling task preemption [3,8], which however, may incur significant preempting overhead, particularly for resource-intensive tasks.

In contrast, for Pigeon, the tasks for jobs are distributed to different groups. This enhances the chance for tasks from later jobs to be served before tasks from the earlier jobs due to heterogeneous task execution time distribution. This helps reduce the chance of head-of-line blocking of jobs with small fanout degrees by jobs with large fanout degrees, hence, resulting in better overall performance. While helping more in alleviating head-of-line blocking by dispersing the tasks of a job with a large fan-out degree to more groups, using a smaller group size reduces multiplexing gain. This help explain why Pigeon gives the overall best performance at the group size in a certain range, i.e., 50 to 100.

Master Workload Estimation: Finally, with the parameters given above, we can now estimate the offered task load at a master. Assume that the cluster size, N_c=20,000, and hence, the total number of masters, N_g=200, given the group size, N_w=100. The real trace statistics in Table 4.1 suggest that the average task execution time is more than 100s (from 118s to 661s, to be exact). It means that a master needs to handle about only 1 task per second on average (or equivalently, 1 task per 100 seconds per worker), this overhead is negligible. In the case of a long job with a huge number of tasks, such as a job with 50,000 tasks, each master will see a burst of task arrivals of size 250. This is in stark contrast with a distributed scheduler, who needs to generate and dispatch 50,000 tasks. This example clearly indicates that the resource demand on a master is modest and a single worker should be sufficient to serve as a master, which consumes only 1% (i.e., 1 out of 100) of the total worker resources in the cluster. This means that indeed, Pigeon is a highly scalable solution.

In practice, to save the cluster resource, a master may run in a regular worker as long as the worker has enough resource to act as both a master and a regular worker. An alternative is to allow a worker to run multiple masters. For example, consider a system with 10,000 workers and each group with 100 workers. We may use 10 workers, each hosting 10 masters, instead of 100 workers with one master each, hence, cutting the overhead from 1% to 0.1%.

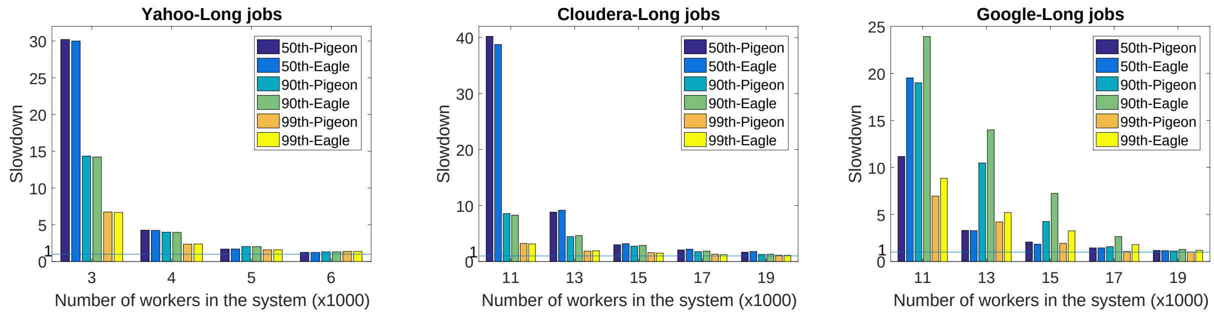


(a) Parallel Scaling (b) Sequential Scaling (c) Sequential Scaling

Figure 4.7: Short job completion time, $W = 20$.

4.4.2 Performance evaluation

The number of workers in the whole cluster is used as a tunable parameter to adjust the load level. We use 50th, 90th and 99th percentile job slowdowns with respect to the case of *unlimited resources* (i.e., the case with zero communication time and zero task queuing time)



(a) Parallel Scaling

(b) Sequential Scaling

(c) Sequential Scaling

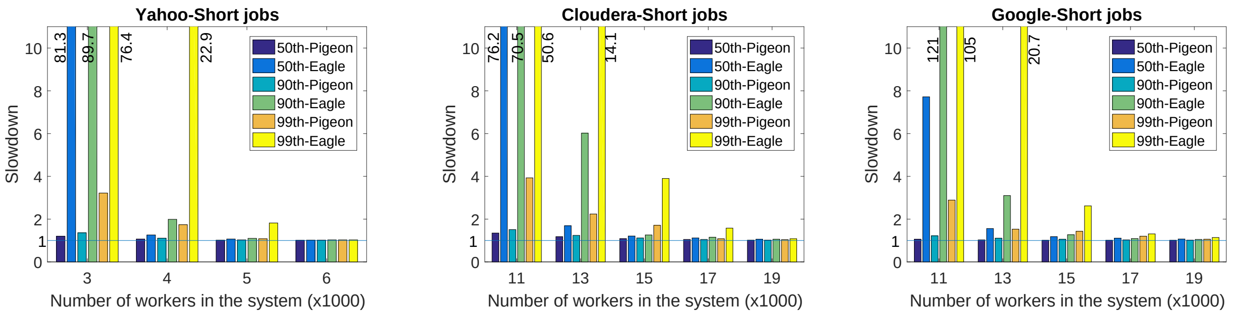
Figure 4.8: Long job completion time, $W = 20$.

for both short and long jobs as performance metrics. More specifically, the xth-percentile short/long job slowdown is defined as the xth-percentile short/long job completion time divided by the xth-percentile short/long job execution time. Here a job execution time is defined as the largest task execution time among all the tasks in the job.

Figures 4.7 and 4.8 give the slowdowns of the 50th, 90th and 99th percentiles of short and long jobs for all the three traces. Here W is set at 20 in Pigeon. First, we note that at the fixed job arrival rate, as the number of workers in the cluster increases, the slowdowns of the two schedulers converge and approach 1 for both short and long jobs. This is expected, because as the cluster size becomes larger, or equivalently, the load becomes lighter, all the jobs experience smaller queuing delays and hence, smaller job completion times, regardless what scheduling mechanism is used. Hence, it is more interesting and important to focus on small cluster sizes or high load regions. As the cluster size reduces, we can see that remarkable performance gaps between the two emerge.

