Improving the Efficiency of Heterogeneous Clouds

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation

von

Michael Kaufmann, Dipl.-Inf.
aus Stuttgart

Tag der mündlichen Prüfung: 17. Oktober 2019

Erste Referentin: Professorin Dr. Martina Zitterbart
Karlsruher Institut für Technologie (KIT)

Zweiter Referent: Professor Dr. Achim Streit
Karlsruher Institut für Technologie (KIT)

Acknowledgements

I pursued a doctoral degree because I wanted to become an expert on at least one topic, know it inside out. However, the more I learned about it, the more it became clear that there's always more I don't know than what I know. Nevertheless, I have learned a tremendous amount about scheduling of distributed applications, resource management and machine learning.

This work is the result of a four and a half year long journey through a deep and complex labyrinth. Not every chosen path leads to the goal and sometimes one has to take a step back and choose another path. However, there's something to be learned on every path and once one finds the right one, the journey can be exciting and (at times) even fun!

On my journey I was accompanied by many people. First and foremost is my advisor, Prof. Dr. Martina Zitterbart, who gave me the opportunity to pursue my wish and Bernard Metzler for his support throughout my journey. I'd also like to thank my manager Haris Pozidis for giving me time and space to work on my research and my colleague Kornilios Kourtis for many fruitful discussions and his paper writing lessons.

I also want to thank Thomas Parnell for this invaluable help with machine learning and his unique ability to give me the confidence of being on the right path as well as Celestine Mendler-Duenner, Andreea Anghel, Jonas Pfefferle, Animesh Trivedi, Adrian Schuepbach, Nikolas Ioannou, Patrick Stuedi, Marc Dombrowa and Marcel Schaal from IBM Research and Matthias Flittner, Mario Hock, Robert Bauer and Hauke Heseding from the Institute of Telematics at the Karlsruhe Institute of Technology for lending me their ears and fun evenings in the Weinkeller in Dagstuhl.

However, the greatest thanks go to my wife Edith for supporting me during long hours of work, busy weekends and coping with a not-so-relaxed me, especially during the last year.

Last but not least I'd like to thank my parents for supporting me and encouraging me to pursue an academic career.

Michael
Rueschlikon, October 22, 2019

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AF	application framework.
AS	application scheduler.
BG	background.
BSP	bulk synchronous parallel.
CART	Classification and Regression Tree.
CDF	Cumulative Distribution Function.
CNN	convolutional neural network.
CoCoA	Communication-efficient distributed dual Coordinate Ascent.
CSV	comma separated values.
DAG	directed acyclic graph.
DNN	deep neural network.
DRV	application driver.
EX	execution environment.
FG	foreground.
FPGA	Field-Programmable Gate Array.
GBDT	Gradient Boosted Decision Tree.
GLM	generalized linear model.
GPU	Graphics Processing Unit.
HTTP	Hypertext Transfer Protocol.

JIT	Just-In-Time.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
LR	Linear Regression.
lSGD	Local SGD.
MCMF	Min-Cost Max-Flow.
MILP	Mixed Integer Linear Programming.
ML	machine learning.
MPI	Message Passing Interface.
MSE	mean square error.
mSGD	Mini-batch SGD.
NN	neural network.
NVMeF	NVMe over Fabrics.
RDD	Resilient Distributed Dataset.
RDMA	remote direct memory access.
REST	Representational State Transfer.
RL	reinforcement learning.
RM	resource manager.
RNN	recurrent neural network.
RPC	remote procedure call.
SCD	stochastic coordinate descent.
SDCA	stochastic dual coordinate ascent.
SGD	Stochastic Gradient Descent.
SLO	Service-Level Objective.
SQL	Structured Query Language.
SSP	stale synchronous parallel.
SVM	support vector machine.

TPU Tensor Processing Unit.

1. Introduction

A mere decade ago, distributed applications for science and commercial use cases were executed on dedicated compute clusters. These clusters were owned and operated by large research facilities and companies and provided a stable hardware platform for several years. For instance, the Sequoia Blue Gene/Q supercomputer is in service since 2011 [128] without major modifications. On this platform, applications are exclusively assigned static partitions with nodes that are almost fully separated from nodes of other partitions [20]. This provided an environment for applications to execute in a predictable manner and thus allowed for simple application scheduling.

Cloud computing has quickly and dramatically changed this situation. Cluster resources, once considered physical objects, are nowadays seen as utilities, i.e., virtual objects that can be acquired when needed and released once the need has ceased. Cloud providers, such as Amazon, Microsoft, Google and IBM, offer shared, virtualized access to their compute infrastructure to anybody, at any time. However, the sharing and virtualization of hardware resources has also impaired the stability and predictability of the environment. Hardware is upgraded regularly, without control or notification of the user. Even within the same instance type, multiple hardware models from different vendors with varying performance characteristics can be found [27]. Additionally, interference due to sharing of hardware resources with other users can cause additional performance variations. This means that distributed applications have to use resources with varying performance characteristics at the same time.

But not only hardware resources are heterogeneous, applications are too. Distributed application frameworks, such as Spark [18], Flink [52] or Tez [59] use directed acyclic graphs (DAGs) to represent applications. DAGs can have multiple diverging and converging paths. On each path, tasks execute different functions on different data with varying resource demands and task runtimes. Other application frameworks, such as those used for distributed machine learning (ML) training, restrict the flexibility of the scheduler due to the properties of the algorithms they are mainly used for.

1.1 Problem statement

Resource utilization efficiency and fast application execution are highly desirable goals in cloud computing. The more efficient resources are used, the lower the cost and the energy consumption of application execution becomes, which is a major concern for cloud

operators. Fast application execution, on the other hand, is beneficial for users, as it reduces their bill and allows them to get results quicker. However, heterogeneity of DAGs, tasks, hardware and accumulated state (due to data and Just-In-Time (JIT) compiler caches) can cause stalls in application execution: Each stall, however, leaves allocated resources idle and reduces utilization efficiency and application performance.

In consequence, distributed application frameworks use approaches to cope with heterogeneity. Several approaches based on user-provided or automatically generated resource requirement estimates exist [75, 64, 33, 44] to cope with hardware heterogeneity. Many approaches, however, consider only applications as a whole, and ignore heterogeneity within applications, such as DAG- and task-heterogeneity.

The most common approach to address DAG-, task-, hardware- and state-heterogeneity is to ignore it. In micro-tasks [6, 18, 37, 36, 59], work is instead split up into a large number of short-running tasks. This enables *natural* load balancing on heterogeneous clusters: Faster resources execute tasks in a shorter amount of time than slower resources and are thus assigned more tasks. However, micro-tasks have two important issues:

(1) Increasing the number of tasks also increases task management (start, initialization, communication and coordination) and scheduling overheads. In consequence, the fraction of time that resources spend on executing application code decreases. However, the purpose of a system is to efficiently execute application code, while reducing the time spent managing the execution as much as possible.

(2) Arbitrarily increasing the number of tasks is incompatible with an important class of applications: ML training, which uses iterative-convergent algorithms, such as mini-batch Stochastic Gradient Descent (SGD) for the training of neural networks (NNs). Doing so impairs their efficiency on an algorithmic level and can increase the total amount of work that needs to be performed to achieve the same result.

1.2 Research questions and contributions

The overarching goals of this thesis are to increase resource utilization efficiency and to decrease application runtime at the same time, in face of DAG, task and hardware-heterogeneity. Resource utilization efficiency refers to the amount of time that is needed to perform a certain amount of work on a resource, e.g., execute a task on a CPU. Hence, if the same resource performs the same amount of work in less (more) time, resource utilization efficiency increases (decreases). This is in contrast to resource utilization, which just refers to fraction of time a resource is busy to perform any work while it is allocated.

The following research questions are addressed in support of these goals:

(1) How can scheduling of individual applications of distributed, DAG-based data-analytics frameworks on heterogeneous clusters be improved?

(2) How can resources be shared efficiently across applications of these frameworks, at small, sub-second time-scales?

(3) How can resources be utilized efficiently for distributed iterative-convergent ML training algorithms in heterogeneous, elastic environments?

The following research contributions are made in this thesis:

(1) Executor-state-aware scheduling on heterogeneous clusters. The main challenge addressed is the identification and exploitation of factors that influence task runtime. A detailed analysis of task runtime behavior of Apache Spark applications lead to the recognition of executor state, i.e., the state of the process that executes a task, as a major factor. The main contribution is stage packing, a novel scheduling technique that exploits executor state to speed up task and application execution. Stage packing itself is agnostic to hardware-heterogeneity. Using task runtime predictions and simple rules, the execution selection process has been improved further on heterogeneous clusters. Both have been implemented in HCL-SP and integrated into Apache Spark. Compared to vanilla Spark, HCL-SP reduces application runtime by ≈1.4× on average and increases resource utilization efficiency by the same factor. At the same time, resource utilization was tripled to ≈30%.

(2) Efficient resource sharing across applications at small time-scales. The work on HCL-SP has shown the importance of executor state and that despite improvements in resource utilization, executors still idle for most of the time. To reduce the latter, executors – or the resources they allocate – need to be shared efficiently. Current systems, however, need to shut down and restart executors, which is a time-consuming process and loses their valuable state. The main contribution is the design and implementation of Mira, a resource manager and elastic application scheduler that treats executors as shareable resource which can be reassigned across applications within milliseconds instead of seconds. Mira is based on HCL-SP and integrated into Spark. The evaluation of Mira shows that resource utilization quadrupled to more than 95% on a shared cluster compared to vanilla Spark. At the same time, average application runtime is almost cut in half.

(3) Elastic and hardware-heterogeneity-aware scheduling and execution of state-of-the-art distributed iterative-convergent machine learning algorithms. Mira has shown that with elastic execution, resource utilization can be increased significantly. For one important class of applications, however, elastic execution is a challenge: Distributed iterative-convergent ML training algorithms that are needed to process vast amounts of data for today's digital economy. These algorithms can become less efficient when executed with a large number of tasks and therefore need to perform more work to achieve the same result. This makes micro-task-based system inept for this class of algorithms. In consequence, only few elastic [82, 114,

112] and no hardware-heterogeneity-aware distributed ML training framework exist. The main issue addressed in this contribution is to enable elastic, load-balanced execution of distributed iterative-convergent ML training algorithms without impairing their performance. To that end, a new execution model, uni-tasks, has been devised. In uni-tasks, not tasks, but small chunks of data are scheduled. Data chunks can be moved freely across nodes and, at the same time, be processed by only a single task on each node. It is this property that allows elastic scaling and load balancing in heterogeneous clusters without performance impairments. Uni-tasks has been implemented in Chicle, a new distributed ML training framework and scheduler. The evaluation with two state-of-the-art distributed ML training algorithms – Communication-efficient distributed dual Coordinate Ascent (CoCoA) and Local SGD (lSGD) – shows that both algorithms can benefit from uni-tasks on heterogeneous clusters, in elastic scenarios and in face of stragglers (intermittently and unpredictably long-running tasks): The total number of epochs needed to converge decreases by up to one order of magnitude compared to a micro-task approach. Chicle is one of the first elastic ML training framework [82, 114, 112] and – to the best of my knowledge – the first one that also supports hardware-heterogeneity-aware load balancing.

1.3 Publications and patents

Work presented in this thesis is based on four publications and one presentation. Additionally, one patent has been filed.[1]

1.3.1 Publications

1.3.1.1 Conferences

- Michael Kaufmann, Kornilios Kourtis, Adrian Schuepbach and Martina Zitterbart. “Mira: Sharing Resources for Distributed Analytics at Small Timescales”. In: *2018 IEEE International Conference on Big Data*. Seattle, WA, USA (December 2018).
- Michael Kaufmann, Kornilios Kourtis, Adrian Schuepbach and Martina Zitterbart. “Mira: Sharing Resources for Distributed Analytics at Small Timescales”. In: *GI INFORMATIK 2019: Best of data science made in DACH*. Kassel, DE (September 2019).

[1] Further patent applications are in preparation at the time of writing.

1.3.1.2 Workshops

- Michael Kaufmann and Kornilios Kourtis. “The HCl Scheduler: Going all-in on Heterogeneity”. In: *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud17)*. Santa Clara, CA, USA (July 2017).
- Michael Kaufmann, Thomas Parnell, and Kornilios Kourtis. “Elastic CoCoA: Scaling In to Improve Convergence”. In: *NeurIPS 2018 Systems for ML workshop*. Montreal, Canada (December 2018).

1.3.1.3 Presentations

- Patrick Stuedi, Michael Kaufmann and Adrian Schuepbach. “Serverless Machine Learning on Modern Hardware Using Apache Spark”. In: *Spark + AI Summit*, San Fransisco, CA, USA (June 2018).

1.3.2 Patents

- Michael Kaufmann, Thomas Parnell, Kornilios Kourtis. “Elastic training of generalized linear models via re-partitioning based on feedback from the training algorithm.” (filed)

1.4 Structure of this thesis

Chapter 2 provides general background information and defines important terms that are used throughout this dissertation. Specific concepts and issues are introduced in each main chapter.

Subsequently, three chapters with original work are presented. Chapter 3 presents HCL-SP and the work on the optimization of individual application schedules on heterogeneous clusters. Chapter 4 presents Mira and covers the work related to efficient resource sharing across multiple applications at small time-scales. Chapter 5 presents uni-tasks and Chile and describes the work on elastic scheduling and execution of distributed ML algorithms.

Chapter 6 reviews and concludes the work presented in this thesis and gives an outlook on open questions and potential future research directions. Supplemental material is provided in the appendix (Chapter A).

2. Background

This chapter provides relevant general background information for the work presented in this thesis and complements background sections of main chapters.

2.1 Heterogeneity

The different types of heterogeneity which are considered in this thesis are listed in Table 2.1.

Term	Definition
Hardware heterogeneity	Refers to different capabilities and performance characteristics of hardware, e.g., CPUs.
State heterogeneity	Refers to heterogeneity of resources due to different states, e.g., data and JIT caches.
Application heterogeneity	Refers applications with heterogeneous DAGs and tasks, as well as different resource and runtime requirements across applications.
DAG heterogeneity	Refers to different resource and runtime requirements of stages and their tasks in a DAG and paths through the DAG.
Task heterogeneity	Refers to different resource and runtime requirements of tasks.

Table 2.1: *Definitions of types of heterogeneity considered in this thesis.*

The following sections elaborate on each type of heterogeneity.

2.1.1 Hardware heterogeneity

This section describes two major sources of hardware heterogeneity in large-scale clusters that back the cloud.

2.1.1.1 Multiple hardware generations and configurations

In many clusters, hardware heterogeneity occurs due to the deployment of multiple generations and configurations of hardware. The reason is the frequent but partial upgrade and replacement of old or failing hardware [33]. It can be observed in small clusters, e.g., as the 17 node test cluster with three hardware generations and configurations, that was used for most experiments in this dissertation, to large clusters, e.g., operated by Google [28] and Amazon [27].

Different generations and configurations of hardware resources are largely[1] compatible with each other and applications can often be executed on any resource without changes (especially for interpreted languages such as Java and Python). Execution speed of applications and tasks can vary, though. Sources for execution speed variations are, for instance, different clock rates, cache sizes or memory capacities and speed. This is an issue as even within cloud instance types, e.g., on Amazon, hardware resources are not identical [27].

In order to prevent stalls and reduce application runtimes in such environments, schedulers need to be aware of these performance variations. While most application frameworks (AFs) and application schedulers (ASs) can deal with different hardware capacities, such as core counts and memory sizes, they are oblivious to different performance characteristics [18, 55, 66, 36, 42]. The most common approach to deal with differently fast hardware resources are *micro-tasks* (see Section 2.2.3.2 for details).

2.1.1.2 Accelerators

The most obvious form of heterogeneity is the use of hardware accelerators, most notably Graphics Processing Units (GPUs) [63, 130] that have gained lots of traction, e.g., with ML applications. The deployment of accelerators in data centers that back the cloud is motivated by shrinking performance gains of new generations of general-purpose compute hardware and the relatively higher performance and energy efficiency of special-purpose accelerators for certain workloads [76]. Aside from GPUs, other special-purpose compute accelerators, such as Google's Tensor Processing Unit (TPU) [83] and Field-Programmable Gate Arrays (FPGAs) [79] are being deployed.

The fundamental difference between most compute accelerators and general purpose CPUs is that they provide a higher level of hardware parallelism. For instance, a 2019 Intel Xeon CPU provides up to 36 hardware threads [121], while an Nvidia Turing based GPU scales up to 3072 hardware threads [123]. For algorithms that can exploit these levels of parallelism, e.g., matrix multiplication and other embarrassingly parallel algorithms, compute accelerators can be beneficial.

Applications, however, need to be adapted and optimized for such accelerators. While some frameworks, such as TensorFlow [62] and PyTorch [85] provide optimized implementations of some algorithms for CPUs and GPUs, and therefore allow applications to

[1] If no CPU-model specific optimizations, e.g., during compilation, are used.

use either, they often still require some application code changes to utilize GPUs[2]. Moreover, as soon as algorithms provided by the framework or libraries are insufficient, custom application-specific code has to be written, which is less likely to support both, CPUs and accelerators, as doing so implies additional development effort. For schedulers, this often means that they have no choice but to schedule tasks on either accelerators or CPUs, which simplifies scheduling decisions as no alternatives have to be considered.

Another, albeit less common in cloud environments, type of accelerators are network and storage accelerators, e.g., remote direct memory access (RDMA) and NVMe over Fabrics (NVMeF), which enable high-throughput and low-latency access to remote memory or storage while freeing up the CPU for other tasks. Abstraction layers, such as MPI [2], that allow applications to use these types of accelerators transparently exist. Where these are not applicable, applications need to be written specifically with these accelerators in mind, in order to make use of them. Moreover, with few exceptions [11, 126], communication is generally limited to peers that feature the same type of accelerator, thus limiting the freedom of schedulers for applications that can use these accelerators.

The use of accelerators is supported by many resource managers (RMs) and AFs. YARN [39], Mesos [21], Kubernetes [64] and Borg [60] allow the assignment of labels to nodes. Labels are strings, e.g., `nvidia-turing-gpu`, without inherent meaning, that can be used to specify the presence of a GPU or another accelerator in a node. During resource acquisition, the AS can specify node labels to control on which type of node the RM will allocate resources on its behalf. The problem is, however, that AFs, such as Spark, do not *understand* these labels nor do they know which tasks of the application can make use of them. In consequence, user-provided labels are applied to the entire application. If, for instance, only few tasks of an application can make use of a GPU, all tasks are necessarily executed on GPU nodes, blocking this presumably rare and expensive resource for other applications to use. Moreover, due to power-envelope constraints, nodes with accelerators may not feature the highest-performing CPUs and therefore may impair runtime of tasks that cannot make use of GPUs.

2.1.2 State heterogeneity

State, e.g., due to data or JIT caches can impact performance of resources and thus make otherwise identical resources appear heterogeneous.

- JIT-compilers translate interpreted code into native (machine) code such that subsequent executions of the same code is accelerated.
- Data caches can reduce the access time.

State is considered with delay scheduling [17], where the scheduler waits for a period of time for the availability of a resource where a large fraction of input data (either cached

[2] In PyTorch, for instance, some function names are suffixed with `CPU` or `GPU`, depending on which one is used.

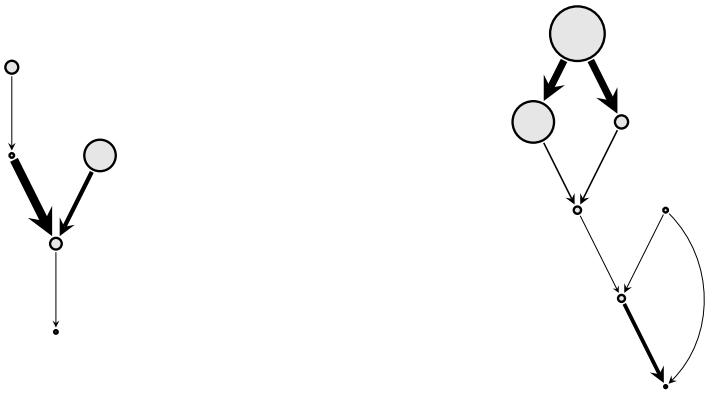
or stored) is located in order to reduce data transfer overheads. This assumes that data transfer impairs performance more than the added delay. Schedulers of serverless frameworks retain and reuse running containers [95, 91, 111, 69] to accelerate the execution of future instances of an application and to reduce resource allocation overheads.