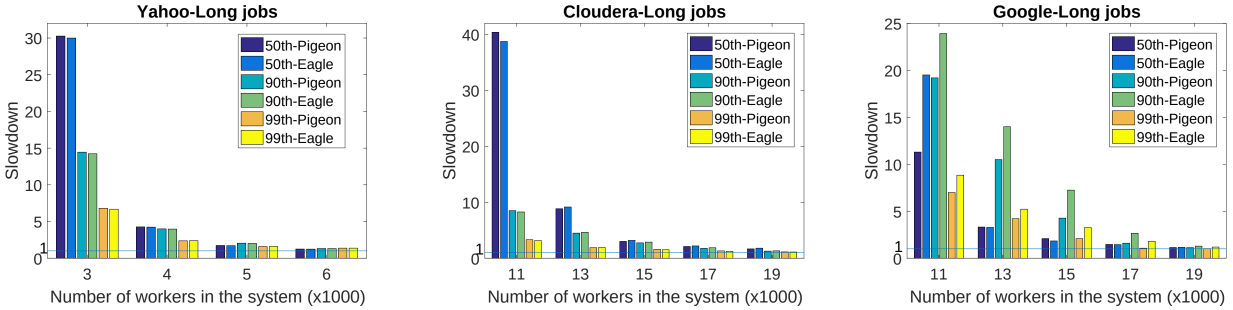
In the case of the Yahoo trace, at the cluster size of 3K, the slowdowns for short jobs in Pigeon are about 1.3, 1.5 and 5.3 times which indicates the queuing times are less than one job execution time for the 50th and 90th percentiles, and just above 3 job execution times for the 99th percentile. The results indicate that Pigeon achieves excellent short job performance even at very high cluster loads (about 95%). In contrast, the slowdowns for Eagle are above 70 times for all the three percentiles, implying that for Eagle, the queuing times are more than 70 job execution times for short jobs. Similar results can be found with the Google and Cloudera traces as shown in Figures 4.7(b)-(c). In what follows, we explain why Pigeon outperforms Eagle by such big margins.

Eagle improves over Hawk, as detailed in [19], by allowing workers who are handling long jobs to reject the probes coming from distributed schedulers who handle short jobs. This allows a distributed job scheduler to issue more rounds of probes to discover workers that are not handling long jobs, hence alleviating the head-of-line blocking effect for short jobs. However, most lower priority (i.e., non-reserved) workers can still be blocked by the long jobs, either at high load or whenever a long job with a large fan-out degree arrives. In this case, after multiple rounds of random probing, most of the tasks from short jobs are forced to be served by the high priority (i.e., reserved) workers, which however, may become the bottlenecks themselves. For example, for the Yahoo trace, consider the case of a cluster with 3K workers and 60 high priority workers (2% as set in Eagle [19]) for short jobs. When a long job with 5900 tasks (i.e., the maximum number of tasks in a job for the Yahoo trace) arrives, each low priority worker has to serve, on average, about 2 tasks of the job. After the tasks of the long job are placed, all the upcoming short jobs following this long job are forced to be served by only 60 high priority workers after a number of rounds of probing. In other words, all the low priority workers are blocked by the long job, hence resulting in big job completion time for short jobs. The key difficulty is that as a hybrid scheduler, Eagle distributes tasks from short and long jobs independently by distributed and centralized schedulers, respectively.



(a) Parallel Scaling (b) Sequential Scaling (c) Sequential Scaling

Figure 4.9: Short job completion time, $W = \infty$.



(a) Parallel Scaling (b) Sequential Scaling (c) Sequential Scaling

Figure 4.10: Short job completion time, $W = \infty$.

In contrast, Pigeon allows centralized scheduling of tasks coming from both short and long jobs and full resource sharing at the group level. This makes it possible for Pigeon to largely remove head-of-line blocking without starving the long job through weighted fair queuing and worker reservation. Again, consider the above example where a long job with 5900 tasks arrives at a cluster with 3K workers. Assume that the workers are divided into 30 groups of 100 each with 2 (i.e., 2%) workers reserved for the tasks from short jobs. Now about 197 (i.e., 5900/30) tasks from the long job are sent to each group. In a given group, the master dispatches as many tasks out of 197 to the available low priority workers as possible and the rest to the low priority queue, e.g., with 10 to the available low priority workers at the load of 90% (i.e., about 90% or 88 out of 98 are currently busy) and 187 queued. The upcoming tasks of short jobs are either served by an idle reserved worker or queued in the high priority queue. However, in addition to the 2 high priority workers, whenever a low priority worker becomes idle, it will first have high chance (19/20 at $W = 20$) to serve a task from the high priority queue. Unlike Eagle, most of the long tasks (i.e., 187) are not queued at the low priority workers, but centrally at the master, high priority tasks following these low priority tasks will not be blocked by the latter from accessing the low priority workers. Moreover, a task at the head of the high priority queue is likely to find an idle low priority worker soon, because the probability that one out of 98 busy lower priority workers will finish its task in the near future is high. This explains why Pigeon can significantly outperform Eagle in terms of short job performance, especially in the high load region.

The fact that Pigeon performs slightly better than Eagle even for long jobs, despite the use of the weighted fair queuing for short jobs over long ones, as depicted in Figure 4.8, can be explained as follows. First, Pigeon generally reserves a smaller number of workers for short jobs than Eagle (i.e., 9% vs. 17% and 7% vs. 9% in the cases of the Google and Cloudera traces, respectively and 2% vs. 2% in the case of the Yahoo trace), hence allowing more workers to be used by the long jobs. This explains why overall Pigeon outperforms Eagle in the cases of the Google and Cloudera traces but not as much in the Yahoo trace. Second, for all the real traces studied, the overall execution time for short jobs constitutes less than 20% of the total job execution time, implying that the possible negative impact of giving high priority to short jobs (i.e., letting W=20) on the performance of long jobs is quite limited.

We also test the effect of W by comparing the setting of $W = \infty$ against that of W=20. The results are given in Figures 4.9 and 4.10. We can see that only the short job completion times at very high load are different. For example, when W changes from 20 to ∞, the slowdowns of the 50th, 90th, and 99th short job completion times for the Yahoo trace are reduced from 1.3, 1.5 and 5.3 to 1.2, 1.4 and 3.6, respectively, in a cluster with 3K workers. while the corresponding slowdowns for long jobs are within 2%. The results indicate that the performance of Pigeon is indeed insensitive to W, in the range of [20, ∞].

The above results clearly demonstrate that Pigeon is a much more effective job scheduler than Eagle in terms of both design complexity (e.g., without probing phase, without having to run two different types of schedulers, and no worker involvement of scheduling) and performance.

4.5 Performance Evaluation on EC2 Cloud

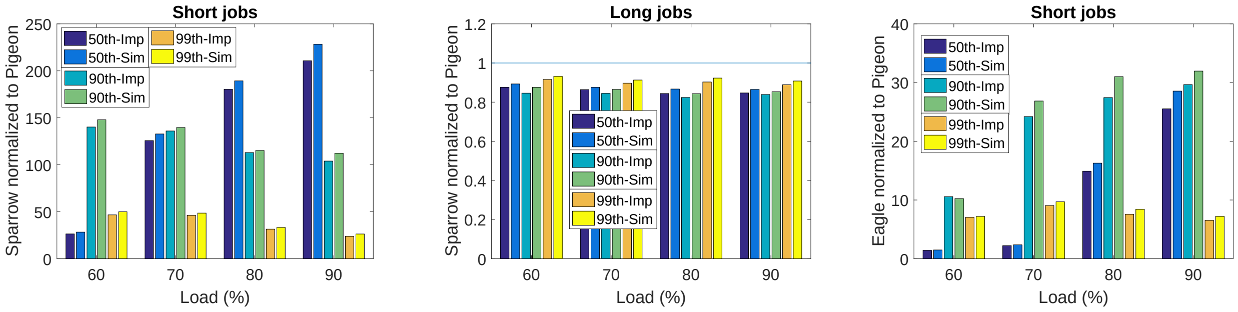
In this study, we compare the performance of Pigeon against both Sparrow [43] and Eagle [19], the state-of-the-art distributed and hybrid job schedulers, respectively, in a small cluster on the Amazon EC2 cloud. The Pigeon implementation includes two parts: the Pigeon scheduler code and the Spark plug-in. Distributed Pigeon schedulers are concurrently deployed at the application frontends, exposing services to allow the framework to submit job scheduling requests using remote procedure calls (RPCs). All RPCs for internal communications between modules of a Pigeon scheduler are defined with Apache Thrift [?]. We directly run the available open source implementation codes for Sparrow [43] and Eagle [19]. m4.large instances are used to serve as workers, masters and schedulers.

The cluster is composed of 10 schedulers and 120 workers. For Pigeon and Eagle, 10% of the workers are reserved for short jobs. In Pigeon, the workers are divided into 3 groups with 40 workers each. One worker in each group is selected as a master and W is set to infinite (i.e., each master runs two strict priority task queues). A sample job trace including 5000 jobs is extracted from the Google trace. The task execution time is scaled to the range of 10ms to 100s and the job fanout degree is scaled to the range of 1 to 100. The short job cutoff time is set at 1s. It turns out that 10% jobs are long jobs, which however, consume about 88% overall task execution time, in line with the statistics of the original trace.

We use average job arrival rate as a tuning knob to adjust the cluster load. As the Poisson arrival process has been widely considered a good model for datacenter workload, we assume that the job arrival process follows the Poisson distribution. The experimental results are also compared against the simulation results. The simulators for Pigeon and Eagle are the same as the ones described in the previous section and the open source event-driven simulator for Sparrow [43] is used.

We find that the short job performance for Sparrow and Eagle are very sensitive to the number of schedulers in use (by changing the number of schedulers from 1 to 10). This is because the processing delay in the probe phase becomes non-negligible compared to the job execution time for short jobs. In contrast, Pigeon offers almost the same performance, regardless how many schedulers are used. In all the experiments, we use 10 schedulers to minimize the impact of the processing delay for Sparrow and Eagle.

Figure 4.11 depicts both measured (on EC2) (denoted as Imp) and simulated (denoted as Sim) 50th, 90th and 99th short and long job completion times normalized to Pigeon. The results for Sparrow and Eagle are depicted in Figures 4.11 (a) and (b) and Figures 4.11(c) and (d), respectively. Clearly, the experiment results are consistent with the simulation results. The differences between experiment and simulation are within 15% for short jobs and 5% for long jobs, mainly caused by the unaccounted processing overhead in the simulation.



(a) Parallel Scaling

(b) Sequential Scaling

(c) Sequential Scaling

Figure 4.11: Experiment vs Simulation. Sparrow (a) short job and (b) long job; Eagle (c) short job and (d) long job.

As Sparrow does not distinguish between short and long jobs, it incurs up to 200 (10) times longer short job completion times than Pigeon (Eagle), although it offers up to 15% better long job completion times than both Pigeon and Eagle. This means that Sparrow is not effective in supporting short jobs in the presence of heterogeneous workloads. We also see that Pigeon provides significant performance gain for short jobs over Eagle. For example, at 90% load, the 50th, 90th and 99th percentile short job completion times for Eagle reaches about 25, 30 and 7 times longer than those for Pigeon. Pigeon and Eagle achieve comparable performance for long jobs at all cluster loads. The experiment results indicate that Pigeon is highly effective in handling heterogeneous jobs, which agrees with the simulation results obtained from the previous section.

The Pigeon project information and all the simulation and prototype implementation source codes can be found at https://github.com/ ruby-/pigeon.

4.6 Practical Considerations

This section discusses some practical implementation issues, i.e., how to handle master failure and how to deal with heterogeneous workers and task assignment constraints.

4.6.1 Master Failure Recovery

A master plays a key role in a group. If a master fails, all the group information, such as the queued tasks and idle worker lists, are lost. In a full-fledged implementation of Pigeon, one may borrow a failure recovery mechanism widely used in the traditional distributed systems for failure recovery [54]. To allow fast recovery from a master failure, a second master is selected in a group. The second master can be another worker. A master needs to periodically update the second master on the group information and task state information. Whenever a master failure is detected, the second master can immediately take the master responsibility from the failed master without losing any state information. After a master failure happens, the second master sends a notice to each worker in the group and each scheduler in the cluster to notify them of the changes, so that the subsequent tasks and idle worker notices are sent to the new master. Now the second master acts as a new primary master of the group and then a new secondary master should also be chosen for subsequent backups.

If both masters fail at the same time, the group information is lost. To quickly recover the group information, any worker in the group that detects such a failure can take the responsibility as a master. It broadcasts a message into the group to ask worker status. Each worker sends its response back to the new master with its status (idle or busy, priority, executing task, etc.). The new master also needs to send a message to each scheduler to get the task information sent to the group to recover the task queue list in the group. In case that multiple workers take the responsibility as a new master at the similar time, these workers can elect one as the new master based on some rules, e.g., the timestamp of master declaration time, CPU power or storage capacity and so on.

4.6.2 Dealing with Heterogeneous Workers and Tasks with Assignment Constraints

In the Pigeon design, we implicitly assumed that the same number of workers are assigned to each group and all the workers have the same processing power. In practice, however, the number of workers in a group may not be conveniently set to be the same. Even if the numbers of workers assigned to different groups are the same, different workers may have different processing powers. In this case, the schedulers in Pigeon may need to assign tasks to different groups in proportion to their relative processing powers to balance the task load among groups. More specifically, the probability of a task assigned to a group is proportional to the group's processing power.

Moreover, in practice, some tasks may have to be assigned to specific workers, as the needed resources or data are only available at those workers. All these may cause load imbalance among worker groups and hence have a negative impact on the performance of Pigeon. One possible solution is to require that all masters report their queue lengths for all the priority queues periodically to distributed schedulers. This will allow distributed schedulers to make more informed decision as to how to balance the load among groups. The well-balanced task assignment will reduce the overall job completion latency and increase the overall throughput, and hence resulting in high system utilization.