2.1.3 Application heterogeneity

Different applications have different resource requirements. For instance, distributed ML training applications often require multiple GPUs or CPUs for several minutes or hours, whereas some data analytics applications (e.g., SQL queries) can run as short as a few seconds and, due to their often non-linear DAGs, have fluctuating resource demands.

2.1.3.1 DAG heterogeneity

Application DAGs can be heterogeneous [67], in that they are comprised of a diverse set of stages, that each perform different functions. Moreover, stages are split up into multiple tasks that each perform the same function on different (amounts of) data, as depicted in Figures 2.1a and 2.1b for two queries of the TPC-DS benchmark [88, 116]. In this figure, a node corresponds to a stage and edges correspond to data dependencies. The size of a node represents the accumulated task runtime of the corresponding stage and the thickness of an edge the amount of data that is transferred between two stages.



(a) TPC-DS query 41 **(b)** TPC-DS query 44

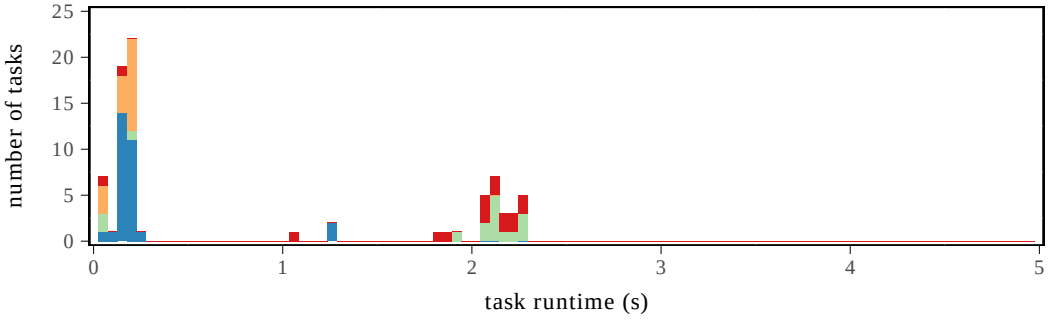
Figure 2.1: *Example of the heterogeneity of application DAGs for two TPC-DS queries. Based on data from Section 4.4.3.*

2.1.3.2 Task heterogeneity

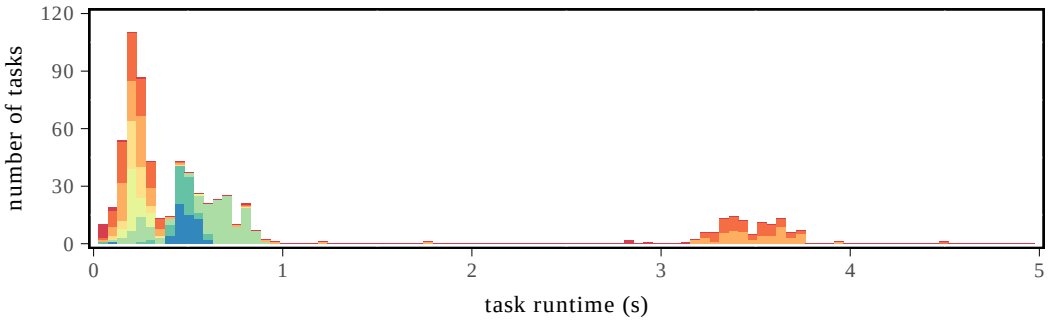
Tasks execute functions on partitions of the input data. These functions transform input data into output data. Tasks of different stages can execute different functions, thus their

runtime behavior is naturally different from each other. Figures 2.2a and 2.2b show histograms of the task runtime distribution for the same applications as in Figures 2.1a and 2.1b. The histograms show the different task runtime distributions of both applications and thus the heterogeneity across applications. The spread of task runtimes within each histogram also shows the heterogeneity within each application: The longest tasks run 475× and 326× longer than the shortest tasks for query 41 and 44 respectively.

Even within the same stage, where all tasks execute the same function on different data partitions, task runtimes can vary greatly. In query 41, for instance the longest task within one stage is on average 60× longer than the shortest task. In query 44, the spread is 176× (the spread is visible by comparing minimal and maximal task runtimes for same-colored bars in both figures).



(a) TPC-DS query 41 with 4 stages and 145 tasks



(b) TPC-DS query 44 with 8 stages and 686 tasks

Figure 2.2: *Example of relative task runtime distributions for two TPC-DS queries. The histograms show two aspects: The runtime distribution across all tasks and the runtime distribution within each stage. Stages are color-coded. Based on data from Section 4.4.3.*

Many application frameworks and their schedulers, such as Spark and its DAG and task scheduler, ignore DAG and task heterogeneity. Instead, these frameworks [6, 18, 37, 36, 114] use the micro-tasks concept to address DAG- and task-heterogeneity. Splitting up the computation into many small (micro-)tasks allows the scheduler to reduce wait times in front of barriers (as those depend on the longest running task) without the need to be aware of DAG nor task heterogeneity (see Sections 2.2.3.2 and 2.2.4.2 for details).

2.1.4 Cluster topology

Cluster topology is a further type of heterogeneity that makes otherwise homogeneous resources appear heterogeneous, in that the runtime of applications can vary depending on the relative location of used resources. The main reason therefore is network I/O and interference thereof between applications. Moreover, the cluster topology often correlates to failure domains. For instance, all servers in a rack, connected to a top-of-rack switch, can be considered one such failure domain, as they all communicate via a single switch. Topology-awareness is supported by many RMs and AF such as YARN, Mesos, Kubernetes and Spark, which know several locality levels, e.g., rack, node and process.

Cluster topology is not considered in this thesis.

2.2 Scheduling

A schedule, or execution plan, defines when and where *n tasks* of *m* applications are executed on *j* machines. Scheduling is the decision problem of finding a schedule, such that a set of constraints are fulfilled. Constraints can be hard (requirement) or soft (preference) and include:

- data and control dependencies among tasks
- hardware resource requirements or preferences
- resource location requirements or preferences
- Service-Level Objectives (SLOs)

Scheduling is also an optimization problem of finding a schedule where the value of an objective is minimized or maximized. Common objectives include:

- task/application runtime (shorter is better)
- task/application throughput (higher is better)
- task/application execution cost (lower is better)
- resource utilization efficiency (higher is better)
- scheduling latency (lower is better)

Objectives are often contradictory such that a single schedule cannot minimize/maximize two or more objectives the same time. For instance, minimizing execution cost is typically not compatible with minimizing application runtime, as the latter may require using more resources, which increases the execution cost. It is therefore generally not possible to find *the* optimal schedule, as trade-offs between objectives have to be made.

In order to compute a schedule the scheduler also requires:

- A **resource model** that represents hardware resources and determines the understanding of the scheduler of the hardware resources.
- An **application model** that represents the application and determines the understanding of the scheduler of the application.
- An **execution model** that defines how work is partitioned and distributed across resources and determines the freedom of the scheduler when making scheduling decisions.

2.2.1 Resource model

Hardware resources contain countless aspects that are irrelevant to solve the scheduling problem. Therefore, abstract resource models are used that represent the actual hardware at varying levels of detail, depending on the scheduling objectives.

For instance, the Completely Fair Scheduling (CFS) scheduler of the Linux kernel schedules individual tasks on CPU cores on a single node. It represents the CPUs of a node in a hierarchical model that resembles the physical cache and memory hierarchy [73]. The Spark task scheduler, on the other hand, uses virtual resources (executors), which represent a tuple of CPU cores and an amount of memory. Executors are only distinguished by the location of task input data as executor-local, node-local and rack-local, which resembles the hierarchy of cluster network fabric.

In general, the model used to represent hardware resources needs to be detailed enough to provide the scheduler with all information it needs to make a scheduling decision but should not contain additional information to avoid overheads during schedule computation. For instance, the resource model for a hardware-heterogeneity-aware scheduler needs to contain, at the very least, information about the hardware class (model/configuration), while a scheduler intended solely for homogeneous clusters does not need this information.

2.2.2 Application model

Similar to resources, application frameworks and their schedulers also use abstract models to represent applications they execute. Application models are closely tied to scheduling challenges. Two of them are relevant here and are introduced in the following.

2.2.2.1 Linear model

Linear models execute one or more instances of a function at a time without branches (on the model level). A widely used linear model is bulk synchronous parallel (BSP). BSP [1] is used to implement many distributed ML training algorithms, such as CoCoA and mini-batch SGD. In BSP, an application with a level of parallelism p is executed on $v \leq p$ (virtual) resources (e.g., CPUs, nodes, executors). The application consists

of multiple *supersteps* (henceforth referred to as *stages*) during which resources execute tasks independently. The completion of a stage is checked periodically and the application proceeds to the next stage if it has. Stages of an application may perform different or, in case of iterative applications, the same function. BSP provides strong consistency guarantees, i.e., results of tasks of stage i can be seen by all tasks of stages $> i$. Figure 2.3b shows an excerpt of a BSP-based application execution. Here, four nodes execute three stages with four tasks each.

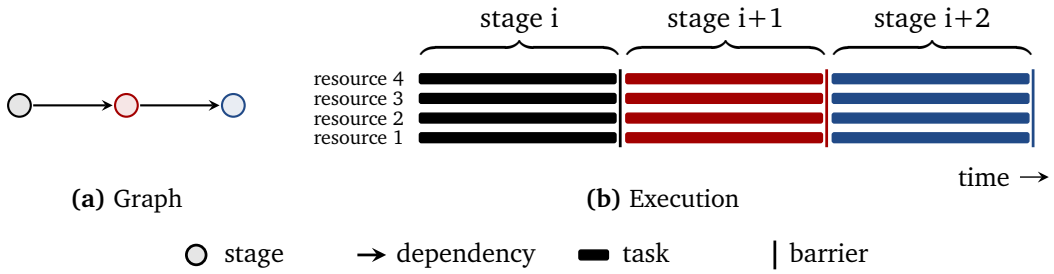


Figure 2.3: *Depiction of a linear model graph (a) and an idealized execution thereof (b) on $v = 4$ resources. Stages (here: $p = 4$) and tasks are color-coded.*

A drawback of BSP is, that it is susceptible to stalls in heterogeneous environments and in the face of stragglers if $p \not\gg v$. Stragglers are intermittently and unpredictably long-running tasks (see Section 2.2.4.1 for details on stragglers). This can reduce resource utilization efficiency and slow down the application execution.

2.2.2.2 Directed acyclic graph (DAG)

The DAG model is a generalization of the linear model. It allows to represent more complex program flows with divergence and convergence of multiple paths within the model, instead of only a single path. DAG is used to represent applications in many distributed application frameworks [18, 59, 52, 55, 96, 99, 67, 5, 48].

Each path consists of one or more stages with data or control dependencies to prior stages on the same path. Each stage s is itself comprised of p_s independent tasks. Barriers exist at the end of each stage and at convergence points. In contrast to linear models, barriers only affect individual stages and converging paths. Figure 2.4 shows an exemplary depiction of a DAG and its execution. As DAGs also contain barriers, they suffer from similar issues and consequences thereof as linear models: If $p_s \not\gg v$, they are susceptible to stalls in heterogeneous environments and in the face of stragglers. In addition to that, stalls can occur in front of convergence points, if stages pass their respective barriers at different points in time, as depicted in Figure 2.4b in front of the blue stage.

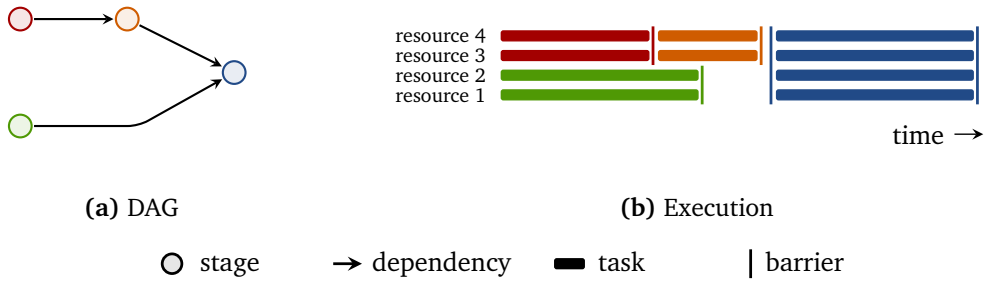


Figure 2.4: *Depiction of DAG (a) and an idealized execution thereof (b) on $v = 4$ resources. Stages may have a different number of tasks. Stages and tasks are color-coded.*

2.2.3 Execution model

The execution model defines how work is split up and distributed across resources. A continuous spectrum w.r.t. the size and number of splits exists. On one side, there are execution models with only a single process per node for the entire application execution. On the other size, there are micro-tasks, where an arbitrary number of small tasks are scheduled across all available nodes.

2.2.3.1 Single process

In this model, a single (or fixed number) of process(es) are executed on each node. This model is used by many Message Passing Interface (MPI) applications, for instance by Snap ML [103] and PyTorch [85]. The main advantage of this model are neglectable scheduling overheads, as processes only need to be scheduled once, at the beginning of the application. However, it does not inherently allow balancing load, elastically scale execution or mitigate stragglers. Instead, applications need to implement these capabilities themselves within the constraints of a fixed number of processes.

2.2.3.2 Micro-tasks

The micro-tasks[3] model is central to many distributed application frameworks, such as MapReduce [6], Spark [18] and others [37, 36, 114, 6]. Each stage s is split up into a large number p_s of small, independently executable tasks, such that $p_s \gg v$. The *size* of a task is defined by the fraction of the total work it performs, i.e., the fraction of the input data it processes. The size of a task also correlates to its runtime. This enables the scheduler to make fine-grained scheduling decisions in order to address the following scheduling challenges:

- **Load balancing:** As $p_s \gg v$, the scheduler can balance task load across all available resources by assigning tasks to resources as soon as they have finished executing

[3] Also referred to as tiny-tasks in related work [37, 36, 87].

the previous task. Moreover, in a heterogeneous cluster, the scheduler can assign more tasks to faster resources than to slower resources using the same mechanism.

- **Elastic scaling:** As long as $p_s > v$, the scheduler can use additional resources that become available during execution. Conversely, if resources are removed during execution, the scheduler can reschedule pending tasks on the remaining resources.
- **Straggler mitigation:** Stragglers are unpredictably and intermittently slow running tasks. As tasks are shorter in general, the assumption is that stragglers become shorter as well. Furthermore, the scheduler can execute pending tasks on other resources (than where the straggler is executed) such that the delay of the stage finish is reduced.
- **Fair resource sharing:** The large number of tasks gives the scheduler ample opportunity to intervene and reassign resources if it detects unfair resource sharing.
- **Error recovery:** As each task is small, the work that is lost in case of an error and needs to be repeated is equally small.

Overheads. The scheduling flexibility of micro-tasks also comes at a cost: Due to the increased number of tasks and constant per-task overheads, the fraction of overheads to the actual work performed increases. These constant overheads include: scheduling overheads, task launch and initialization overheads, coordination and synchronization overheads.

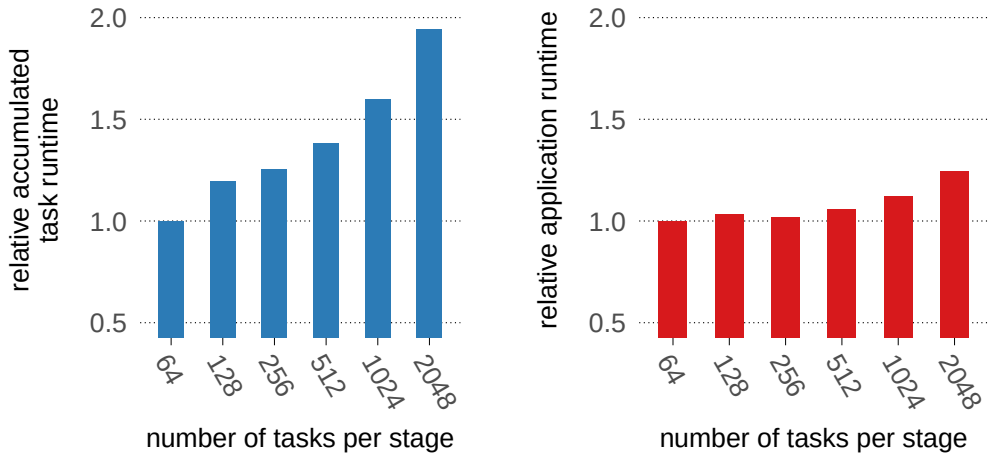
To quantify this overhead, a set of 90 TPC-DS queries [116, 88] were executed on Spark on using the test setup described in Section A.2.2, except for two parameters (`spark.default.parallelism` and `spark.sql.shuffle.partitions`). These parameters control the maximal number of tasks that are used by Spark for each stage. Not all stages can be split up into an arbitrary number of stages, hence some stages use fewer tasks.[4] For both parameters, values between 64 and 2048 have been evaluated (with the same values for both each time). A higher value increases scheduling flexibility.

For each value, the accumulated runtime of all tasks of each query was measured. This time includes task launch, initialization, data retrieval, execution and storage of results data. It does not include scheduling overheads, as the scheduling functionality of Spark is scattered across multiple classes and components which did not allow for reliable measurements. Each query was executed three times for each value.[5]

Figure 2.5a shows the results of this evaluation with the number of tasks per stage on the x-axis and the relative accumulated task runtime, i.e., the relative amount of work necessary, on the y-axis. A lower y-value indicates a more efficient utilization of resources. It shows that with an increasing number of tasks, more total time is spent executing tasks. As the same effective work (executing the application) is performed in

[4] On average, a setting of 2048 uses ≈20× as many tasks as a setting of 64.
[5] One iteration over all queries takes approximately 22 hours on the test cluster.

all cases, resources perform this work less efficiently. Moreover, the increasing scheduling flexibility cannot compensate for the additional work that needs to be performed and application runtime actually increases slightly, as shown in Figure 2.5b.



(a) Overheads **(b)** Application runtime

Figure 2.5: *Overheads (lower is better) and change in application runtime (lower is better) of micro-tasks on Spark for a set of 90 TPC-DS queries relative to 64 tasks per stage.*

Additionally, a smaller experiment has been performed using the test setup described in Section A.1.2 where four nodes are slowed down to increase the heterogeneity of the test cluster. Here, only two values for the number of tasks per stage, 56 (used for Spark experiments in this thesis) and 200 (the default), have been evaluated. Similar to the previous experiments, the accumulated task runtime increased by 1.3× on average when increasing the number of tasks from 56 to 200. However, application runtime was reduced slightly, by 3.6%, showing that micro-tasks can improve application runtime in some scenarios.

Limitations. Apart from overheads, some algorithms, such as mini-batch SGD for distributed ML training of NNs and other distributed ML algorithms, are not compatible with the micro-task approach, as they become less efficient on an algorithmic level when the number of tasks increases (see Chapter 5 for details) and therefore require more total work to be performed to achieve a certain result.

2.2.4 Scheduling challenges

This section introduces the three scheduling challenges which this work focuses on, as well as methods to address them. The general assumption is that the scheduler cannot

suspend or move a task once it has started. As DAGs and linear models are effected similarly, the simpler linear model is used to exemplify the effects and mitigation strategies for each scheduling challenge. Differences to the DAG model are noted where necessary. Table 2.2 lists currently used methods to address each scheduling challenges, which are elaborated on in the following sections.