4.7 Related Work

Today's datacenter job schedulers can be classified into three categories, i.e., centralized, distributed and hybrid. The earlier job schedulers, e.g., Jockey [24], Quincy [34], Tetrished [59], Delay Scheduling [65], Firmament [26] and Yarn [41] are centralized by design. A centralized scheduler can potentially provide high worker utilization, as it has a global view of the worker status for individual workers. But the scalability and head-of-line blocking are the major problems concerning centralized scheduling solutions. The scheduling decisions and status reports can overwhelm a centralized scheduler and cause additional job delay. Some shared-state schedulers, e.g., Apollo [7], Omega [52], and Mesos [30], use a centralized resource manager to maintain shared state. The job distributors are distributed but the decision making is based on the shared status of the cluster resource availability. The shared status is updated by the distributed schedulers and/or workers. However, the shared state may not be always up-to-date and hence may result in job placement conflict and retries. This approach still requires a central entity for shared status maintenance. Recent work BigC [10] and Karios [20] propose to deal with job heterogeneity by suspending long jobs' tasks via lightweight virtualization to enable preemption on individual workers, but have shown significant overhead in preempting resource-intensive tasks.

Yarn Federation [25] is developed to address the scalability issue of Yarn [41]. In Yarn Federation, a cluster is split into sub-clusters. Jobs are distributed to sub-clusters, Jobs are distributed to sub-clusters, each of which in turn performs job scheduling (i.e., distributing tasks of received jobs). With the coordination between resource managers and nodes from different sub-clusters, the tasks of a job can span the entire cluster, not limited to the sub-cluster the job is mapped to. As a result, YARN federation is more of a quasi-centralized task scheduling solution than a hierarchical one. Hydra [16] leverages the Yarn Federation architecture, in which a collection of loosely coupled sub-clusters coordinates to provide the illusion of a single massive cluster. In contrast, Pigeon is indeed a two-level hierarchical task scheduling solution, in which the tasks from a job spans across multiple groups (or sub-clusters). First, distributed job schedulers evenly distribute tasks of jobs to all group masters. Then, in turn, each group master, which is job-agnostic, uses priority queues to differentiate the scheduling of short and long tasks. Moreover, while YARN federation aims to address the scalability issue of the resource manager in YARN, Pigeon mainly aims to address job heterogeneity concerning centralized and hybrid job scheduling solutions.

Sparrow [43], on the other hand, is a fully distributed job scheduler based on random batch-based probe and late task binding. Although free from the scalability issues that plague the centralized job schedulers, the distributed schedulers and workers in Sparrow need to maintain fairly large amounts of task related state information and incur high communication cost for probing, including probe management, probe queuing, probe processing, and redundant probe removals. Furthermore, it does not perform well in highly loaded clusters nor in the presence of heterogeneous workloads. Another probe-based distributed scheduler, Peacock [38], organizes workers in a ring overlay network and a probe can be rotated to its neighbors at fixed time intervals to balance the probe queue lengths among workers. Peacock, however, requires that the workers communicate with each other to form and maintain a ring topology. Moreover, it inherits much of the drawbacks pertaining to probe-based solutions in general.

To solve the scalability issue while providing high performance in the presence of heterogeneous jobs, Hybrid schedulers [19, 21, 37, 63] are proposed. Hybrid schedulers combine a centralized scheduler and a set of distributed schedulers. Mercury [37] uses distributed schedulers to place jobs without latency requirement and a centralized scheduler to place jobs with guaranteed resource requirement. Hawk [21] uses a centralized scheduler for long job placement and the distributed schedulers for short job placement. The short job scheduling is similar to the techniques used in Sparrow, i.e., batch probing and late task binding based. Some workers are reserved to serve short jobs only, as a way to mitigate head-of-line bocking. Moreover, an idle worker can steal tasks belonging to short jobs from other workers to improve efficiency. Eagle [19] improves over Hawk by introducing sticky batch probe with each probe staying on a worker until all the tasks of the job finish. It also allows multiple rounds of probing to mitigate head-of-line blocking. These hybrid schedulers need a central scheduler that can still pose a potential bottleneck. Moreover, short job scheduling is still probe-based and hence, inheriting its shortcomings.

More complex queuing mechanisms than priority queuing are being used to minimize the job performance. Queue reordering [19,32,52,59] is used to reduce the job completion time. More complex queue management techniques [47] such as appropriate queue sizing, prioritization of task execution via queue reordering, and starvation freedom are also being used to improve the efficiency of job scheduling.

Some job scheduling solutions [17,36,57] are developed to improve the service level objectives (SLOs) violations. Morpheus [36] is designed to reduce the SLOs violations through automatically deriving SLOs and job resource models from historical data, relying on recurrent reservations and packing algorithms to enforce SLOs, and dynamic reprovisioning to mitigate inherent execution variance. The tail-cutting techniques [17,57] can help mitigate the impact of stragglers on the job tail-latency performance.

Pigeon differs from the existing solutions in two important aspects. First, it is a hierarchically distributed solution to avoid head-of-line block in centralized schedulers. Second, it is free of the probing phase, a technique shared by all the existing distributed and hybrid solutions.

4.8 Implementation

This subsection briefly describes our Pigeon open source project. The implementation of Pigeon scheduler consists several modules that are used to evaluate and compare Pigeon, Sparrow and Eagle's performance on the AWS (Amazon Web Services) EC2 cluster (with over 100 nodes):

- Pigeon Cloud Engine: This is the core module written in Java. It provides all important features of the Pigeon scheduler described as in the above paper;
- Pigeon Spark Plugin: This module is released with Apache Spark 2.1 and enables Pigeon scheduler features within the Spark;
- Pigeon EC2 Deploy: This Python-based module contains multiple Python or Shell scripts that enable auto-deployment of Pigeon on AWS EC2;
- Modified Sparrow and Eagle: As for the comparison against Pigeon scheduler, each of the mentioned open source software was slightly modified from their original version for the feature support or bug fix issues. None of their core scheduling policies are changed;
- Pigeon plot toolkit: Our collected experiment data and the Python script used for result analysis are also publicly available at: https://github.com/ruby-/data-collection. A quick review of how we depict the results can also be found in the Appendix A.

Next, we provide the detailed implementation information about Pigeon core and the rest modules.

4.8.1 Core Pigeon Module

Pigeon cloud engine is the core component of for the Pigeon scheduler. As shown in Figure 4.12, because Pigeon scheduler is only responsible to dispatch the incoming requests to the delegated Pigeon masters in the second layer, it's typically light-weighted in a properly configured and balanced scenario. And the user can send job request to any distributed scheduler in the frontend as depicted. For the data and transition layer, we adopt Apache Thrift as the RPC (Remote Procedure Calls) support. A detailed discussion on the Thrift RPC framework is out of the scope of this dissertation, so here we only give a brief introduction. Thrift is an interface definition language (IDL) and binary communication protocol used for defining and creating services for numerous languages, thus the use of Pigeon is not necessarily tied to a specific language. The input job requests are typically formed as a number of job trace files collected from the real cluster in the industry. Each line of entry in the request file represents a job executed and recorded, specifies the time stamps on arrival and a list tasks with their execution latency.

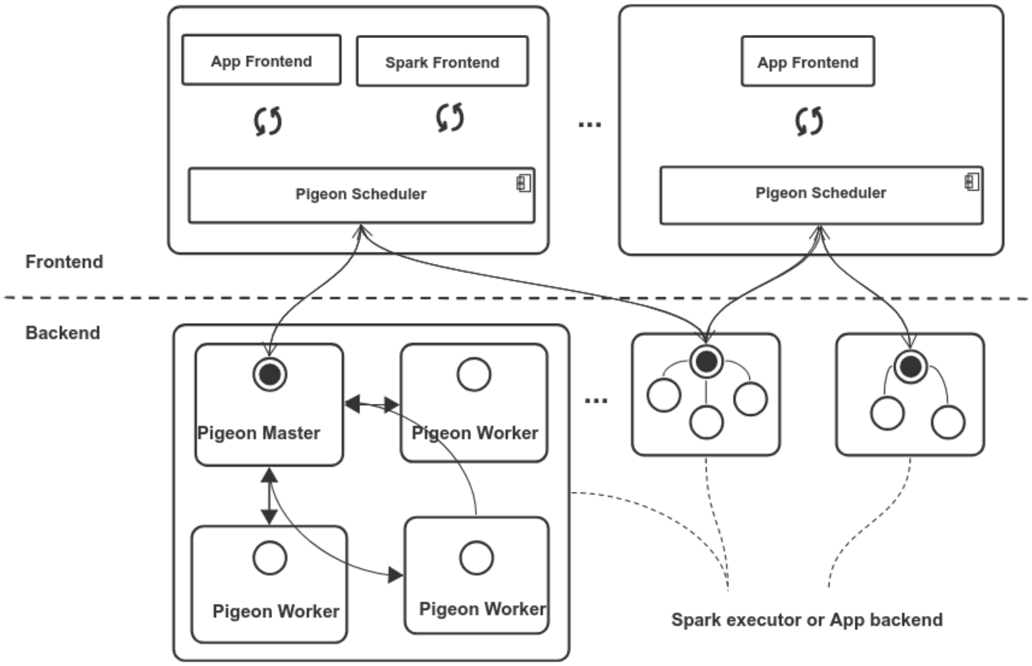


Figure 4.12: Pigeon system components

The Pigeon master is selected from the Pigeon workers running at the backend of each worker node. At startup, all Pigeon workers belong to the same Pigeon group will register themselves with the delegated Pigeon master specified in the configuration. In the current release of Pigeon, the worker node by default requires at least one CPU and sufficient amount of memory for the type of input fed in, but a future extension to the fully utilization of the resources can be trivial to achieve.

In Figure 4.13 we show the most important remote procedure calls invoked by the Pigeon engine. As shown in the figure, a typical workflow of Pigeon involves multiple calls in between the long-running *frontend* daemon and *backend* daemon procedures (the pigeon masters and workers). The pigeon frontend can be implemented in any applications or platforms that extend the exposed interfaces (e.g. *SubmitJob() method*). The Pigeon scheduler immediately dispatch the tasks to the delegated master nodes attached to it, and keeps track of all tasks that belong to the same job request. The Pigeon master maintains a list of idle workers and continuously process the upcoming task requests and finished tasks among its group for that list. Once all tasks belong to the same job request completes, the Pigeon master shall notify the scheduler, then the scheduler reports back to the frontend once all tasks belong to the same job request are completed.

4.8.2 Spark Plugin and Other Modules

We also provide a Pigeon Spark Plugin for the user to evaluate Pigeon on their own Spark workloads. This implementation is written in Scala and Java and can also be found in our public repository: *https://github.com/ruby-/spark-parent_2.11*. A README file can

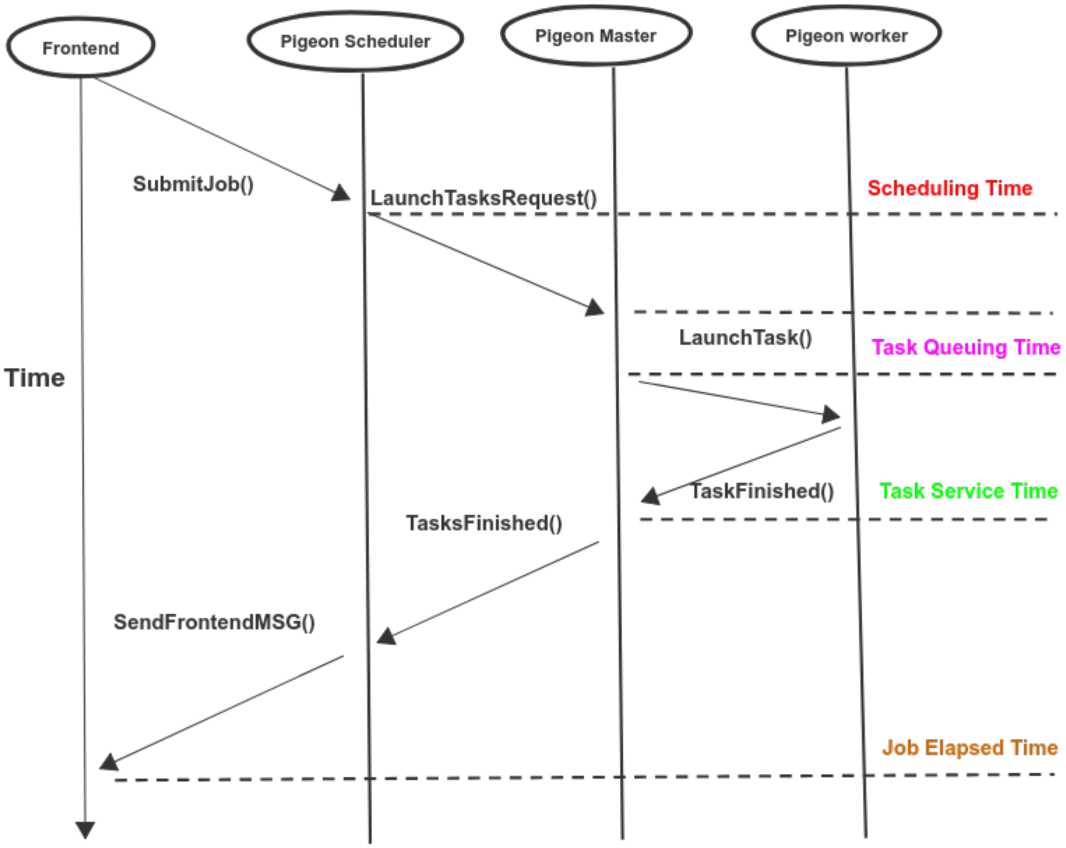


Figure 4.13: Pigeon Core RPC (parameters not shown)

be found on the page to guide the user through the usage of that plugin.

Also, for the convenience of deploying Pigeon scheduler on the EC2 cloud, we embed the AWS SDK [2] and provide multiple CLI (Command Line Interface) tools as a separate module in our Pigeon open source project for user to automatically deploy and manage their Pigeon cluster. This toolkit can be retrieved at the same public code repository (the Pigeon Core) under the */deploy* folder and a README file with an example on startup, deploy, collect data and finally shutdown the cluster.