Stragglers	micro-tasks, worker replication, speculative execution, relaxed consistency
Heterogeneity	micro-tasks, resource requirement specification/prediction
Elasticity	micro-tasks, checkpoint & restart

Table 2.2: *Concepts and strategies to address scheduling challenges.*

2.2.4.1 Stragglers

Stragglers are tasks that occur intermittently, unpredictably and run exceptionally long without any apparent reason. Typical causes for stragglers are interference between tasks on the same node, I/O and networking contention as well as garbage collection (e.g., for JVM-based applications). The probability of any of these causes occurring increases with the number of used resources, hence the further an application scales out, the higher the risk for any single task is to become a straggler. A single straggler, however, is enough to stall progress of an application, as a barrier can only be passed once the last task has reached it.

Figure 2.6 shows an example of this issue: A single tasks per stage becomes a straggler and delays the completion of each stage, while resources of other tasks idle.

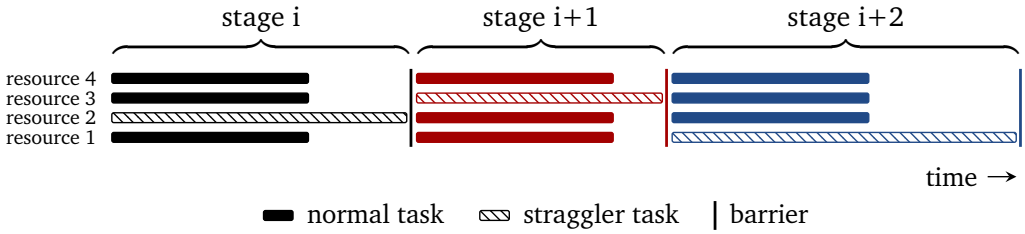


Figure 2.6: *Example of the impact of stragglers on the duration of a stage. Stages are color-coded.*

Individual paths of a DAG behave like a linear model. For the former, the scheduler can schedule tasks from other paths on resources that become idle due to stragglers. Multiple mitigation and prevention strategies have been developed, which are presented in the following:

Micro-tasks

The scheduler can adjust the execution plan for all pending tasks. Once a task is running, its execution plan is set. The defining characteristic of the micro-tasks (Section 2.2.3.2) is the large number of small tasks, hence the fraction of tasks that is running at any point in time is relatively smaller than with a small number of large tasks. This has two benefits:

(1) As tasks get smaller in general, so do stragglers.

(2) With a large fraction of pending tasks, the scheduler can adjust the execution plan to execute tasks on idle resources instead of waiting for a straggler to finish.

This allows the scheduler to hide stragglers that occur early in the execution of a stage but not those that occur late, as here, the fraction of pending tasks becomes smaller. Figure 2.7 depicts the impact of stragglers on the finish time of stages: In stages i and $i+1$ the straggler can be hidden, whereas in stage $i+2$, no further tasks remain to hide the straggler and resources 2 – 4 become idle.

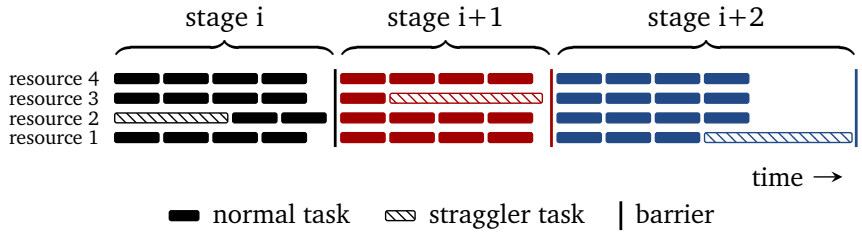


Figure 2.7: *Example of straggler mitigation using micro-tasks. Stages are color-coded.*

Speculative execution and task replication

Another class of straggler mitigation techniques execute copies of tasks. These techniques can be further subdivided into reactive and proactive.

- Reactive approaches [6, 8, 13, 32] monitor task progress and start copies of suspected stragglers with the expectation that copies will finish before the originals. This reduces stalls in the application and tail latencies, at the cost of extra resource utilization.

- Proactive approaches [30, 62, 122] start copies of certain tasks from the beginning and use the first available result. This also comes at the cost of extra resource utilization, even if no stragglers occur and can therefore reduce resource utilization efficiency.

TensorFlow [62] implements a proactive approach for distributed ML training applications based on worker replication. Instead of executing only K tasks, m additional tasks are executed during each iteration. Each task performs the training process on a disjunct

set of training samples. The barrier at the end of the iteration can be passed as soon as K out of $K + m$ tasks have finished, hence up to m stragglers can be tolerated without increasing the runtime of an iteration. A similar approach is proposed by Karakus et al. [122]. TensorFlow's approach exploits the stochastic nature of ML training algorithms where training samples can be processed in any order. It is not applicable in other cases.

Relaxed consistency models

Relaxed consistency models such as stale synchronous parallel (SSP) [31] work by allowing stage $i + n$ to start before all tasks of stage i have finished, thus allowing tasks of up to $n + 1$ subsequent stages to run in parallel. For $n = 0$, SSP degrades to BSP. It is the most common approach to address stragglers in iterative distributed ML training frameworks. Several variations of SSP exist and are used in (or can be used with) many distributed ML frameworks [23, 19, 25, 34, 43, 47, 82, 114, 104]. Figure 2.8 shows an exemplary comparison between a BSP and a SSP-based training process with staleness $n = 1$.

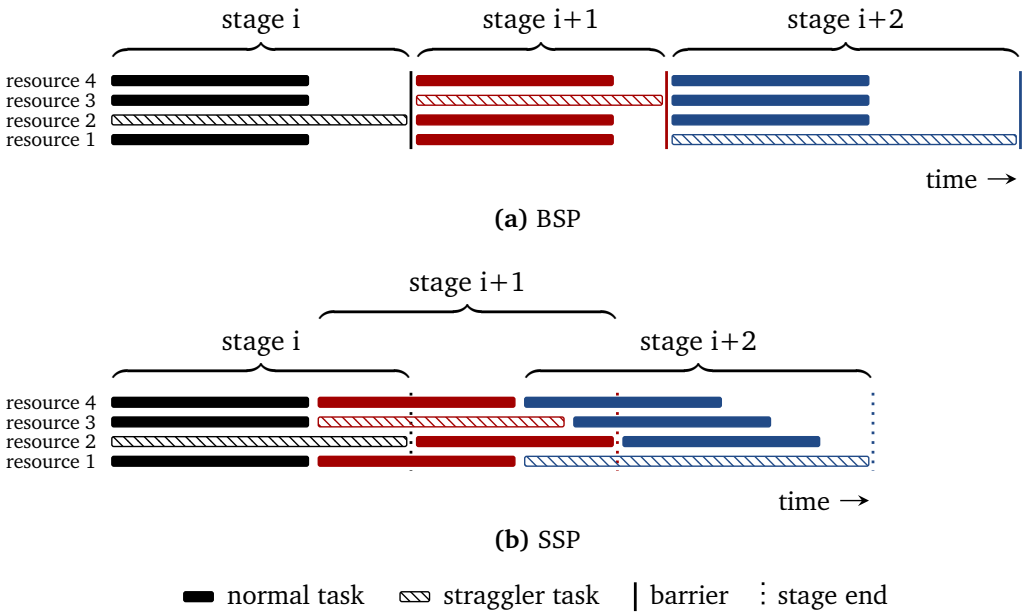


Figure 2.8: *Exemplary comparison of a BSP and SSP (with $n = 1$) schedule with stragglers and identical task runtimes in both cases. Iterations are color-coded.*

The relaxed consistency guarantees of SSP introduce bounded errors in the computation of f_Δ (the model update function) by omitting updates from slow nodes (e.g. node 2 in iteration i in Figure 2.8b) in the current iteration, which have to be corrected in subsequent iterations. As Ho et al. [34] show, this increases the total number of iterations (and therefore epochs) needed to converge. This is compensated by the reduced time per iteration up until a certain point after which the increase in number of epochs outweighs the per-iteration runtime savings. Furthermore, SSP assumes that stragglers are intermittent

and occur on random nodes, hence this approach cannot compensate for static performance differences encountered in heterogeneous systems. Relaxed consistency models, such as SSP, are only applicable if bounded errors can be tolerated.

Other approaches

Wrangler [50] builds a prediction model based on application logs and node utilization metrics. Monitoring these node utilization metrics during runtime, Wrangler predicts whether a node would become overloaded if a specific task were to be executed on it immediately. If that is the case, task execution is delayed until node load has decreased to prevent the task from becoming a straggler.

2.2.4.2 Hardware heterogeneity

Hardware heterogeneity across resources used within the same application, causes permanent, predictable runtime differences in tasks, as depicted in Figure 2.9. Figure 2.9 depicts a BSP schedule on a heterogeneous cluster and shows the steady task runtimes on each node and scheduling gaps that emerge in such a scenario.

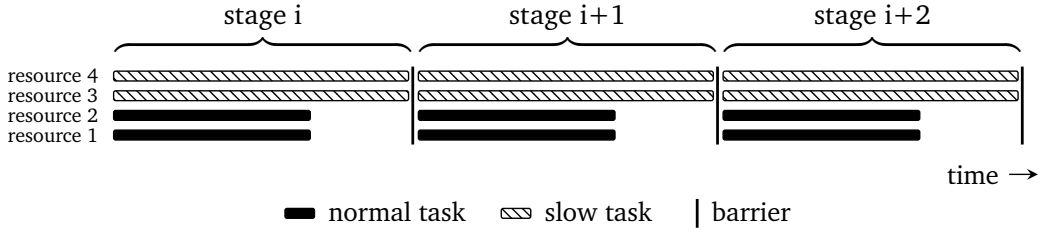
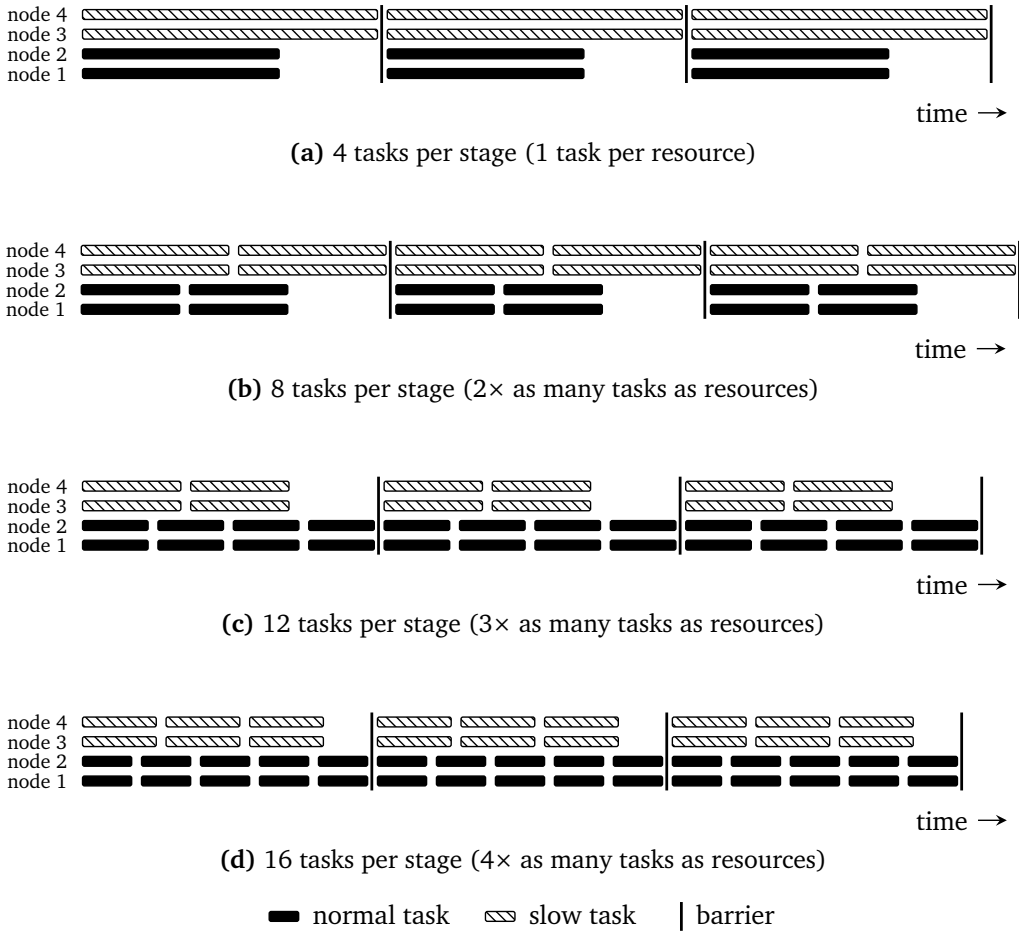


Figure 2.9: *Exemplary depiction of a BSP schedule on a heterogeneous cluster where two resources (3 and 4) are slower than the others (1 and 2). In contrast to stragglers (Figure 2.8), slowdowns due to heterogeneity are steady and predictable.*

Micro-tasks

Micro-tasks is a common approach to address heterogeneity. Due to the large number of small tasks, the scheduler can balance load in a hardware-heterogeneous cluster without needing to be aware of hardware-heterogeneity, simply by assigning tasks to idle resources. Faster resources execute tasks quicker than slower resources and thus become idle sooner, such that more tasks will be scheduled on them. Figures 2.10a through 2.10d depict load balancing with varying number of tasks and shows how an increased number of tasks can reduce idle times.



(a) 4 tasks per stage (1 task per resource)

(b) 8 tasks per stage (2× as many tasks as resources)

(c) 12 tasks per stage (3× as many tasks as resources)

(d) 16 tasks per stage (4× as many tasks as resources)

Figure 2.10: *Exemplary depiction of micro-task schedules for 3 stages with 1-4× as many tasks as nodes, on a system where slow nodes (3 and 4) require 50% more time to execute a task than fast nodes (1 and 2).*

Other approaches

Other approaches based on the specification or automatic prediction of hardware resource requirements exist to address hardware-heterogeneity. These methods are discussed in Chapter 3.

2.2.4.3 Elasticity

Elasticity refers to the addition of removal of resources during the execution of an application and the ability of the scheduler and application to gracefully, i.e., without failing the entire application, adjust to that. Elasticity is important in shared environments where applications start and finish at random points in time and the (fair) resource share of each application changes over the course of their execution.

Micro-tasks

Similar to load balancing in heterogeneous clusters, micro-tasks can also be used to elastically scale in and out by moving pending tasks from nodes that need to be removed to nodes that remain and from existing nodes to newly added nodes. Figure 2.11 depicts an elastic scale-out of a BSP application.

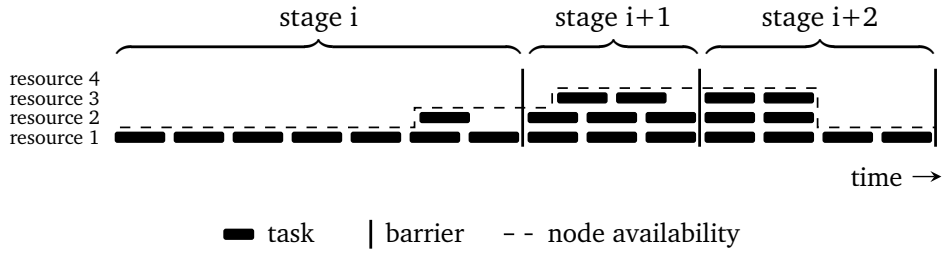


Figure 2.11: *Exemplary depiction of an elastic micro-task schedule for 3 stages with 8 tasks each.*

Checkpoint & restart

Application state can be stored (checkpointed) to a persistent storage after which the application is shut down and restarted using the new resource assignment. This method is used in the Optimus elastic ML training framework [112].

2.2.4.4 Others

Other scheduling challenges exist but are not addressed in this work. They include resiliency and fairness, both of which can also be addressed using micro-tasks.

2.3 Apache Spark

This section presents an overview of Apache Spark [18, 78] and concepts it is based on. This description is based on Spark 2.2.1. Spark is a Java Virtual Machine (JVM)-based distributed data analytics framework, focused on iterative and interactive data analysis, such as Structured Query Language (SQL) processing. Its main abstraction is the **Resilient Distributed Dataset (RDD)**, a read-only collection of objects (e.g., numbers, strings, tuples) that are partitioned and distributed across one or more nodes. RDDs are created by loading data from an external source, e.g., a file system, or by applying a set of transformations on existing RDDs.

Figure 2.12 shows an overview of Spark's driver/worker architecture with its most relevant modules. The driver executes the sequential part of an application and controls the distributed execution of its parallel parts. The worker executes long-running task executor processes. In the following, the function of all modules and their interactions are described.

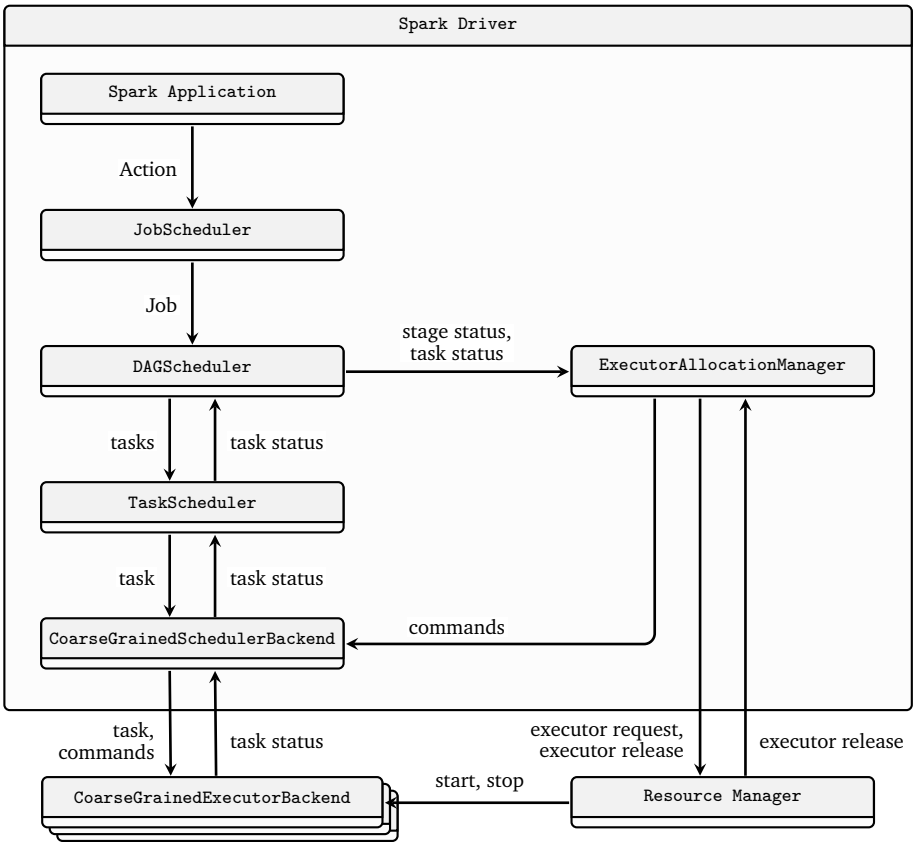


Figure 2.12: *Spark architecture overview.*

- **Spark application:** Spark applications describe transformations on data. A set of transformations with a result is called an action. For instance, `c = a.zip(b).collect` is an action that transforms RDD `a` with I values a_i and RDD `b` with I values b_i into RDD `c` with value pairs (a_i, b_i) for $0 \leq i < I$. The execution of the `zip` transformation and materialization of RDD `a` is triggered by the call to `collect`. Spark applications consist of multiple actions that can be executed sequentially or in parallel.

- **`JobScheduler`:** The job scheduler compiles an action into a logical execution plan, called lineage graph. A lineage graph is a directed acyclic graph (DAG) with RDDs as nodes and dependencies as edges, represents the logical execution plan of an action. The lineage graph for the above stated action is shown in Figure 2.13. The type of RDD specifies the type of operation that is performed on the input data. For instance, the `ZippedPartitionsRDD2` constructs tuples from each value of two equally large input RDDs. Any (user-specified) function that performs the type of operation defined by an RDD can be used to compute a RDD.