The rest modules are related to Sparrow [43] and Eagle [19] open source projects. Both software are slighted modified from their original version to accommodate at least the following features: 1) Major bug fix; 2) Support auto-deploy on EC2 cloud; 3) Support frontend interface and retrieve performance matrics. Both projects are re-distributed under the same license with their original copy and can be found in the following public code repositories:

- Sparrow-mod: https://github.com/ruby-/sparrow-mod
- Eagle-mod: https://github.com/ruby-/eagle-mod

4.9 Conclusions

In this paper, we propose Pigeon, a distributed hierarchical job scheduler for datacenters. In Pigeon, workers are divided into groups. Each group has a master worker which centrally manages all the tasks handled by the group. Weighted fair queuing is used to provide priority service differentiation between tasks of short jobs and tasks of long jobs. A small portion of workers in each group are reserved to serve short job tasks only. The ability of each master in managing its group resources centrally makes Pigeon highly effective in scheduling heterogeneous jobs. The analysis, simulation and experiment results demonstrate that Pigeon outperforms Sparrow and Eagle by significant margins. Pigeon is implemented and tested in Amazon EC2 cloud, which has validated the Pigeon simulator used for the Pigeon evaluation.

Chapter 5

Conclusion

In this dissertation, we present a new performance model and job scheduler for today's data centers. Our developed tools and collected experiment results are all published as open source projects that may directly contribute to other related researches in academia or industry.

The first presented modeling approach (IPSO) in this dissertation is inspired by the classic Amdahl's model which can be applied in a wide domain of parallel computing. This new model generalizes the classic performance models of its predecessors and renders them as its special cases. The direct extension of this new model can be unfolded at two variations: a deterministic version and a stochastic version. While the earlier can be used as a quick and qualitative analysis method, the latter is more accurate and can be applied in quantitative measurement or scenarios when strict system modeling is required. In this research, our experiment studies on eight selected benchmarks and two case studies found in other publications verify IPSO's feasibility and efficacy in a real practice.

In the consideration of a common feature of today's data-intensive applications, a performance model which can account for multi-staged workloads is developed (USBA) in this research. As a direct descendant of IPSO, the USBA model generalizes all mentioned classic performance models and can be applied in both stochastic and deterministic scenarios. Moreover USBA can also be directly utilized in the analysis of multi-staged workload such as Spark applications. In this work, we also use our trained USBA model to predict the speedup of four selected machine learning benchmarks at an EC2 cluster which spins up to 260 nodes in size. Our experiment results verify this model's accuracy and its value in guiding resource provisioning for data-intensive applications. At the end of this work and the previous one, we release our modeling tool as an open source project.

Move on towards the investigation of resource provision techniques in modern data centers, we allocate several important factors that can greatly impact the performance of execution such as head-of-line blocking, long job starvation, etc. And based on these observations, we narrow down several major causes of job execution delay in current distributed or centralized solutions and design a new distributed, hierarchical scheduler called Pigeon. Pigeon leverages the speedup of job scheduling by its quick dispatching ability on top layer and special queuing mechanism on the second layer. Our experiment results on the AWS EC2 cloud computing platform significantly outperform existing schedulers such as Sparrow or Eagle. We also open source our Pigeon project under MIT license after this work's been published.

In our next steps, we plan to transplant the Pigeon cloud engine into today's edge computing environment. The vibrant developments of today's IoT (Internet of Things) techniques indicate potentially a great value of the light-weighted, efficient job scheduler for the next generation of IoT clusters. Specifically, we are targeting at releasing Pigeon cloud engine depending on container orchestration systems, such as Kubernetes-based edge computing platforms (e.g. K3S, KubeEdge, etc). The key challenge is to integrate the Pigeon cloud engine into existing edge computing platforms and enable its management on the microservices, also from the users' aspect, Service Level Agreement (SLA) must be met.

Appendix A

Python script: Pigeon plot

This python script plots our main results of Pigeon scheduler against Sparrow and Eagle for 50, 90 and 99 percentiles job latency. Collection of raw experiment data can be retrieved from our public repository at https://github.com/ruby-/data-collection.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random

# randomize color for visualization
# cmap=plt.cm.get_cmap(plt.cm.viridis,143)
# c=cmap(random.randint(1,144))
#################################
#           Functions
#################################
# Available gradient for cmaps:
```

```
# 'Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds',
#             'YlOrBr', 'YlOrRd', 'OrRd', 'PuRd', 'RdPu', 'BuPu',
#             'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn', 'YlGn'
#
def gradientbars_0(bars):
    grad = np.atleast_2d(np.linspace(0, 1, 256)).T
    ax = bars[0].axes
    lim = ax.get_xlim() + ax.get_ylim()
    for bar in bars:
        bar.set_zorder(1)
        bar.set_facecolor('none')
        x, y = bar.get_xy()
        w, h = bar.get_width(), bar.get_height()
        ax.imshow(grad, extent=[x, x + w, y, y + h],
                  aspect="auto", zorder=0)
    ax.axis(lim)

def gradientbars(bars, cname, py, height):
    grad = np.atleast_2d(np.linspace(0, 1, 256)).T
    ax = bars[0].axes
    lim = ax.get_xlim() + ax.get_ylim()

    i = 0
    for bar in bars:
        bar.set_zorder(1)
        bar.set_facecolor('none')
        x, y = bar.get_x(), py
        w, h = bar.get_width(), height[i]
        i += 1
        ax.imshow(grad, extent=[x, x + w, y, y + h],
                  cmap=plt.get_cmap(cname), aspect="auto", zorder=0, alpha=0.8)

    ax.axis(lim)

####################################
#              Adjustments
####################################
data_path = "https://raw.githubusercontent.com/ruby-/data-collection/master/long_90_tile.csv"
width = 0.15  # the width of a bar
fig_h = 6  # figure height
```

```
fig_w = 8  # figure width
fig_ylable = 'Pigeon␣Speedup'  # y-lable
fig_xlable = 'load␣level␣(0~100%␣of␣total␣#nodes)'  # x-lable
fig_title = '90-tile␣plot␣(long)'  # figure title
# x-lables for bars
labels = ['50%', '60%', '70%', '80%', '90%']

# Gradiently fade-out ratio of sparrow(max) bar to figure top margin
fadeYTop = 0.8
# scale of sparrow above threshold, adjust this value to change fading-bar height
scale = 2
# Space left for legend plot
lscale = 0.5
# uncomment below to choose postion for legend plot
legend_pos = 'upper␣right'
# legend_pos = 'upper left'

# Rotation of text label above each bar
txt_rotate_n = 60
```

```

#####################################
#               Data input
#####################################
df = pd.read_csv(data_path)
eagle = df['eagle']
eagle_simulated = df['eagle_s']
sparrow = df['sparrow']
sparrow_s = df['sparrow_s']