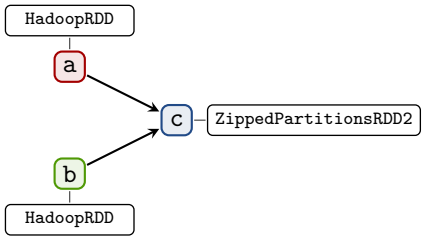


Figure 2.13: *Logical execution plan (lineage graph) for* `c = a.zip(b).collect`.

- **`DAGScheduler`:** The DAG scheduler compiles the logical execution plan into a physical execution plan. The physical execution plan for Figure 2.13 is shown in Figure 2.14. This plan contains **stages** with executable functions that perform the (user-)specified transformation on the input RDDs. Each stage is further split up into a configurable number of tasks, one per RDD partition. Each task independently executes the specified function on one partition. This is Spark's way to parallelize execution.

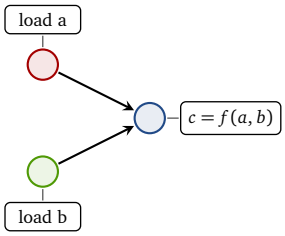


Figure 2.14: *Physical execution plan for* `c = a.zip(b).collect`. *Here,* $f(a, b) \rightarrow ((a_i, b_i))$, *with* $0 \leq i < I$

  To execute the physical plan, the DAG scheduler traverses it in reverse order until it finds a stage without any missing dependencies. All tasks of stage are sent to the task scheduler. Stages are executed in sequence unless more executors than executable task of a single stage exist in which case another stage can be executed at the same time. The DAG scheduler also emits status events, such as stage ready for execution and task finished.

- **`TaskScheduler`:** The task scheduler schedules tasks of a stage on available executors. It tries to find executors that are as close to the input data as possible in order to reduce data transfer overheads.

- **`CoarseGrainedSchedulerBackend`:** The local relay that forwards commands and events to and from an executor.

- **`CoarseGrainedExecutorBackend`:** The main class of the long-running executors. It is exclusively associated with a `CoarseGrainedSchedulerBackend` of a specific application from which it receives commands and task descriptions (task binary

and closure, i.e., global variables and call arguments) and returns task results and events. `CoarseGrainedExecutorBackend` can execute multiple tasks concurrently. `CoarseGrainedExecutorBackend`s are started and stopped by the resource manager.

- **`ExecutorAllocationManager`:** The interface between Spark and an external resource manager, such as YARN or Mesos. The executor allocation manager listens to stage and task status updates to determine the current task load. If more executable tasks than executors exist, it asks the resource manager to start more `CoarseGrainedExecutorBackend` instances until a configurable limit is reached. It does so in multiple rounds which last a configurable number of seconds (1s per default). During each round, it doubles the number of requested executors compared to the previous round, until the number of executable tasks is larger than or equal the number of executors. If the number of executors exceeds the number of executable tasks for a configurable amount of time (60s per default), executors are released.

2.4 Definitions

This section provides definitions to important terms used throughout this work. Definitions are grouped into categories: General (Table 2.3), execution (Table 2.4), scheduling (Table 2.5) and ML (Table 2.6). Terms related to heterogeneity have been defined in Table 2.1.

Term	Definition
Work	Execution of an algorithm to solve a problem.
Physical resource	Physical components of a computer system, such as CPU, memory and I/O attached devices (disks, network adapters). Physical resources are of fixed quantity and can be shared across and allocated by processes.
Virtual resource	An entity that allocates quantities of one or more physical resource and can itself be allocated. Virtual resources can be created and destroyed.
Executor	An executor $E = (C, M)$ is a long-running process that allocates C compute and M memory resources for the execution of tasks. Executors are virtual resources. Executors may use an additional, unspecific amounts of resources to manage the execution of tasks.
Resource allocation time	Amount of time t_{alloc} a resource is allocated.

Term	Definition
Resource busy time	Amount of time $t_{busy} \leq t_{alloc}$ a resource is executing a task while allocated.
Resource idle time	Amount of time $t_{idle} = t_{alloc} - t_{busy}$ a resource is allocated but not busy. It does not include time resources are not allocated.
Resource utilization	The resource utilization r_{util} is the fraction of time t_{busy} a resource is busy relative to the time it is allocated t_{alloc}, i.e., $r_{util} = t_{busy}/t_{alloc}$.
Resource utilization efficiency	The resource utilization efficiency r_{eff} is the ratio of work w performed to the resources allocated r_{alloc} for a given amount of time t_{alloc}, i.e., $r_{eff} = w/(r_{alloc} \times t_{alloc})$. r_{eff} cannot be measured in absolute terms, as w is abstract, but only relative to another r'_{eff}: $$\begin{aligned}\Delta r_{eff} &= \frac{r_{eff}}{r'_{eff}} \\ &= \frac{\frac{w}{r_{alloc} \times t_{alloc}}}{\frac{w}{r'_{alloc} \times t'_{alloc}}} \\ &= \frac{r'_{alloc} \times t'_{alloc}}{r_{alloc} \times t_{alloc}}\end{aligned}$$ r_{eff} utilizes resources Δr_{eff} as efficient as r'_{eff}.
Task	A task executes a function on input data and produces output data.
Stage	A stage is a set of tasks that all execute the same function on different partitions of a dataset. A stage is part of a job.
Job	A DAG of stages that can be executed independently. An application consists of one or more jobs. *Note: This definition of the term job is not used consistently in related work. A job can also refer to an application. Here, the Spark terminology is used.*
Data parallelism	The parallel processing of disjunct partitions of a dataset on parallel or distributed systems.
Task wave	A set of tasks of the same stage that run concurrently.

Term	Definition
Node performance profile	A vector that specifies the performance of nodes relative to a reference.
Node availability profile	A list of tuples. Each tuple contains a node, a timestamp and event (add or remove). It specifies the availability of nodes over a period of time.

Table 2.3: *General definitions.*

Term	Definition
Ready, Executable	A task, stage or application is ready or executable if all of its control and data dependencies are met.
Pending	A task, stage or application is waiting to be executed.
Executing	A task, stage or application is currently being executed.
Finished	A task, stage or application has finished execution.

Table 2.4: *Execution-state-related definitions.*

Term	Definition
Task scheduler	A scheduler that schedules individual tasks
DAG scheduler	A scheduler that schedules DAGs and uses a task scheduler to schedule individual tasks.
Application scheduler	A scheduler that schedules entire applications, that consist of one or more DAGs and tasks. It uses a DAG and task scheduler to schedule DAGs and tasks.
Global scheduler	A scheduler that schedules all applications running on a cluster at the same time. A global scheduler may only provide resources to applications or directly schedule applications, DAGs and tasks on resources.
Resource manager	A software that controls access to resources of a cluster and assigns them to applications using a set of policies.

Table 2.5: *Scheduler-related definitions.*

Term	Definition
Training	The process of finding parameters of a ML model such that an objective function is minimized or maximized.
Iteration	Round of global communication across all tasks. In each round, training samples are processed independently by tasks.
Epoch	One pass over all training data.
Feature	Measurable property of a phenomenon.
Label	Output data for a given set of feature values.
Test/training/validation accuracy	The fraction of test/training/validation set samples for which the model predicted the correct label.
Convergence rate	The rate with which the training algorithm converges towards an optimum. Convergence rate can be measured over epochs or time.
Training dataset	Set of samples that is used to train the model.
Validation dataset	Set of samples to validate the model during hyperparameter tuning and to monitor training progress.
Test dataset	Set of samples to validate the final model.
(Local) solver	An algorithm that is executed by each task to compute a model update during each training iteration.

Table 2.6: *ML-related definitions.*

2.5 Explanation of diagrams

This section explains types of diagrams used throughout this thesis.

2.5.1 Mira

2.5.1.1 Executor allocation diagrams

Figure 2.15 shows an example of an executor allocation diagram. It shows number of tasks and executors on the y-axis and time on the x-axis.

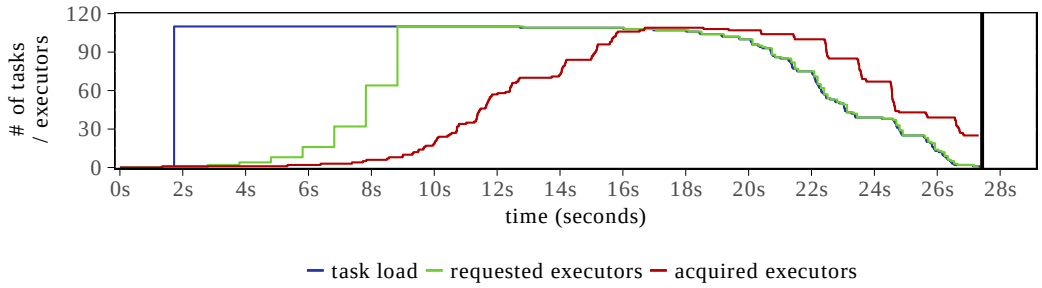


Figure 2.15: *Task load and executor allocation diagram.*

The diagram shows three plot lines:

- The blue task load line shows the number of tasks that are executable or executing. Once a task has finished, it does not count towards the task load anymore.
- The green line shows the number of executors that are requested from the resource manager. A request is kept active until the executor is not needed anymore, hence once an executor has been acquired, it still counts towards the number of requested executors.
- The red line shows the number of executors that have been acquired and can execute tasks.

The y-axis is scaled to fit the maximal task load. The black vertical bar towards the right end of the diagram represents the application end.

2.5.2 Chicle

2.5.2.1 Swimlane diagrams

Swimlane diagrams are used to visualize task execution and workload balance on nodes. Figure 2.16 shows an example of a swimlane diagram with three separate plots. Each plot consists of 16 rows - one for each node.

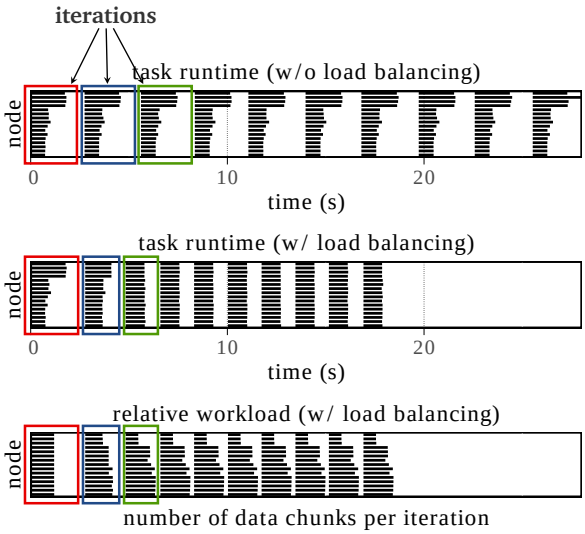


Figure 2.16: *Example of swimlane diagram used in Chapter 5. Here, it depicts the load balancing process on a heterogeneous cluster. The first three iterations are highlighted in each plot.*

The top two plots depict the task execution with (first plot) and without (second plot) a certain feature (here: load balancing). The x-axis shows the time in seconds and the y-axis the nodes. Black horizontal bars in each row represent the time a task is busy. Horizontal space in-between black bars represents the time a task is idle, i.e., waiting or communicating. One iteration lasts from the beginning of one black bar to the beginning of the next. The first three iterations are highlighted with red, blue and green boxes. For instance, during the first iteration (red box) tasks on the top four nodes run about twice as long as tasks on all other nodes, as depicted by the relative length of the bars. In the first plot, this difference remains across all iterations. In the second plot, the task runtimes are aligned during the second (blue box) and third (green box) iteration, as depicted by the equalization of the length of the bars.

The third plot shows the **relative workload of tasks (in number of data chunks) on each node on the x-axis and not time** per iteration for the second plot. For instance, during the first iteration (red box), the relative workload for all tasks is the same, as depicted by the equally long bars for the first iteration in the third plot. For the same iteration, the task runtimes vary across nodes (second plot). During the second (blue box) and third (green box) iteration, workload is shifted between nodes: As task runtimes get shorter (longer), the corresponding workload decreases (increases) as depicted by the changing lengths of bars in the both plots. The lengths of each bar in the third plot accurately represents the number of data chunks of a task on a node for a specific iteration, relative to all other tasks on all other nodes for all other iterations.

The third plot is only present for load balancing experiments.

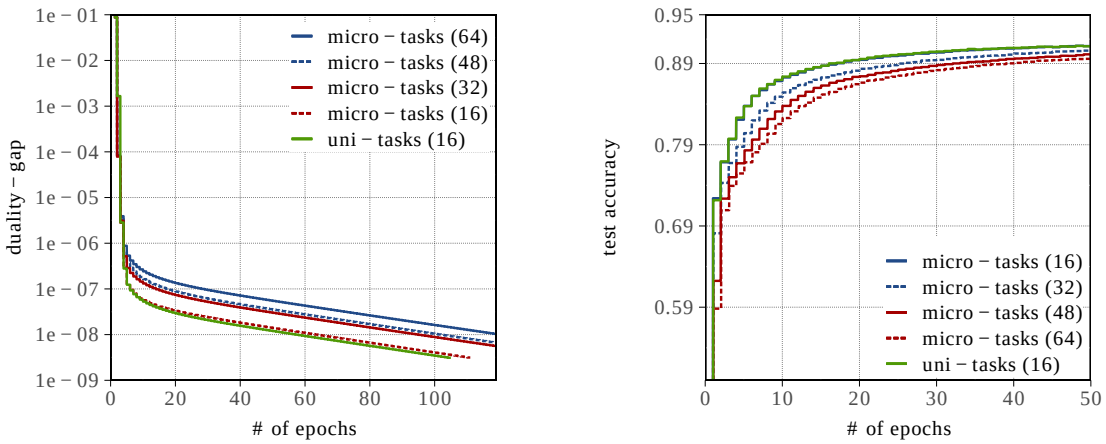
2.5.2.2 Convergence plots

Convergence plots are used to visualize the convergence of ML training algorithms with a convergence metric (duality-gap or test accuracy) on the y-axis and time or number of epochs on the x-axis.

For each dataset, a target duality-gap or test accuracy is defined. In general, plots are scaled such that:

- The x-axis covers the range until the last plot line has reached the target.
- The y-axis range covers the range until after the target.

In some cases, *interesting* parts are not clearly visible with this scaling and plots are zoomed in.



(a) CoCoA **(b)** Mini-batch SGD

Figure 2.17: *Example of a convergence plot for (a) CoCoA, where a lower duality-gap is better and (b) mini-batch SGD, where a higher test accuracy is better. Here, it depicts the load balancing process on a heterogeneous cluster.*

Each plot line represents the average convergence over multiple runs for a single type of experiment. If convergence is plotted over the number of epochs, values of all runs are directly averaged. If convergence is plotted over time, this is not possible, as the time the convergence metric is measured varies. Here, the value of the convergence metric is interpolated at fixed points across the plotted time range. 100 points are used per second. The interpolation method used is R `approx` function with `method=''constant''`. Values for each run are subsequently averaged at said fixed points and plotted.

Tests were typically executed for a fixed period of time, hence if the x-axis shows the number of epochs, plot lines may end at different points, as the time per epoch differs across experiments.

3. Efficient resource utilization within applications

This chapter presents work on the scheduling of distributed heterogeneous applications on heterogeneous clusters.[1] Heterogeneous applications contain heterogeneous directed acyclic graphs (DAGs) and tasks that exhibit varying resource demands throughout the execution. Heterogeneous clusters contain nodes with different types of hardware resources. When ignoring heterogeneity in applications or hardware, scheduling decisions can lead to prolonged application execution and inefficient resource utilization.

Heterogeneity in clusters is not a new phenomenon. Owned and operated by large companies and research organizations, large clusters have always been used for the execution of large scale, distributed applications. Over time, clusters were partially updated and extended with newer, faster nodes, or such with different capabilities, e.g., GPUs. This introduced heterogeneity into clusters. Cluster managers and schedulers, such as LSF [120], often use separate queues for different types of nodes to hide heterogeneity from applications. This has changed with the advent of the cloud. A study from 2012 [27] has shown that instance types (e.g., *m1.large*) on the Amazon cloud may be backed by as many as five different generations and models of CPUs, even with CPUs from different manufacturers. As the user has no influence over the specific hardware model, distributed application frameworks that run in such cloud environments need to be aware of or even exploit hardware heterogeneity in order to efficiently utilize these resources.

Furthermore, many recent distributed application frameworks, such as Spark [18], Flink [52] and Tez [59], internally represent applications as DAGs. Each path of the DAG executes different functions on different data, resulting in heterogeneous resource requirements and execution durations. Most of these frameworks, though, focus on scalability [57], ignoring either kind of heterogeneity, at the cost of resource utilization efficiency. At the current scale of the cloud, this approach is not sustainable anymore in terms of total cost, energy consumption and the impact on the environment. For this reason, it is important to address heterogeneity-awareness shortcomings of these frameworks and their schedulers. The goal of the work presented in this chapter is therefore to devise, implement and evaluate methods to improve schedules of applications, such

[1] The work presented in this chapter is based on the publications "The HCl Scheduler: Going all-in on Heterogeneity" [84] and in part on "Mira: Sharing Resources for Distributed Analytics at Small Timescales" [107].

that they utilize heterogeneous hardware resources more efficiently, without prolonging application runtimes.

Work presented in this chapter shows the benefits of DAG-, task-, hardware-heterogeneity-aware scheduling strategies for DAG-based distributed application frameworks at the example of Spark. Due to the complexity of distributed application frameworks, initial experiments with HCL have been performed in simulation, which demonstrated general benefits. For the simulation, it was assumed that task runtime is independent of which tasks were previously executed on the same resource (executor state). This is a commonly made assumption (see Section 3.5 for details). The subsequent task runtime analysis on a real cluster has shown, however, that due optimization strategies employed by distributed application frameworks, executor state significantly impacts task runtime, hence this assumption does not hold. In consequence, a new scheduling algorithm, stage packing (SP), has been devised which shows the benefits of state-heterogeneity-aware scheduling. SP is based on insights gained from the task runtime analysis and implemented in HCL-SP and integrated into Spark.

The main contributions of this chapter are summarized as follows:

(1) A detailed analysis of task runtime behavior and the predictability thereof, for a set of benchmarks (TPC-DS [88]) executed on Spark [18]. It resulted in the insight that executor state is a main determinator of task runtime, next to the node class.

(2) A novel scheduling technique, stage packing, that exploits insights gained from the prior analysis. The evaluation of stage packing using the TPC-DS benchmark suite on a heterogeneous 15 node test cluster showed that application runtime can be reduced by $\approx$1.4$\times$ on average, while resource utilization efficiency increases by the same factor.

This chapter is structured as follows: A brief introduction into the problem of scheduling heterogeneous applications on heterogeneous clusters is introduced in Section 3.1, followed by three main sections:

(1) The HCL scheduler and a simulative evaluation of heterogeneity-aware scheduling of Spark application is presented in Section 3.2.

(2) A detailed analysis of task runtime behavior of Spark applications on a real heterogeneous test cluster and the predictability thereof is presented in Section 3.3.

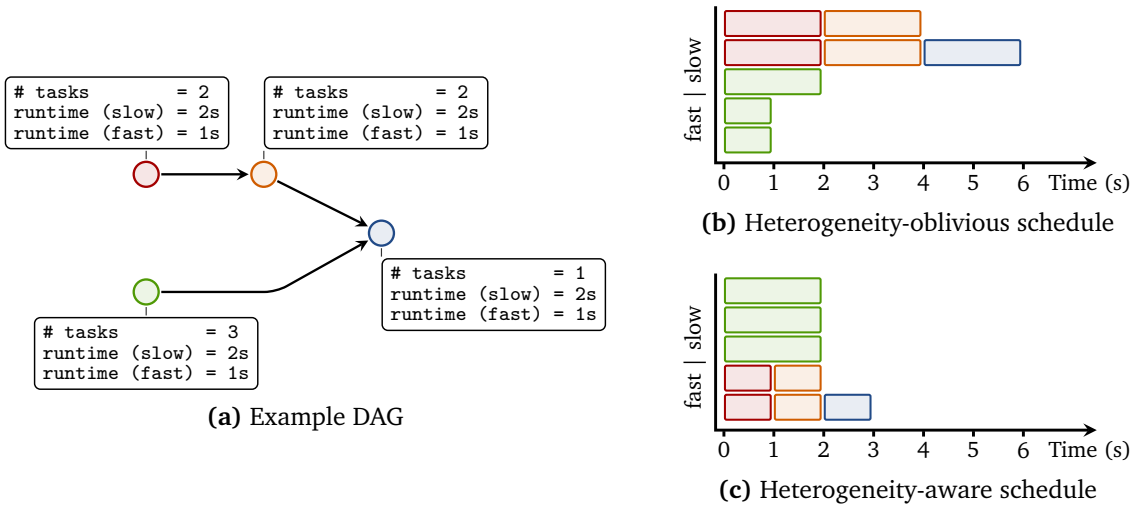
(3) The stage-packing scheduling technique and its implementation in the HCL-SP scheduler is described and evaluated in Section 3.4.

Subsequently, related work (Section 3.5) is presented and the chapter is concluded (Section 3.6). Section A.1 accompanies this chapter with additional information.

3.1 Introduction

A major issue for the efficient execution of distributed applications is how to allocate hardware resources to applications and their tasks. This issue has drawn lots of research efforts over the last years [9, 17, 21, 39, 38, 36, 37, 60, 54, 56, 65, 67, 66]. Many schedulers for distributed systems have been built under the simplifying assumptions of DAG, task and hardware homogeneity. These assumptions do not hold anymore. However, giving up on these assumptions increases the complexity of the scheduling problem a trade-off between the overhead of making scheduling decisions and the cost of making sub-optimal scheduling decisions has to be made. In the face of DAG-, task- and hardware-heterogeneity, the latter increases.

Figure 3.1 shows a simple example of an DAG-, task- and hardware-heterogeneity-oblivious and -aware schedule for a four-stage application DAG[2] on a five node cluster with two fast and three slow nodes.



(a) Example DAG

(b) Heterogeneity-oblivious schedule

(c) Heterogeneity-aware schedule

Figure 3.1: *Example of DAG-, task- and hardware-heterogeneity-oblivious and -aware schedules of a simple application with four stages on a cluster with two fast and three slow nodes (resources). Tasks and stages are color-coded.*

A DAG-, task- and hardware-heterogeneity-oblivious scheduler (such as the one in Spark) (Figure 3.1b) selects resources simply according to the availability of idle resources or according to data-locality where child tasks are scheduled on resources closest to the input data. It does not recognize that scheduling a three-task stage (green) on two fast and one slow resource is not beneficial but slows down the other path of the DAG. A DAG-, task- and hardware-aware scheduler (Figure 3.1c), on the other hand, can recognize this and schedule the three-task stage (green) non-greedily on the three slow nodes, which allows the other path to use the two fast nodes, reducing the application runtime.

[2]See Section 2.2.2.2 for details on application representation.

3.2 HCL

HCL (Heterogeneous CLuster) is a DAG-, task- and hardware-heterogeneity-aware application scheduler. In contrast to most related work (Section 3.5), HCL assumes that sharing resources across applications incurs high overheads and should be avoided. Instead, it schedules individual applications on heterogeneous clusters and improves application runtime using task runtime predictions.

The main objective of HCL is to evaluate potential gains of DAG-, task- and hardware-heterogeneity-aware scheduling strategies for distributed applications. Therefore, some challenges have been excluded intentionally, such as collection of runtime metrics and task runtime prediction. Moreover, to avoid extensive development efforts that are incurred by integration into a real distributed application framework (AF), such as Spark, HCL uses a simulator to simulate task execution. Figure 3.2 shows an overview of HCL.

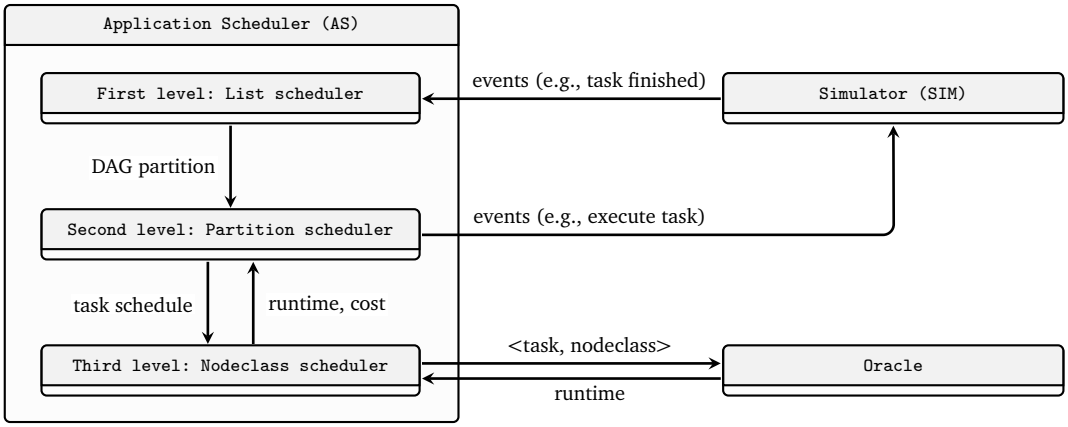


Figure 3.2: *Overview over HCL.*

HCL consists of three main components:

(1) An application scheduler (AS) that implements the DAG and task scheduler. The AS itself consists of three scheduling levels and uses task runtime predictions from the oracle and a cluster model to compute DAG-, task- and hardware-heterogeneity-aware schedules for each task.

(2) An oracle, that produces task runtime predictions based on previously recorded traces of real applications running on Spark.

(3) A simulator (SIM) that mimics a real AF, such as Spark. It sends and receives events, e.g., *task finished*, *execute task*, similar to an AF.

In the following sections, each component is described.

3.2.1 Assumptions

The following assumptions are made:

- The runtime of a specific task only depends on the used resource, the location of the input data and the communication bandwidth. It explicitly does not depend on any state. The latter is a common (albeit often implicit) assumption [3, 14, 18, 40, 65, 67, 66].
- Task runtimes can be predicted accurately.

3.2.2 Application scheduler (AS)

The AS is responsible for scheduling a single application on the simulated cluster. The AS makes few assumptions on how a *good* schedule looks like. However, due to the inherent complexity of DAG scheduling – even simple DAG scheduling problems are NP-hard [7] – heuristics had to be implemented to compute schedules in a reasonable amount of time (several minutes) even in simulation. To that end, HCL implements a three level scheduling algorithm:

(1) A list scheduler [7] (Section 3.2.2.1), similar to Bittencourt et al. [14] reduces the complexity of the scheduling problem. List schedulers rank tasks according to some metric (e.g., critical path) and select resources for the highest ranked task. Instead of tasks, DAG partitions of limited depth are ranked and resources are selected for an entire partition instead of individual tasks. This constitutes a compromise between complexity and context that the scheduler can used to make decisions.

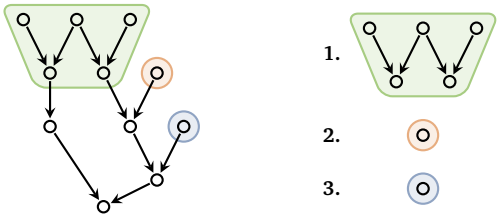
(2) A partition scheduler (Section 3.2.2.2) maps all tasks of a partition to node classes. A node class contains nodes that are hardware-wise identical (homogeneous) and have pair-wise identical communication cost. Node classes are used to reduce computational complexity. A random-tree-walk algorithm reduces the end time for the partition. The node class scheduler is used to compute schedules of a mapping. Tasks are executed according to the schedule with the lowest end time.

(3) A greedy, data-locality-aware node class scheduler (Section 3.2.2.3) places tasks on executors within a node class and computes the expected end time for each task. The necessary task runtime predictions are provided by the oracle.

In the following, all three levels are described in detail.

3.2.2.1 First level: list scheduler

The first level list scheduler partitions the application DAG into partitions of limited depth and ranks them. Partitions are passed to the second level scheduler in ranking order. Figure 3.3 visualizes both steps.



(a) Partitioning **(b)** Ranking

Figure 3.3: *List scheduler: Partitioning and ranking.*

Partitioning. Partitions are independently schedulable sub-DAGs, i.e., they cannot have unmet external dependencies. Their sole purpose is to reduce the search space for the second level scheduler during schedule computation, while still providing context to each task, e.g., child, parent and sibling relationships. Tasks in the partition are assumed to influence each other, either due to data dependencies or because they can be executed at the same time. Tasks outside the context are assumed to not do so, as they are only remotely related to tasks inside the partition. Knowing this context allows the second level scheduler to consider the impact of scheduling decisions on the most affected tasks while ignoring tasks that are less affected.

The depth of a partition is determined by a configurable range as well as the DAG structure. The first-level scheduler always attempts to build partitions with at least the minimum depth, if possible. However, in some cases (for instance for the red and blue partitions in Figure 3.3), this is not possible as unmet data dependencies would remain once the maximal depth is reached.

In-between the minimal and maximal depth, the first-level scheduler extends partitions as long as tasks are convergence points. Convergence points are of particular importance as paths merge there and waiting times in front of barriers at convergence points can arise if path end times are not balanced. If all converging paths are part of the partition, the second level scheduler can balance path end times.

Partitions are constructed incrementally using three properties of DAG nodes (which are tasks):

- *uprank*: the length of the longest path from a node to an input node. An uprank of zero indicates that the node is an input node.
- *downrank*: the length of the longest path from a node to the output node. A downrank of zero indicates that the node is the output node. There can be only one output node.
- *depth*: the depth at which it was inserted into the partition.

After sorting nodes with an uprank of zero (input nodes) in descending order of their downrank, a partition is created for the first node and grown from there on by adding

child nodes and their ancestors. A child can only be added if the following conditions are true:

- max(depth, uprank) of a task is smaller or equal the minimum partition depth or is smaller or equal the maximum partition depth and is a convergence point.
- none of its ancestors is part of another partition.

Listing 3.1 lists the top-level partitioning function of an application, which uses the `grow` function in Listing 3.3 to grow the partition to its full size. Tasks are added to a partition using the `addTask` function in Listing 3.2.

```
1  void Application::partition(int min, int max) {
2    list<Task*> sortedInput; // subset of tasks with uprank == 0, sorted in descending
3                             // order of their downrank.
4
5    for (task : sortedInput) {
6      if (task->partition != NULL) // Skip tasks that are already part of a partition.
7        continue;
8
9      Partition* p = new Partition();
10     this->addPartition(p);       // add partition 'p' to application
11     p->addTask(task, 0);         // add 'task' to new partition 'p' at depth 0
12
13     // Grow 'p' from its frontier, a set of newly added and not fully explored tasks.
14     while (p->frontier.size() > 0) {
15       task = p->frontier.begin();
16       p->grow(task, task->depth(), min, max);
17       p->frontier.erase(task);
18     }
19   }
20 }
```

Listing 3.1: *Simplified C++ code of the DAG partitioning function.*

```
void Partition::addTask(Task* task, int depth) {
  // There might be other children to add to this partition.  Add it to the frontier
  // for further exploration.
  if (max(depth, child->uprank()) <= min)
    this->frontier.push_back(task);

  task->setDepth(depth); // Remember at which depth the task was added to the partition.

  for (parent : task->parents()) {
    if (parent->partition == NULL) // skip parents that are already in the partition.
      this->addTask(parent, depth-1);
  }
}
```

Listing 3.2: *Simplified C++ code of the DAG partition addTask function.*

```
void Partition::grow(Task* seed, int depth, int min, int max) {
  for (child : seed->children()) {
    // Skip children that are already part of a partition (can only be this partition)
    if (child->partition != NULL)
      continue;
    // Don't consider children if it would extend the partition depth beyond the
    // maximum.
    if (max(depth, child->uprank()) > max)
      continue;
    // Don't consider children if it would extend the partition depth beyond the
    // minimum, except if they are a convergence point.
    if (max(depth, child->uprank()) > min && child->parents.size() <= 1)
      continue;
    // Recursively check partitions of ancestors. Only children whose ancestors are
    // either in this partition or in no partition can be added.
    if (!checkAncestorPartitions(child, this))
      continue;

    // Child has passed all tests and can be added.  All ancestors are added as well.
    this->addTask(child, depth+1);
  }
}
```

Listing 3.3: *Simplified C++ code of the DAG partition grow function.*

Whenever a task finishes the partitioning is re-evaluated and partitions are updated if their depth has changed.

Ranking. Partitions are ranked according to their priority. The priority ρ of a partition P is the sum of the average runtime τ_t^{avg} of all tasks $t \in P$ on all resources, weighted by the uprank r_t^{up} of a task and computed as follows:

$$\rho = \sum_{t \in P} \frac{\tau_t^{avg}}{1 + r_t^{up}} \tag{3.1}$$

The higher the uprank r_t^{up}, the later a task will be executed and the more uncertain the schedule becomes, which is accounted for by the weight $1/(1 + r_t^{up})$.

Another choice to determine the ranking would be the critical path of the application DAG. However, the critical path only considers the runtime of individual tasks. This is inadequate, if partitions contain many small tasks. If not all tasks can run in parallel, because their number exceeds that of available resources, the necessary runtime of a partition increases without increasing the critical path length.

3.2.2.2 Second level: partition scheduler

The partition scheduler implements lookahead scheduling by considering future tasks of a partition as well as the context of tasks (i.e., sibling tasks). Lookahead scheduling can improve schedules in the following conditions:

- If a large amount of data is transferred between a parent and child task, the parent task can either be scheduled on or near the preferred resource of the child task (e.g., if the child requires a GPU and the parents doesn't) or the resource of the parent task can be reserved for its child task and not used by other tasks to *reduce data transfer overheads*.

- In case where the number of fast resources is not sufficient to reduce the time of all tasks to each a barrier, the scheduler can decide to schedule all tasks on slower resources and use fast resources for other tasks, as shown in Figure 3.1.

The scheduling algorithm implemented here maps tasks to node classes using a random walk across a tree, which contains all possible mappings for all tasks to all node classes. The tree for a partition DAG with n levels (i.e., of depth n) has $n + 1$ levels. Levels 1 to n in the tree correspond to levels 1 to n of the partition DAG. An additional root node is placed at level 0 of the tree. A partition depth of n results in a lookahead value of $n-1$. A path from the root node to any leaf node contains mappings for all tasks of a partition to node classes. These mappings are translated into a schedule by the node class scheduler. Schedules for N randomly selected paths are computed and compared w.r.t. their length. The shortest schedule is executed. N is configurable.

Listing 3.4 shows the code of the random tree walk function. It traverses the mapping tree in a depth-first manner, generates mappings and computes schedules for them level by level. Once a leaf has been reached, the length of the best schedule is compared to

that of the latest schedule, which replaces it, if it is shorter. After mappings have been explored, the first ready task on each executor is executed. The schedule is recomputed when partitions are updated by the first level scheduler.

```
// The shortest best (shortest) schedule and its end time, the latter is initialized
// with infinity.
Schedule bestSchedule(∞);

void Partition::walk(Partition* p, Cluster* c, int level, int N) {
  vector<Task*>      tasks       = p->getTasksOnLevel(level);
  vector<NodeClass*> nodeclasses = c->getNodeclasses();

  int maxIndex = pow(nodeclasses.size(), tasks.size()); // # of node classes ^ # of tasks.

  for (int step = 0; step < min(N, maxIndex); step++) {
    // Randomly select a tasks-to-nodeclasses mapping (for function see Listing 3.5) for
    // all tasks on the current level. A mapping corresponds to a node in the tree.
    // If, however, the number of possible indices is smaller than the maximal number
    // of steps 'N', then iterate over all mappings sequentially.
    int index    = (N < maxIndex) ? random() % maxIndex : step;
    map<NodeClass*, list<Task*>> mapping = getMapping(index, tasks, nodeclasses);

    int endTime = 0;
    // Generate schedules for that mapping
    for (pair<NodeClass*, list<Task*>> m : mapping) {
      NodeClass*  nc    = m.first;
      list<Task*> tasks = m.second;
      // Ask the node class scheduler to compute a schedule for the current path
      // section.  This call changes the internal state of the node class scheduler as
      // it needs to remember the schedule to correctly schedule subsequent levels. It
      // returns the end time of the schedule.
      endTime = max(endTime, nc->schedule(level, tasks));
    }

    if (level < p->depth()) {
      walk(p, c, level+1, N); // Explore mappings for the next level
    } else {
      // This is the last level, i.e., the schedule is complete.  Check if it is
      // shorter than the current best schedule.
      if (endTime < bestSchedule.endTime()) {
        bestSchedule.resetSchedule(); // discard old schedule
        // Save schedules of all node classes in the 'bestSchedule'. 'getSchedule()'
        // returns the internal state of the node class schedule.
        for(NodeClass* nc : nodeclasses) {
          bestSchedule.addSchedule(nc->getSchedule()); // add new best schedule
    }}}

    // Remove schedules for tasks of this level as a different mapping for the same
    // level will be explored next.
    for (pair<NodeClass*, list<Task*>> m : mapping) { nc->resetSchedule(level); }
  }
}
```

Listing 3.4: *Simplified C++ code of the partition scheduler random-tree-walk function.*

```
map<NodeClass*, list<Task*>> Partition::getMapping(size_t index, vector<Task*> tasks,
                                                   vector<NodeClass*>nodeclasses) {
  map<NodeClass*, list<Task*>> mapping;

  for (Task* task : tasks) {
    size_t ncIndex = index % nodeclasses.count();

    // Some tasks may already be scheduled.  In that case, the node class cannot change
    // from what was selected previously. 'isScheduled()' remains true when the task is
    // being executed.
    if (task->isScheduled()) {
      ncIndex = task->nodeclass()->index(); // get index of node class it was scheduled on.
    }

    mapping[nodeclasses[ncIndex]].push_back(task);
    index /= nodeclass.count();
  }

  return mapping;
}
```

Listing 3.5: *Simplified C++ code of the partition scheduler tasks to node class mapping function.*

3.2.2.3 Third level: node class scheduler

The node class scheduler is a homogeneous, I/O-cost-aware task scheduler. As all nodes within a node class are hardware-wise identical, no hardware-heterogeneity-awareness is required here. It is further assumed that all executors are configured identically.

The node class scheduler schedules tasks on executors. To find the best executor for a task, it computes the earliest start time on each executor. The earliest start time depends on:

- The maximal actual or estimated finish time of any of its parents.
- The estimated I/O time to transfer input data from its origin to the selected executor. This depends on the size of the input data as well as the network link bandwidth.

The executor with the earliest start time is allocated. Listing 3.6 shows the task scheduling code.

```
void NodeClass::schedule(int level, list<Task*> tasks) {
  for (Task* task : tasks) {
    Executor* best;
    long      earliestStartTime = MAX_LONG;

    for (Executor* exec : executors) {
      // Determine latest finish time + data transfer times for each parent.  Each task
      // has a list of input edges 'in'.  Each input edge is a pair with a parent task and
      // the amount of data that it reads from the parent.
      list<pair<Task*, long>> predecessors = task->in();
      for (pair<Task*, long> p : predecessors) {
        Task* parent   = p.first;
        long  dataSize = p.second;

        // Determine end time of the parent (actual, if the parent has already
        // finished or estimated, otherwise).
        long t0 = parent->endTime();
        // Determine data transfer time w.r.t.\ minimal bandwidth of any link between
        // the parent executor and the current executor.
        long t1 = dataSize / getMinBandwidth(parent->executor(), exec);

        if (t0+t1 < earliestStartTime) {
          earliestStartTime = t0+t1;
          best              = exec;
        }
      }
    }

    // Allocate executor from the estimated earliest runtime for the predicted runtime.
    long        runtime = oracle->predictRuntime(task, best);
    Allocation* alloc   = allocate(task, best, earliestStartTime, runtime);

    // Remember this allocation and the corresponding level to be able to remove all
    // allocations for a level later.
    this->allocations[level].insert(alloc);
  }
}
```

Listing 3.6: *Simplified C++ code of the node class scheduling function.*

3.2.3 Oracle

The oracle uses pre-recorded application traces and produces exact task runtimes upon request via the `long Oracle::predictRuntime(Task* task, Nodeclass* nodeclass);` Application traces have been collected on a *fast* node class node and augmented with a *slow* node class by scaling fast runtime values by 1.5×. Task runtimes are identified by task and nodeclass, hence the oracle is a simple database.