#####################################
#               Vars
#####################################
# split sparrow up/below threshold value
ruler = max(np.maximum(eagle, eagle_simulated))
threshold = ruler * 1.5
v_sparrow = np.array(sparrow)
v_sparrow_s = np.array(sparrow_s)

# Setting the positions and width for the bars
pos = list(range(len(eagle)))
# Adjust this value to leave space for legend plot
```

```
offset = ruler * lscale

# colors of bars
c1 = 'seagreen'  # eagle
c2 = 'goldenrod'  # eagle_s
c3 = 'darkred'  # sparrow_below_threshold
c4 = 'red'  # sparrow_above_threshold
c5 = 'mediumslateblue'  # sparrow_s_below_threshold
c6 = 'slateblue'  # sparrow_s_above_threshold
c7 = 'dimgray'  # threshold line color
c8 = 'dimgray'  # text above threshold line color
c9 = 'Reds'  # gradient color for sparrow
c10 = 'Purples'  # gradient color for sparrow_s

#################################
#            Graphix
#################################
# Plotting the bars
fig, ax = plt.subplots(figsize=(fig_w, fig_h))
# Setting axis labels and ticks
ax.set_ylabel(fig_ylable)
ax.set_xlabel(fig_xlable)
ax.set_title(fig_title)
ax.set_xticks([p + 1.5 * width for p in pos])
ax.set_xticklabels(labels)

eagle_bar = plt.bar(pos, eagle, width,
                    alpha=0.8,
                    color='w',
                    hatch='',
                    facecolor=c1,
                    edgecolor='w',
                    label=labels[0])

eagle_simulated_bar = plt.bar([p + width for p in pos], eagle_simulated, width,
                              alpha=0.69,
                              color='w',
                              hatch='x',
                              facecolor=c2,
                              edgecolor='w',
                              label=labels[1])

# upper
```

```
sparrow_above_threshold_tmp = np.maximum(v_sparrow - threshold, 0)
indices_up = np.nonzero(sparrow_above_threshold_tmp)
sparrow_above_threshold = sparrow_above_threshold_tmp[indices_up]

sparrow_s_above_threshold_tmp = np.maximum(v_sparrow_s - threshold, 0)
indices_up_s = np.nonzero(sparrow_s_above_threshold_tmp)
sparrow_s_above_threshold = sparrow_s_above_threshold_tmp[indices_up]
# lower
sparrow_below_threshold = np.minimum(v_sparrow, threshold)
indices_below = np.where(v_sparrow < threshold)
sparrow_below_threshold_shown = v_sparrow[indices_below]

sparrow_s_below_threshold = np.minimum(v_sparrow_s, threshold)
indices_below_s = np.where(v_sparrow_s < threshold)
sparrow_s_below_threshold_shown = v_sparrow_s[indices_below]

```

```
# sparrow bar below threshold
sparrow_bar_below = plt.bar([p + width * 2 for p in pos],
    sparrow_below_threshold, width,
                                   alpha=0.8,
                                   color='k',
                                   hatch='',
                                   facecolor=c3,
                                   edgecolor='w',
                                   ls='-',
                                   lw=None,
                                   label=labels[2])

sparrow_s_bar_below = plt.bar([p + width * 3 for p in pos],
     sparrow_s_below_threshold, width,
                                     alpha=0.99,
                                     color='k',
                                     hatch='\\',
                                     facecolor=c5,
                                     edgecolor='w',
                                     ls='-',
                                     lw=None,
                                     label=labels[2])
```

```

# sparrow bar above threshold
 if len(indices_up[0]):
    sparrow_bar_upper = plt.bar([p + width * 2 for p in np.
        nditer(indices_up)], sparrow_above_threshold, width,
                                  bottom=
                                      sparrow_below_threshold[
                                      indices_up],
                                  alpha=0.8,
                                  color='k',
                                  hatch='',
                                  facecolor=c4,
                                  edgecolor='w',
                                  ls='-',
                                  lw=None,
                                  label=labels[2])
    # draw the bars above threshold as faded
    sparrow_dist_ratio = sparrow_above_threshold / \
        (np.amax(sparrow_above_threshold))
    sparrow_dist_arr = (ruler * scale -
                          threshold) * fadeYTop *
                              sparrow_dist_ratio
    gradientbars(sparrow_bar_upper, c9, threshold,
       sparrow_dist_arr.tolist())

if len(indices_up_s[0]):
    sparrow_s_bar_upper = plt.bar([p + width * 3 for p in
       np.nditer(indices_up_s)], sparrow_s_above_threshold,
       width,
                                   bottom=
                                      sparrow_s_below_threshold
                                      [indices_up_s],
                                   alpha=0.99,
                                   color='k',
                                   hatch='\\',
                                   facecolor=c6,
                                   edgecolor='w',
                                   ls='-',
                                   lw=None,
                                   label=labels[2])

    sparrow_s_dist_ratio = sparrow_s_above_threshold / \
        (np.amax(sparrow_s_above_threshold))
    sparrow_s_dist_arr = (ruler * scale -
                                        threshold) * fadeYTop *
                                            sparrow_s_dist_ratio
      gradientbars(sparrow_s_bar_upper, c10,
                          threshold, sparrow_s_dist_arr.tolist())


# Setting text labels on each bar
for x, y in zip(pos, eagle):
      plt.text(x, y, '%.1f' % y, ha='center',
                    va='bottom', rotation=txt_rotate_n)

for x, y in zip(pos, eagle_simulated):
      plt.text(x + width, y, '%.1f' %
                    y, ha='center', va='bottom', rotation=
                         txt_rotate_n)

if len(indices_below[0]):
      for x, y in zip(np.nditer(indices_below), np.nditer(
          sparrow_below_threshold_shown)):
            plt.text(x + width * 2, y, '%.1f' %
                          y, ha='center', va='bottom', rotation=
                               txt_rotate_n)
```

```

if len(indices_up[0]):
    for x, y, yPos in zip(np.nditer(indices_up), np.nditer(sparrow_above_threshold + threshold), np.nditer(sparrow_dist_arr)):
        plt.text(x + width * 2,  threshold + yPos, '%.1f' %
                 y, ha='center', va='bottom', rotation=txt_rotate_n)

if len(indices_below_s[0]):
    for x, y in zip(np.nditer(indices_below_s), np.nditer(sparrow_s_below_threshold_shown)):
        plt.text(x + width * 3, y, '%.1f' %
                 y, ha='center', va='bottom', rotation=txt_rotate_n)

if len(indices_up_s[0]):
    for x, y, yPos in zip(np.nditer(indices_up_s), np.nditer(sparrow_s_above_threshold + threshold), np.nditer(sparrow_s_dist_arr)):
        plt.text(x + width * 3,  threshold + yPos, '%.1f' %
                 y, ha='center', va='bottom', rotation=
                         txt_rotate_n)