3.2.4 Simulator

The simulator minics an AF and receives/sends events from/to the AS. Each event contains a time stamp at which it is executed. The simulator executes events in ascending order of their time stamp. Upon execution, a response event is generated. Once all events for a time stamp have been executed, a virtual clock is advanced to this time stamp. The only events are *execute task* and the only response is *task executed*, which is received by the AS. The virtual clock is used by the AS as lower bound for the earliest start time of a task.

3.2.5 Evaluation

The evaluation of HCL addresses the question of whether Spark applications could benefit from DAG-, task- and hardware-heterogeneity-aware scheduling. The experiments presented here show simulative results of a subset of TPC-DS queries [116], which have been scheduled using three different scheduling strategies:

(1) As baseline, a data-locality-aware but DAG-, task- and hardware-heterogeneity-oblivious scheduling strategy was used (*H/W oblivious*). Child tasks are placed on the same node as their parent tasks if free executors are available. This scheduling strategy is modelled after Spark's scheduler.

(2) Data-locality-, hardware- and task-heterogeneity-aware, but not DAG-aware scheduling (*no lookahead*). This scheduling strategy is modeled after that of Paragon [33] and Quasar [44], except without task-interference-awareness, as there is no interference in the simulation.

(3) Data-locality, hardware-, task- and DAG-heterogeneity-aware. This is HCL's scheduling approach . Minimal lookahead was always set to 1. Maximal lookahead set to 1, 2 and 3 to evaluate in how far lookahead scheduling can improve upon the previous, DAG-oblivious method (*lookahead 1–3*).

The traces that served as input to the scheduler were captured in advance, by executing the corresponding query on Spark on a single node with 8 executors. DAG models, task runtimes and I/O volumes were extracted from those traces and used by the oracle during the simulation. This setup was chosen to reduce the impact of data transfer on task runtimes, as HCL models data transfer time using the cluster model. Furthermore, the in the captured traces, data transfer times cannot be distinguished from data processing times, hence an attempt was made to reduce its impact on the task runtime. Spark version 2.2.1 was configured to use 8 shuffle partitions[3], which results in 8 tasks for most stages. Total number of tasks per application range from 40 to 132 (85 on average).

[3] `spark.sql.shuffle.partitions=8, spark.default.parallelism=8`

Application models were executed on a simulated cluster with 8 nodes and 4 executors each. 6 nodes were *slow*, while 2 were *fast*. Slow nodes take 1.5× as long to execute a task as fast nodes. All nodes are connected by a 1GBps connection. This represents a simple cluster where faster resources (e.g., high-end CPUs, GPUs, FPGAs), due to their higher cost and limited applicability (for accelerators), are only available in a limited number of nodes. Two node classes were used, one for fast and one for slow nodes. As reference for each schedule, a theoretical critical path of each DAG was computed, based on task runtimes on the fast nodes and ignoring I/O data transfer time. Hence, this constitutes a lower bound for the schedule length. Each run was repeated five times. The partition scheduling algorithm was configured to explore 50 nodes on each level of the mapping tree.

3.2.5.1 Results

Results in Figure 3.4 show that HCL is able to reduce the schedule length on average from 1.48× of the critical path, when using a heterogeneity-oblivious scheduler (baseline) to 1.10×, when using a maximal lookahead of three. The largest reduction in schedule length is achieved, however, by simply being hardware-heterogeneity-aware without looking ahead. Here, the schedule length is reduced to 1.20× of the critical path. While lookahead scheduling reduces this further, results are not consistently better. This is caused by the limited subset of explored schedules, which is necessary due to the time complexity of the scheduling algorithm.

This indicates that the benefit of looking ahead does not justify the effort that is needed to compute lookahead schedules. Nevertheless, results also show that there is potential for optimizing schedules of Spark applications by using a DAG-, task- and hardware-heterogeneity-aware scheduling strategy.

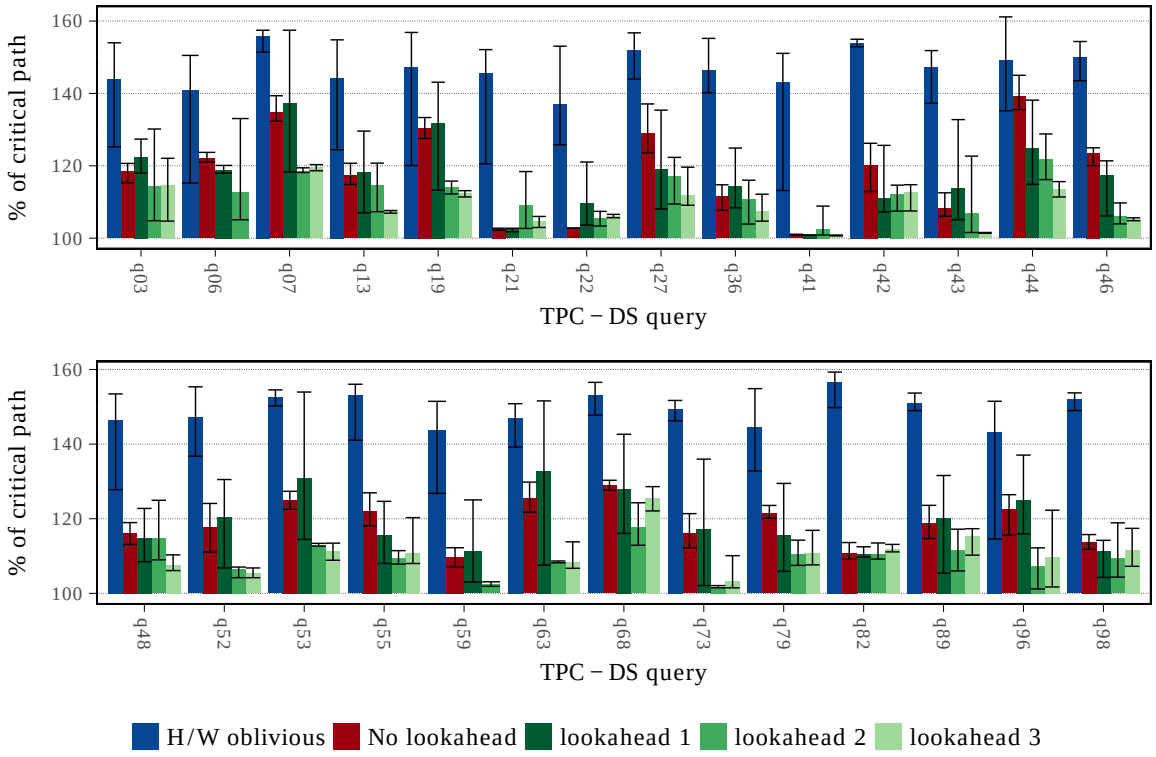


Figure 3.4: *Execution time of different scheduling algorithms relative to the execution time of the critical path. All values are mean values relative to the critical path length with min/max error bars.*

3.3 Task runtime prediction

The initial simulative results have shown that schedules can be improved using task runtime predictions. Hence the next step is to accurately predict task runtimes. This is a two-step process. First, metrics that correlate with the task runtime need to be identified and collected (Section 3.3.1) and second, a prediction model needs to be trained (Section 3.3.2) on said data.

Multiple methods exist to generate prediction models, e.g., closed-form analytical models can be created for each application and stage. Parameters can be used to model the magnitude that each metric impacts the task runtime. The problem is, however, to find parameters, such that the prediction error is minimized. Moreover, the type of function that represents the correlation of a metric to the task runtime may vary across applications and stages, for instance from a linear, to a quadratic or a step function. Machine learning (ML) offers a solution here, as ML training algorithms can find parameters for a model automatically, during the training process. Implementations of many training algorithms exist, which allows to change the type of function that represents these correlations with little effort, to generate prediction models for a variety of applications and stages. For instance, linear and quadratic models can represent linear and quadratic functions, whereas Classification and Regression Tree (CART) can represent step functions. For this reason, ML has been chosen to create prediction models.

The basic idea is that task runtimes are observed over application runs and, over time, the prediction model is improved based on these observations. The underlying assumption here is that many applications are recurring. Agrawal et al. [24] has shown that this is indeed the case for ≈40% of all applications in a data-center. In contrast to some prior work, no explicit benchmark runs [33, 44] or ahead-of-time simulations [26] are considered, as those can delay the execution of applications.

3.3.1 Identification of metrics and collection of data

In order to predict task runtimes, metrics, e.g., input data size, that correlate with the task runtime have to be identified and collected. Ideally, given the same values for all selected metrics across multiple measurements, the task runtime is the same as well. Furthermore, as values for these metrics are needed at scheduling time, for task runtime prediction, they need to be measurable (or computable) at scheduling time as well. Once identified, metrics need to be measured and values correlated across multiple runs without incurring high overheads.

The metrics listed in Table 3.1 have been identified for runtime characterization and later prediction. These metrics have been chosen as their value can be determined during scheduling time and due to their obvious impact on runtime (apart from the first two metrics). While not explicitly stated in related work [26, 45, 66], all but the first two metrics are common choices.

Metric	Type	Description
First task of a stage on an executor	categorical	Whether the current task is the first task of a stage that is executed on a certain executor. The first task needs to transfer data that is shared across all tasks of the same stage (e.g., broadcast variables, task binaries) and cannot benefit from a warm Just-In-Time (JIT) cache. Known by HCL.
First task on an executor	categorical	Whether the current task is the first task ever executed on an executor. The first task may trigger some initialization as well as the loading of additional libraries. Moreover, the JIT cache is still cold, thus even AF-provided functions have not been compiled yet. Known by HCL.
Application ID	categorical	Runtime behavior of tasks varies across applications. Provided by Spark.
Stage ID	categorical	A stage defines the function that tasks of a stage execute. Task runtime varies depending on the function they execute. The stage ID is provided by Spark, however, some modifications were required to make them stable across application runs (see Section 3.3.1.1).
Node class ID	categorical	The node class impacts the speed with which tasks are executed. Known by HCL.
Total input size	numerical	The total input size determines how often a function needs to be executed and therefore determines the task runtime. Provided by Spark.
Local node input size	numerical	The fraction of the total input size that is on the local node but in another executor and can be accessed without network I/O but still requires deserialization. Provided by Spark.
Local executor input size	numerical	The fraction of the total input size that is on the local executor and can be accessed without network I/O nor deserialization. Provided by Spark.
Remote input size	numerical	The fraction of the total input size that needs to be transferred across the network and therefore incurs network I/O overheads. Provided by Spark.
Task runtime	numerical	Time from start to end of task execution on Spark executors. Provided by Spark.

Table 3.1: *Metrics used for task runtime prediction.*

Prior experiments with various micro-benchmarks (e.g., word count, word filter) and TPC-DS queries have shown that the state of an executor can have a significant impact on the task runtime: Freshly started (cold) executors require significantly more time to execute a task than previously used (warm) executors. Additionally, once an executor has executed a task of a stage, the second and subsequent tasks of the same stage on the same (hot) executor can also see significant runtime benefits. As consequence, the first two metrics have been added to represent the executor state (cold, warm and hot). Wang et al. [61] makes a similar observation about the executor state. As noted before, most related work optimizes schedules across applications but assumes no resource reassignment costs across applications. Both metrics also reflect parts of this reassignment cost, next to the cost of shutting down and restarting an executor.

Additional parameters considered in related work relate to interference and to memory consumption tasks or applications. As Spark tasks are executed on uniformly configured executors, no memory consumption metrics are used. Since HCL is not interference-aware, no interference-related metrics have been collected either.

An important consideration when selecting metrics is that adding metrics comes at a cost. Every metric translates into one or more *features* of an ML model, which increases the complexity of the model. A more complex model requires more training samples and increases training as well as inference time.

3.3.1.1 Data collection

In order to improve the prediction model over time, the selected metrics need to be collected during every run and correlated to previous runs. Correlating metrics across runs requires that stages can be matched across runs, as they define the function that is executed by their tasks. Spark DAGs contain stages as nodes and data dependencies as edges. They are isomorphic across runs of the same application but not identical. This is because stage IDs in Spark are not guaranteed to be stable across runs.

A method had to be implemented in Spark[4] to compute stable IDs for each stage, by computing hashes over:

- Call site string, e.g., `count_words at example.scala:42`, which represents the line of code that initiated the creation of the stage.
- A list of hashes of all input sources (Resilient Distributed Datasets (RDDs), file paths) of a stage in order of their appearance.

This ID is not unique. Multiple stages may have the same ID if they are called from within a loop, on the same input data, but with different call parameters. The assumption is that stages with the same stable ID exhibit a similar runtime behavior, as they execute

[4]The corresponding changes had to be made in many places across the Spark source code. However, some changes were omitted due to their complexity, which is why only 90 out of 100 TPC-DS queries will be used for the evaluation, as the remaining 10 queries relied on the omitted changes.

the same function. Moreover, in cases where the same source file name exists multiple times, the call site string has to be extended by the full path of a file.

For the data collection, applications have been executed on Spark with HCL-SP as scheduler. During each execution, the above listed metrics are collected for each task and transmitted to HCL-SP via its REST API. Stages are identified by the stable ID. Metrics are recorded by HCL-SP's oracle component and stored in a comma separated values (CSV) file. This file is read by an external ML training application to train and update the task runtime prediction model.

3.3.1.2 Data analysis

For an initial analysis, the metrics listed in Table 3.1 have been collected for ten runs of each test application. For the analysis, the test cluster was divided into four node classes, one for each CPU model and clock frequency. Node classes and corresponding hardware descriptions are listed in Table 3.2.

Subsequently, the collected data has been analyzed w.r.t. the correlation between the collected metrics and the task runtime. Ideally, task runtimes would be the same across multiple runs w.r.t. to values of the collected metrics. As the task runtime is influenced by a multitude of factors, some variation is expected. However, a high degree of stability of task runtimes across multiple measurements is necessary, as the prediction model, that is trained on this data, can only be as good as the training data.

Test setup. All experiments use the test setup described in Section A.1.2.

Node class	CPU	Memory	OS
1	Intel Xeon E5-2650v2, 2.60GHz	160GiB	RHEL 7.5
2	Intel Xeon E5-2630v3, 2.40GHz	160GiB	RHEL 7.5
3	Intel Xeon E5-2640v3, 2.60GHz	256GiB	Fedora 26
4	Intel Xeon E5-2640v3, 1.20GHz	256GiB	CentOS 7.5

Table 3.2: *Node class to hardware translation.*

Test applications. For these experiments a Spark implementation of the TPC-DS benchmark suite is used. A description of this benchmark is in Section A.1.4.

Results. Plots in Figure 3.5 show the correlation between node classes (see Table 3.2 for a mapping to hardware), total input data size and executor state to the task runtime for a single stage. The selected seven examples represent frequently observed patterns among all 909 stages across all 90 TPC-DS queries.

Data for each of the four node classes is plotted separately side by side. Executor state is marked by differently colored circles, the size of which represents the relative number of measurements with a certain (total input size, task runtime) coordinate. Not every stage uses executors in all three states. Cold executors, for instance, can only be used by early stages[5] of an application, as they will have been used before (and thus be warm) once later stages get scheduled. Furthermore, depending on the number of executable tasks, some stages rarely use executors that have already executed a task of the same stage before, i.e., are hot. This is can be the case for stages with less tasks than available executors (here: 112), hence the plots show a non-uniform usage of executors in different states. Due to the heterogeneity across stages in the selected samples, input data and task runtime ranges vary to a degree that does not allow the presentation with uniformly scaled axes, hence both axes are scaled to the respective value ranges that have been measured for the shown stage.

For instance, in Figure 3.5a, a data loading stage of query 14a is depicted. The figure shows various aspects of the data:

- **X-axis:** Tasks load between zero and 379MiB of data with four main size values of approximately zero, 126, 252 and 379MiB. Two additional size values exist just below 126 and 252MiB, albeit for fewer tasks. The latter can be seen by smaller circles for the corresponding size values.

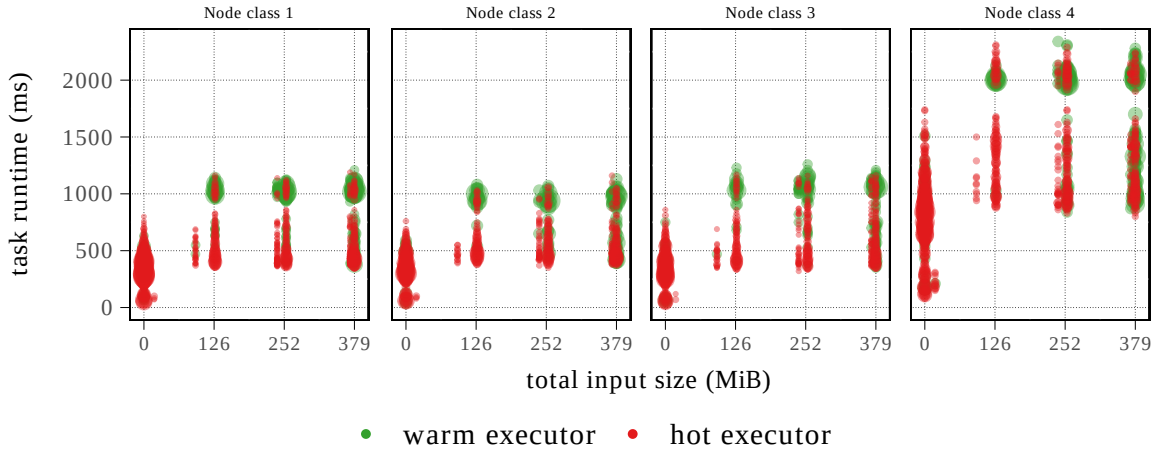
- **Y-axis:** For each node class plot, the task runtime spread is similar for all size values, with the exception of a total input size of zero. This indicates that there is no linear correlation between total input size and task runtime. The plot shows that the only influence on task runtime is whether total input size is zero or not zero.

- **Node class plots:** Tasks run for up to ≈1200ms on node classes 1 – 3 and ≈2400ms on node class 4 (right most plot). On node class 4, which corresponds to the slowest hardware, tasks run generally longer than on node classes 1 – 3. The input data sizes are similar across all four node classes.

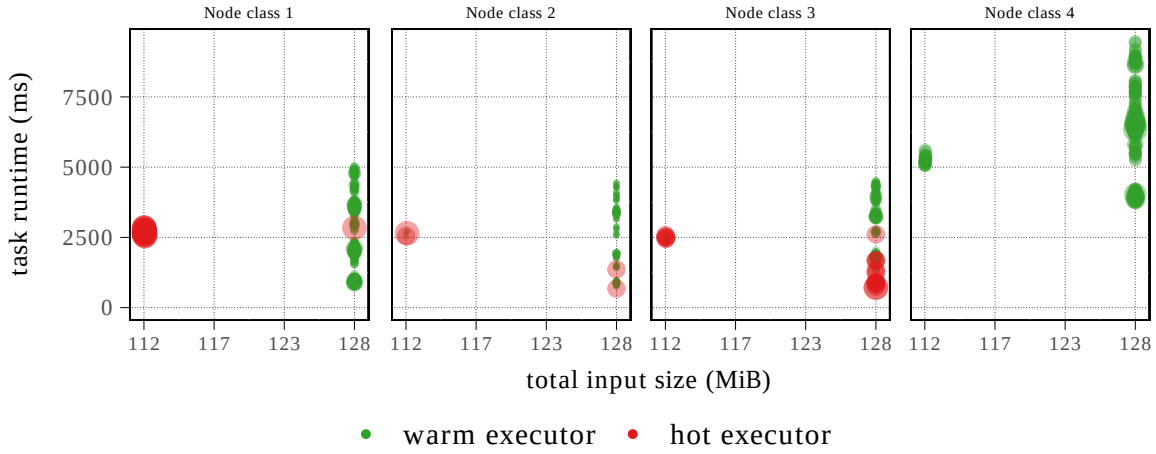
- **Color of circles:** Tasks run on warm (green circles) and on hot (red circles) executors. No task of this stage runs on cold executors (blue circles), hence no blue circles are shown in the plot.

- **Size of circles:** Green circles are larger on the upper end of the runtime range than the lower end, whereas red circles are larger on the lower end of the runtime range, indicating that tasks on warm executors generally run longer than those on hot executors.

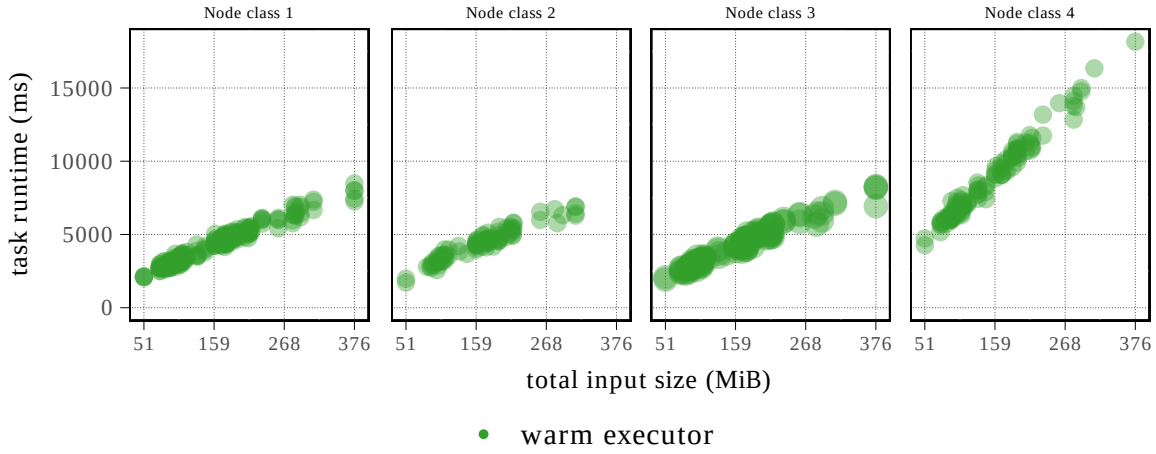
[5] Early stages refers to stages that are executed before an application reaches its peak level of parallelism.



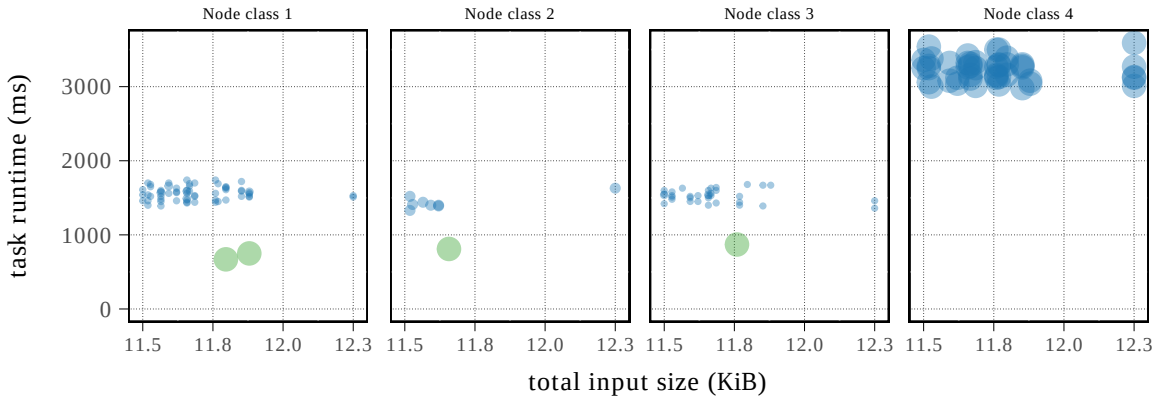
(a) Query 14a (load)



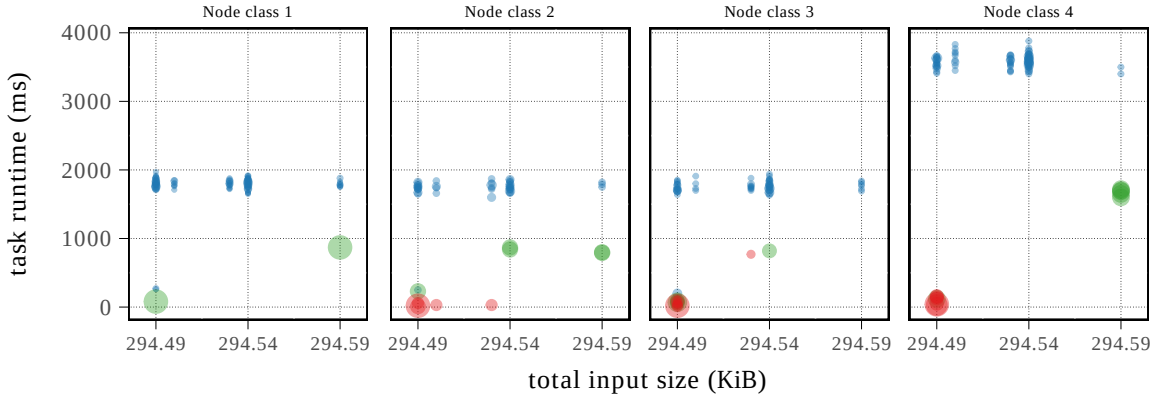
(b) Query 66 (load)



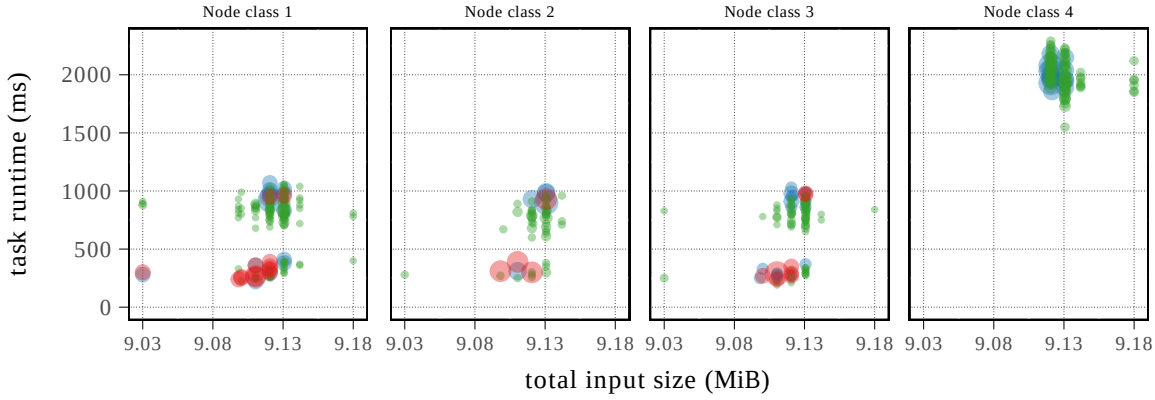
(c) Query 14a (compute)



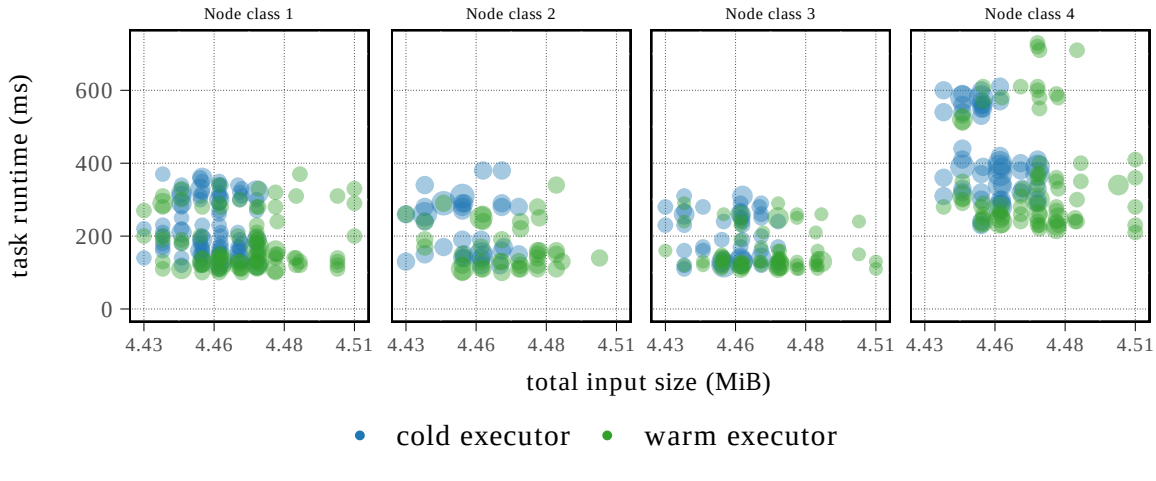
(d) Query 13 (load)



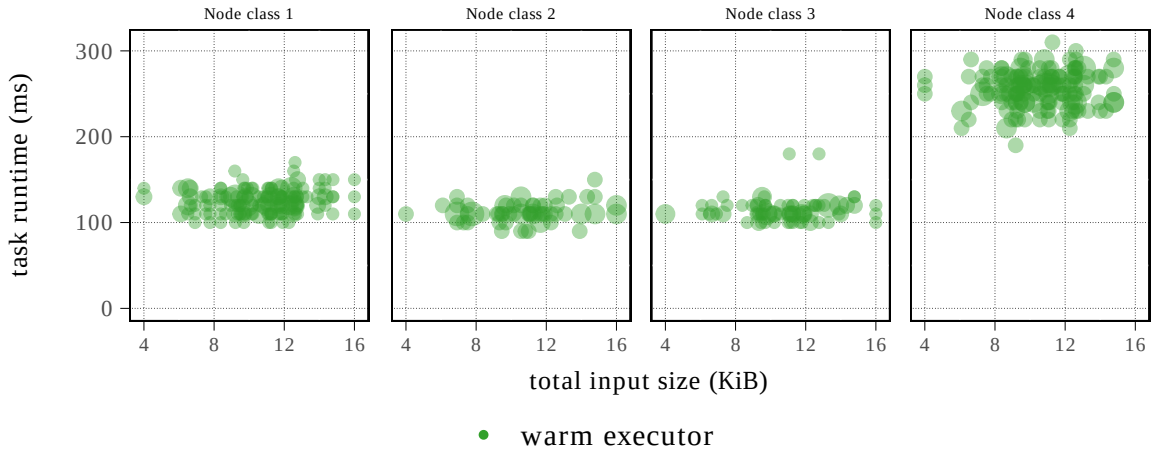
(e) Query 23a (load)



(f) Query 87 (compute)



(g) Query 38 (compute)



(h) Query 15 (compute)

Figure 3.5: *Examples of task runtime correlation w.r.t. total input data size and executor state for a selection of queries and stages. The type of function of a stage is given in parentheses.* Load *indicates that the main function of the stage is to load data from an HDFS file system, whereas* compute *indicates that its main function is to process/transform data. The size of each circle indicates the relative (to other measurements in the same sub-plot) frequency of a measurement.*

The data shown in Figure 3.5 provides insight into several aspects of task runtime behavior and the execution of Spark applications:

(1) As expected, **the node class has a noticeable impact on task runtime**. The slowed down CPUs take ≈2.3× longer to execute tasks than on the ones running at their regular frequency. However, even among nodes that run at their regular frequency, a performance variation of ≈10% has been observed.

(2) **The impact of the executor state on task runtime is significant** (Figures 3.5b, 3.5d, 3.5e, 3.5f and 3.5g). On average, tasks running on cold executors needed

≈2.1× longer than on warm executors and ≈15.4× longer than on hot executors. An analysis of this effect has unveiled that tasks of the same stage can share input data, which is only transferred and deserialized upon first access. Subsequent tasks can access the same stage without additional delay. Furthermore, the Java Virtual Machine (JVM) relies on JIT compilation to accelerate the execution of often used code. By executing the same function (i.e., tasks of the same stage) on an executor repeatedly, the JIT compiler is more likely to compile a function into native code. This accelerates execution of subsequent tasks of the same stage.

(3) For stages that process a wide spectrum of input data sizes by individual tasks (Figure 3.5c), a clear correlation between total input data size and task runtime can be observed. For stages that load data, no such correlation can be observed, even in cases with a wide spectrum of input data sizes (Figure 3.5a). However, a noticeable runtime drop for tasks that load zero bytes, hence don't actually access the file system, can be observed, indicating a high constant overhead when accessing data from the HDFS file system.

(4) The spectrum of input data sizes varies between stages and can be continuous (Figures 3.5c and 3.5h), discrete (Figures 3.5b and 3.5a) or very narrow (Figures 3.5d, 3.5e, 3.5f and 3.5g)

(5) Task runtime does not reliably depend on the selected parameters but can vary significantly (Figures 3.5b, 3.5a, 3.5f and 3.5g), even when accounting for node class, total input data size, executor state, stage and application. However, there are some cases that are close to ideal (Figures 3.5c, 3.5d and 3.5e). Here, task runtime is highly predictable, as can be seen by the small spread of task runtimes for each node class, category and input data size.

(6) The relative standard deviation w.r.t. the mean task runtime is shown in Figure 3.6 for all evaluated queries. While the average relative standard deviation is ≈26%, there are extreme outliers, as can be seen by the error bars. Outliers are stages with mostly (but not exclusively) short tasks (tens to lower hundreds of milliseconds). Here, system noise has a large relative impact. These plots, however, do not differentiate between local and remote data, which the ML model does (see Section 3.3).

The analysis has shown that task runtime, as expected, strongly depends on the CPU model and frequency. However, task runtimes can vary significantly even when the collected metrics are identical (categorical) or similar (numerical). In order to quantify the variance, recorded metrics for each application and stage were grouped according to the executor state and node class. Within each group, data was sorted according to the total input data size into at most ten bins of at least 0.5MiB in size. For each group and bin, the mean m and the standard deviation d as well as relative standard deviation to the

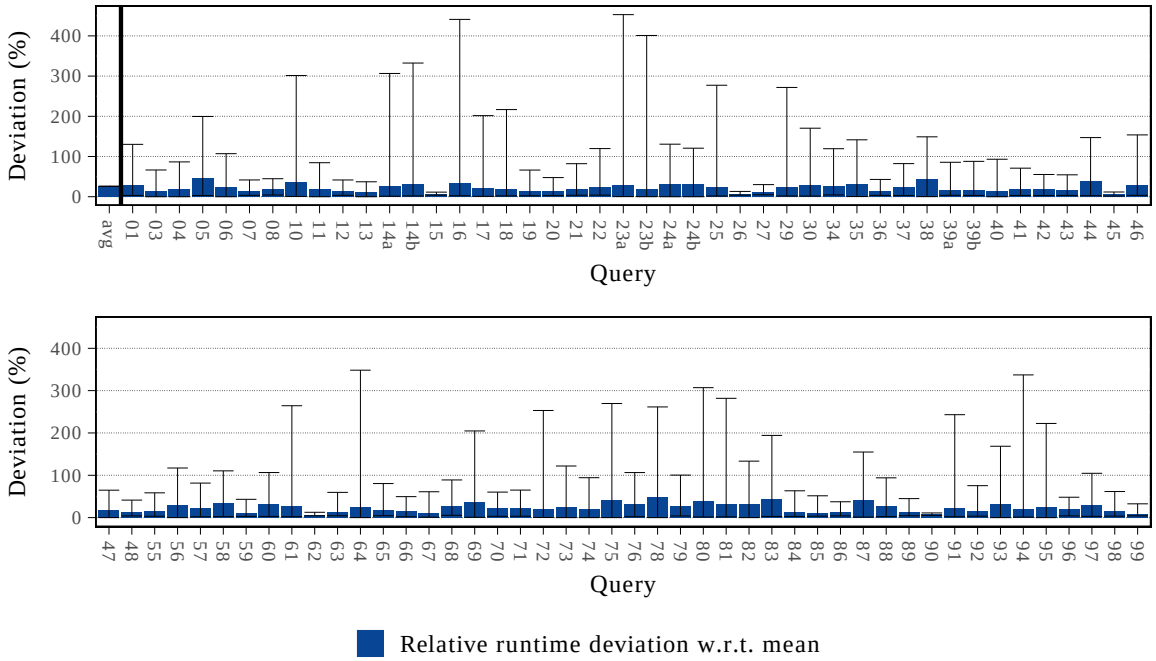


Figure 3.6: *Relative task runtime standard deviation w.r.t. mean runtime of each stage.*

mean m/d was computed. The latter is shown, summarized per query, in Figure 3.6. The relative standard deviation is ≈26% on average, but outliers as large as ≈450% occur. This poses a serious issue for accurate task runtime predictions, as any ML model can only be as good as the data it was trained on. The analysis has also resulted in valuable insights, namely that the executor state is an important factor for the task runtime, in parts even more so than the node class.

3.3.2 Model training

The task runtime analysis has shown that, given the same (or similar) values for measured metrics can result in large task runtime variations. As any model can always only predict a single output value for a set of input values, this variation will necessarily reduce the accuracy of the prediction, independently of the used prediction method. However, the task runtime analysis did not differentiate between local and remote input data, which is partially responsible for the variation. The amount of local and remote data is recorded separately, such that the ML model can differentiate between them. This can improve the accuracy of the prediction. In order to evaluate this, three ML training algorithms have been used to train models (from simple to complex) on the collected data. The recorded runtime metrics (except for the task runtime) are used as *features* (input data) for the ML training algorithms and the corresponding task runtime is used as *label* (output data).

(1) Linear Regression (LR) algorithm from the scikit-learn project [22]. A LR model is able to model linear relationships between features and labels. Using a LR implies

the assumption that the features and labels are linearly correlated. For instance, each input data item contributes a fixed amount of time to the task runtime. Similarly, the node class impacts task runtime by a linear factor.

(2) Classification and Regression Tree (CART) algorithm from the scikit-learn project. CART models can model arbitrary relationships between input and output data and are thus not limited to linear correlations. They are well suited for categorical features, such as the hardware resource class or executor state, and labels. For instance Figure 3.5a shows that input data sizes are not always linear but can be categorized or, as in Figure 3.5h, have no impact on task runtime.

(3) Gradient Boosted Decision Tree (GBDT) algorithm from Catboost [113]. GBDTs are based on an ensemble of CARTs and can improve accuracy. During inference, the prediction of each CART is added up to the final prediction. The potentially higher accuracy comes at the cost of increased training and inference time.

The use of ML algorithms requires the selection of features and preprocessing of the training data. The necessary steps depend on the selected algorithm. In all cases, however, separate ML models are trained per application and stage, as no correlation between stages or applications can be expected.

The following seven features have been selected:

- Executor state, i.e., first task of a stage on an executor and first task on an executor.
- Node class id
- Total input size
- Local node and executor input size
- Remote input size

The following preprocessing steps were performed.

- Training data outliers were filtered out, as training on outliers impairs the accuracy of the model. Outliers are training data records with the same or similar features but vastly different label values. In order to identify outliers, training data records for each stage were grouped according to the executor state and node class. Within each group, data was sorted according to the total input data size into at most ten bins of at least 0.5MiB in size. Each bin contains training data records with the same categorical and similar numerical feature values. For each group and bin, the mean m and the standard deviation d of all task runtimes were computed. Training data records with a task runtime $> m + s$ or $< m - s$ were assumed to be outliers and filtered out.