# Adding the legend and showing the plot
plt.legend(['eagle', 'eagle_simulated', 'sparrow',
               'sparrow_simulated'], loc=legend_pos)

# Setting the x-axis and y-axis limits
plt.xlim(min(pos) - width, max(pos) + width * 5)

if len(indices_up[0]) or len(indices_up_s[0]):
    plt.ylim([0, ruler * 2 + offset])
    plt.yticks(np.arange(0, threshold, 10))
    # horizontal line indicating the threshold
    ax.plot([0., 4.5], [threshold, threshold], "k—", color
       =c7)
    plt.text(pos[0] + width, threshold,
                '>␣{:0.1f}x'.format(threshold), ha='center',
                   va='bottom', color=c8)
else:
    plt.ylim([0, ruler + offset])
    plt.yticks(np.arange(0, threshold, 0.2))

```

```
# plt.grid()
plt.show()
```

BIBLIOGRAPHY

[1] Apache hadoop. `http://hadoop.apache.org/`.

[2] Aws sdk for python. `https://aws.amazon.com/sdk-for-python/`.

[3] Ipso(beta) source code. `https://github.com/ruby-/IPSO.git`.

[4] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, volume 2, pages 4–2, 2017.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of Am. Federation of Infomation Processing Societies Conf.*, pages 483–485. ACM, 1967.

[6] J. Bhimani, N. Mi, M. Leeser, and Z. Yang. Fim: performance prediction for parallel computation in iterative data processing applications. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 359–366. IEEE, 2017.

[7] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *Proceedings of OSDI*, 2014.

[8] J. Brutlag. Speed matters for google web search. In *Google*, 2009.

[9] H. Che and M. Nguyen. Amdahl's Law for Multithreaded Multicore Processors. *Journal of Parallel and Distributed Computing*, 74(10):3056–3069, Oct. 2014.

[10] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proceedings of USENIX Annual Technical Conference*, 2017.

[11] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of VLDB Endowment*, 2012.

[12] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of MASCOTS*, 2011.

[13] Y. Chen, R. Griffith, D. Zats, A. D. Joseph, and R. Katz. Understanding TCP incast and its implications for big data workloads. *;login:*, 37(3):24–38, 2012.

[14] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 98–109, 2011.

[15] R. B. Cooper. *Introduction to Queueing Theory.* North Holland, 1981.

[16] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[17] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2), 2013.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation - OSDI '04*, pages 137–150, 2004.

[19] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 497–509, 2016.

[20] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates . In *Proceedings of ACM Symposium on Clod Computing (SOCC)*, 2018.

[21] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of USENIX Annual Technical Conference (ATC)*, 2015.

[22] N. Draper. Response surface methodology: Process and product optimization using designed experiments: Rh myers and dc montgomery,(wiley, new york, 1995, isbn: 0471581003, pp. 714), 1997.

[23] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's Law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 362–370. ACM, 2010.

[24] A. D. Fergusin, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of EuroSys*, 2012.

[25] A. S. Foundation. Hadoop: Yarn federation, 2018.

[26] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmanent: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of USENIX Symposium on Iperating System Design (OSDI)*, 2016.

[27] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[28] M. D. Hill and M. R. Marty. Amdahl's Law in the multicore era. *Computer*, 41(7):33–38, 2008.

[29] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[30] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of NSDI*, 2011.

[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[32] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling Jobs Across Geo-distributed Datacenters. In *Proceedings of SoCC*, 2011.

[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, 2007.

[34] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of SOSP*, 2012.

[35] M. Jeon, S. Kim, S. won Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the ACM SIGIR*, 2014.

[36] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[37] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 485–497, 2015.

[38] M. Khelghatdoust and V. Gramolim. Peacock: Probe-Based Scheduling of Jobs by Rotating Between Elastic Queuess. In *Proceedings of International Conference on Parallel and Distributed Computing*, 2018.

[39] M. Manivannan, B. Juurlink, and P. Stenstrom. Implications of merging phases on scalability of multi-core architectures. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 622–631, 2011.

[40] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 441. IEEE, 2007.

[41] A. C. Murthy, C. Douglas, M. Konar, O. O'Malley, S. Radia, S. Agarwal, and V. KV. Architecture of next generation apache hadoop mapreduce framework. *Apache Jira*, 2011.

[42] O. O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark.org/Yahoo-Hadoop.pdf,(May)*, pages 1–3, 2008.

[43] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of ACM Symposium on Operating System (SODP)*, 2013.

[44] J. R. Phillips. Zunzun. com online curve fitting and surface fitting web site. *United States*, 2012.

[45] H. Qu, O. Mashayekhi, D. Terei, and P. Levis. Canary: A Scheduling Architecture for High Performance Cloud Computing. *arXiv:1602.01412v1 [cs.DC]*, 2016.

[46] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[47] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient Queue Management for Cluster Scheduling. In *Proceedings EroSys*, 2016.

[48] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of ACM Symposium on Cloud Computing (SOCC)*, 2012.

[49] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[50] D. Richins, T. Ahmed, R. Clapp, and V. J. Reddi. Amdahl's law in big data analytics: Alive and kicking in tpcx-bb (bigbench). In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 630–642. IEEE, 2018.

[51] S. Ristov, R. Prodan, M. Gusev, D. Petcu, and J. Barbosa. Elastic cloud services compliance with gustafson's and amdahl's laws. 2016.

[52] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*, 2013.

[53] R. Sheldon. *Introduction to Probability Models*. Academic Press, 2014.

[54] R. S. Stutsman. Durabilit and Crash Recovery in Distributed In-Memory Storage Systems . In *Dissertation of Doctor Philosophy*, 1987.

[55] X. H. Sun and Y. Chen. Reevaluating Amdahl's Law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[56] K. Suo, J. Rao, H. Jiang, and W. Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of ACM European Conference on Computer systems (EuroSys)*, 2018.

[57] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: cutting tail latency in cloud data stores via adaptive replica selection. In *Proceeding of USENIX NSDI*, 2015.

[58] I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera. Mrpr: a mapreduce solution for prototype reduction in big data classification. *neurocomputing*, 150:331–345, 2015.

[59] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of EuroSys*, 2016.

[60] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[61] S. Wolfram. *The mathematica.* Cambridge university press Cambridge, 1999.

[62] D. H. Woo and H. H. S. Lee. Extending Amdahl's Law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[63] Y. Xia, R. Ren, H. Cai, A. V. Vasilakos, and Z. Lv. Daphne: A Flexible and Hybrid Scheduling Framework in Multi-Tenant Clusters. *IEEE Transactions on Network and Service Management*, 15(1), 2018.

[64] M. Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters.* PhD thesis, University of California, Berkeley, 2013.

[65] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of EuroSys*, 2010.

[66] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[67] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[68] S. M. Zahedi, Q. Llull, and B. C. Lee. Amdahl's law in the datacenter era: A market for fair processor allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14, Feb 2018.