- Numerical values were scaled to a value between 0 and 1 by dividing each value by the maximal value. This step was only performed for the LR algorithm and ensures that features with large a value range have the same impact on the prediction as features with a smaller value range.
- Categorical features (node class id and executor state) were one-hot encoded. This step was only performed for the LR algorithm, as otherwise categories are interpreted as numerical values.

3.3.2.1 Model training and testing

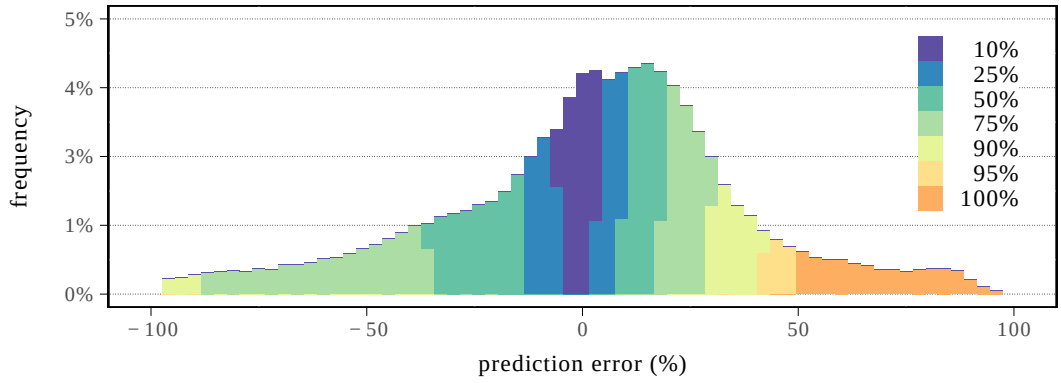
For the model training, the collected data was randomly shuffled and split into a training and a test data set using a 70/30 split. The random shuffle is necessary to ensure that both, training and test data, contain a representative set of records, e.g., records with all executor states.

For both tree algorithms, the maximal tree depth was varied between one and ten. Choosing the right tree depth is important, because a shallow tree may not be able to capture all relevant facets of the training data, while a deep tree may simply remember the training data (overfit). The tree depth which resulted into the model with the smallest mean square error (MSE) on the test set was selected. For all other training algorithm parameters, default values were used.

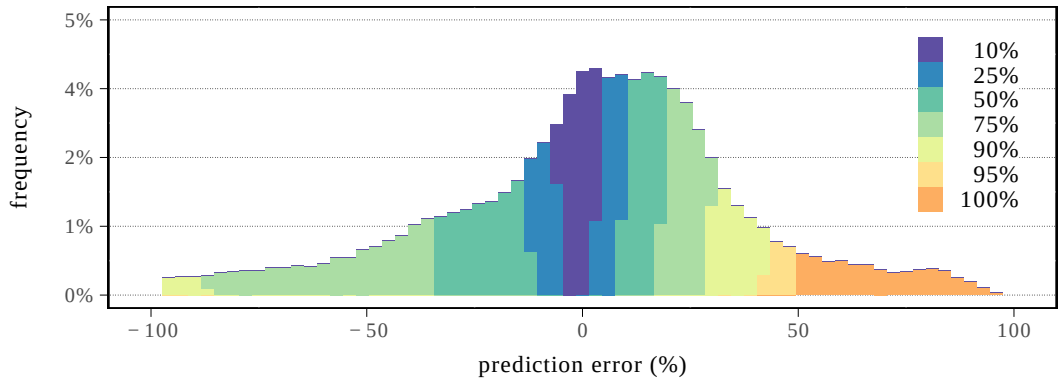
Figure 3.7 and Table 3.3 summarize the results of the task runtime prediction for each ML model.

	absolute mean	**signed mean**	**absolute median**	**signed median**
LR	37.7%	-9.8%	23.0%	3.0%
CART	37.9%	-10.2%	23.0%	3.0%
GBDT	42.8%	-14.8%	23.0%	2.0%

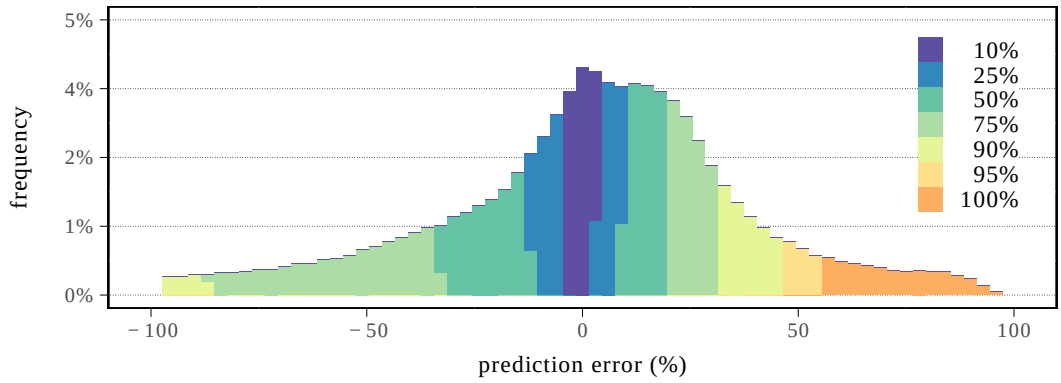
Table 3.3: *Mean and median task runtime prediction error. A negative (positive) value indicates underestimation (overestimation) of the task runtime.*



(a) LR



(b) CART



(c) GBDT

Figure 3.7: *Histogram of the overall task runtime prediction error (excerpt from -100% to +100%) for all three ML models. Colored regions correspond to the relative number of predictions within a given prediction error range.*

3.3.2.2 Prediction accuracy

As expected, given the task runtime variance in the training data, the task runtime prediction error is high. For lookahead schedule as used in HCL, the absolute, not the signed error is important, as errors in both directions impair accuracy of the planned schedule. The absolute mean error is 38% across all applications for LR and CART, and 43% for

GBDT. While absolute median error values are lower (23%) for all algorithms, this is not sufficient to compute schedules based on individual task runtimes. The evaluation of HCL (Section 3.2.5.1) has shown that the potential gain from lookahead scheduling is small, compared to that of simply choosing the fastest node class for each task. However, this small gain was achieved assuming a prediction error of 0%. Even a single task that runs longer than expected can stall an entire stage, and therefore an application. Hence, while the absolute median error is only 23%, 50% of predictions are less accurate, which will further reduce benefits of lookahead scheduling as used in HCL.

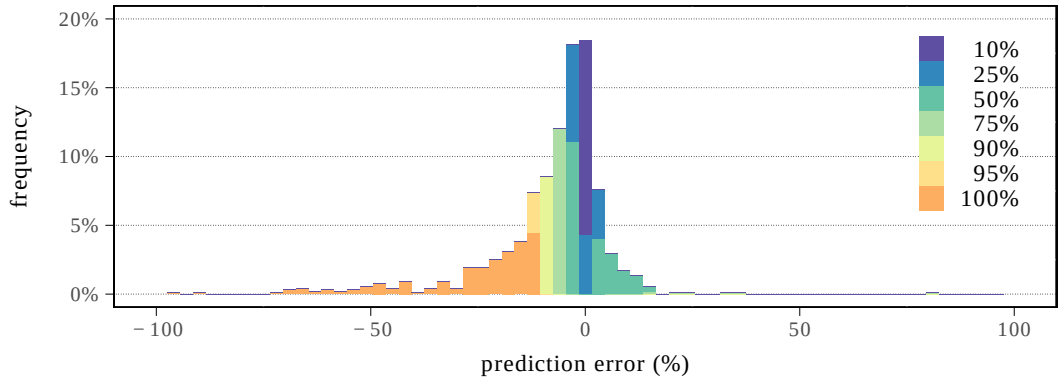
The achieved prediction error is also higher than in related work: Wang et al. [61] achieved a absolute mean task runtime prediction error for four Spark applications of 12.8%. However, the applications that are used by Wang et al. execute simple, iterative algorithms (page rank, k-means, logistic regression and word count), with strictly linear DAGs. The TPC-DS queries used here contain a large variety of heterogeneous, non-linear DAGs. While no systematic tests with simpler applications have been performed as part of this work, singular experiments with micro-benchmarks (e.g., word filter and count) also showed lower prediction errors.

Another source of the prediction error is that the collected metrics do not include interference-related information. While the number of executors per node (eight) that were used during the data collection runs is low compared to the number of CPUs (16 cores, 32 threads) and the available memory (160GiB – 256GiB) per node, interference cannot be completely ruled out. I/O interference seems unlikely, as the HDFS file system resided on RAM-disks and all nodes were connected by a 56Gbps Infiniband network.

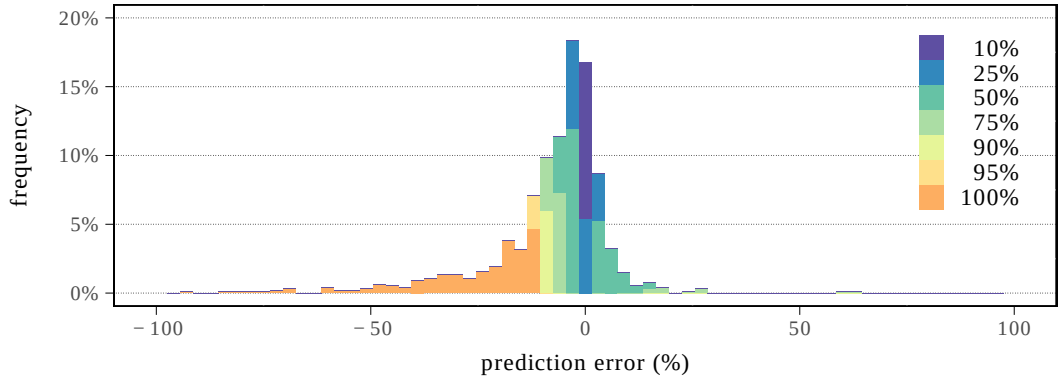
Per-stage prediction accuracy. Table 3.4 and Figure 3.8 show the task runtime prediction error, averaged over all tasks per stage. Here, the prediction error decreases significantly, to a signed median of -4.0% – -5.0%. While this is not sufficient for the approach chosen in HCL, it is for stage packing, which is presented in Section 3.4.

	absolute mean	**signed mean**	**absolute median**	**signed median**
LR	11.1%	-8.5%	6.0%	-4.0%
CART	11.0%	-8.8%	6.0%	-4.0%
GBDT	11.9%	-9.7%	6.0%	-5.0%

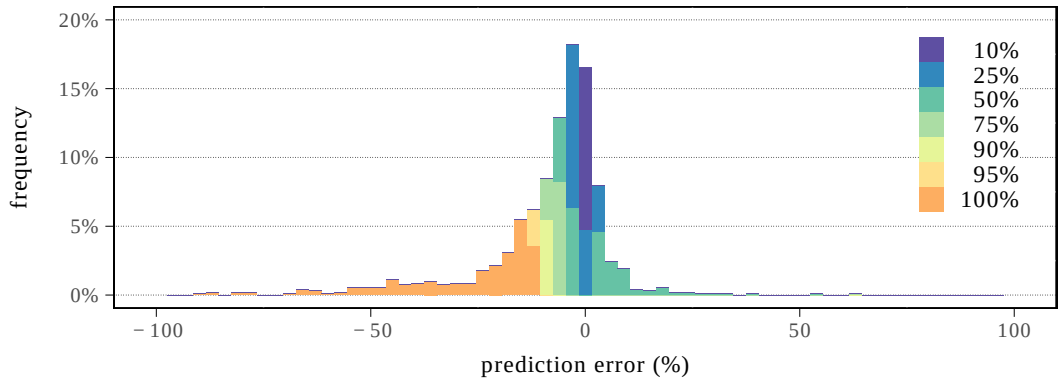
Table 3.4: *Mean and median task runtime prediction error* ***averaged per stage****. A negative (positive) value indicates underestimation (overestimation) of the task runtime.*



(a) LR



(b) CART



(c) GBDT

Figure 3.8: *Histogram of **per stage** task runtime prediction error (excerpt from -100% to +100%) for all three ML models. Colored regions correspond to the relative number of predictions within a given prediction error range. A negative (positive) value indicates underestimation (overestimation) of the task runtime.*

3.3.3 Summary

The results of the task runtime analysis and accuracy of the prediction model allows two courses of action:

(1) Improve the simulation used for HCL such that it accurately reflects the observed task runtime behavior and continue to improve the scheduling algorithm of HCL in simulation. In order for the simulator to accurately reflect the observed task runtime patterns, task runtime prediction needs to be improved. This, in turn, requires the identification and collection of additional metrics that correlate with the task runtime. Given the achieved results and the complexity of current distributed application frameworks, it is unclear, whether this can succeed. Furthermore, the goal of this work is not to build a simulator, but to devise scheduling techniques that work in practice, where the simulator is just a means to an end. Hence, this course of action does not support this goal.

(2) Proceed with real-world experiments. Devise scheduling techniques that reduce application runtime and increase resource utilization efficiency in practice, using the insight gained from the task runtime analysis and the achieved accuracy of the prediction model.

The latter course of action does support the goal of this work and has therefore been chosen and described in the next section.

3.4 HCL-SP (stage packing)

HCL-SP is a DAG-, task, hardware- and state-heterogeneity-aware application scheduler, based on insights gained from the task runtime analysis and the achieved task runtime prediction accuracy (Section 3.3). The main factors that influenced the scheduling algorithm of HCL-SP are:

(1) The impact of the executor state on task runtime is significant (up to ≈15.4× on average) and can exceed that of the node class (2.3× on average).

(2) Given the high computational complexity and the inability to compute accurate runtime predictions for individual tasks, an approach similar to that chosen in HCL would unlikely be beneficial in non-simulated environments.

(3) Averaged per stage, task runtime predictions are more accurate than individually.

For these reasons, a novel scheduling technique, **stage packing**, was devised that exploits the impact of the executor state on task runtimes by packing tasks of the same stage on as few executors as possible without reducing overall parallelism. This increases the chance of using hot executors. Task runtime predictions, averaged per stage and **relative** to that of other stages are used to determine the number of executors to assign to each stage and from which node class to select them. Furthermore, by packing tasks of stages on fewer executors, more tasks run back-to-back, which increases the chance that stragglers can be hidden and reduces their impact on the stage finish time.

3.4.1 Assumptions

The following assumptions are made:

- Executor sharing across applications incurs significant overhead compared to the application runtime and should to be avoided. This assumption justifies the approach to improve schedules of individual applications instead of across applications.
- Task runtimes on hot executors are lower than those on warm and cold executors. This assumption needs to hold in order for stage packing to be beneficial.
- Task runtimes depend on the node class of the executor they're running on. Tasks have a preference order of node classes. This order is the same for all tasks of a stage. The task runtime prediction per node class accurately reflects this order, otherwise less preferred node classes are selected even if preferred node classes are available.
- If the average actual task runtime of stage s_1 is t_1 and that of stage s_2 is $t_2 = a \times t_1$, then the average predicted task runtime t'_1 and t'_2 have the same ratio, i.e., $t'_2 = a' \times t'_1$ with $a = a'$. The larger the difference between a and a', the less accurate is the path weight.
- Applications are recurrent such that the task runtime prediction model can be trained on multiple runs of the same application.

3.4.2 Application scheduler

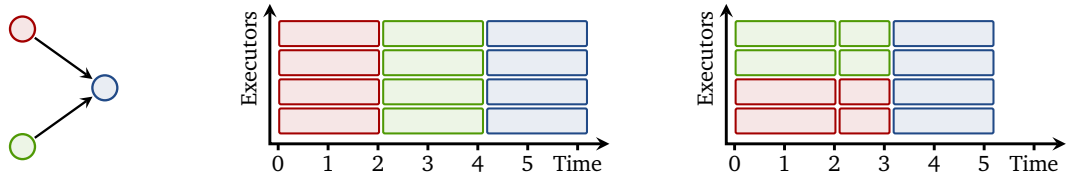
This section describes the stage packing algorithm of HCL-SP. The main idea of stage packing is to pack tasks of a stage on as few executors as possible, without reducing overall parallelism, and therefore to increase the usage of hot executors. Stage packing is not specific to heterogeneous clusters. To the best of my knowledge, no prior work explicitly considers the state of executors when computing DAG schedules.[6]

3.4.2.1 Stage packing

Figure 3.9b shows a schedule for the DAG in Figure 3.9a as it would be computed by Spark's native scheduler. Stages, even those that could run in parallel, are executed back-to-back. This strategy is oblivious to the impact of the executor state on task runtimes. Hot executors are only used if the number of tasks of a single stage exceeds the number of executors. On the other hand, a stage packing schedule (Figure 3.9c) schedules stages in parallel, which increases the usage of hot executors and decreases task runtimes.

[6] Serverless frameworks [69, 91] reuse warm containers to accelerate the execution of event-driven, (mostly) single-task applications, which are not comparable to distributed applications with complex DAGs.

Furthermore, stage packing can help to reduce the impact of stragglers on the finish time of stages. The impact of a straggler increases, the later in the execution of a stage it occurs (see Figure 2.6). In the schedule shown in Figure 3.9b, each task is a last task of a stage on an executor, hence if any task becomes a straggler, the stage finish time is delayed. In the stage packing schedule (Figure 3.9c), on the other hand, only half the tasks of the green and red stages are last tasks. This reduces the number of last tasks and hence the likelihood of a last task to become a straggler.



(a) Example DAG **(b)** Vanilla Spark **(c)** Stage packing

Figure 3.9: *Example of the stage packing scheduling strategy using the example DAG (a) with three stages (four tasks each) and vanilla Spark's scheduling strategy (b) compared to a stage packing scheduling strategy (c). Stages and tasks are color coded. Boxes in (b) and (c) represent tasks of stages with the width representing the task runtime.*

3.4.2.2 Sizing of executor pools

Stage packing requires the partitioning of executors into one or more executor pools. Each pool is assigned exclusively to one stage. As stages on parallel paths of the DAG are executed at the same time, the problem of how to size the executor pool for each stage arises.

Ideally, the executor size corresponds to the amount of work that is performed in each stage, such that stages on converging paths reach the convergence point at the same time. To determine what this size is, the amount work that needs to be performed to reach any convergence point in the DAG is computed. At each convergence point, the input paths are assigned a **path weight** in the range $(0, 1]$, such that all weights add up to one. The total number of executors is split between all input paths in correspondence to these weights. This is done recursively, until an input stage has been reached. The path weight at an input stage represents the fraction of all executors that will be assigned to a stage.

The path weight is computed in two phases.

(1) In the forward phase, the absolute weight of each path from an input stage to the final stage is computed. The absolute path weight $w_P = \sum_{S \in P} w_S$ is the sum of all absolute stage weights w_S of all stages $S \in P$ on path P. w_S is the sum of the

average predicted task runtime τ_{avg} of all tasks $t \in S$ on all node classes $n \in N$, with N being the set of node classes. It is computed as follows:

$$\tau_{avg} = \frac{1}{|S|} \sum_{t \in S} \frac{1}{|N|} \sum_{n \in N} \frac{1}{3} \left(\tau^{c}_{t,n} + \tau^{w}_{t,n} + \tau^{h}_{t,n} \right) \tag{3.2}$$

$$w_S = |S| \times \tau_{avg} \tag{3.3}$$

$\tau^{c}_{t,n}/\tau^{w}_{t,n}/\tau^{h}_{t,n}$ is the predicted runtime of task $t \in S$ on a cold/warm/hot executors on node class $n \in N$. The simplifying assumption that all executor states occur with the same likelihood is made here. In reality, this likelihood varies. However, at the time of computing the path weight, the state of executors on which tasks are executed are still unknown.

(2) In the backward phase, the relative weight $(0, 1]$ of each input paths of a convergence point is computed. The relative path weight at an input stage corresponds to its executor share.

As the path weight is computed before executors are selected, before tasks are scheduled and relies on average task runtime predictions, it is only an approximation of the actual path weight. Therefore, the path weight is recomputed periodically as tasks finish. If necessary, executor pool sizes are adjusted accordingly.

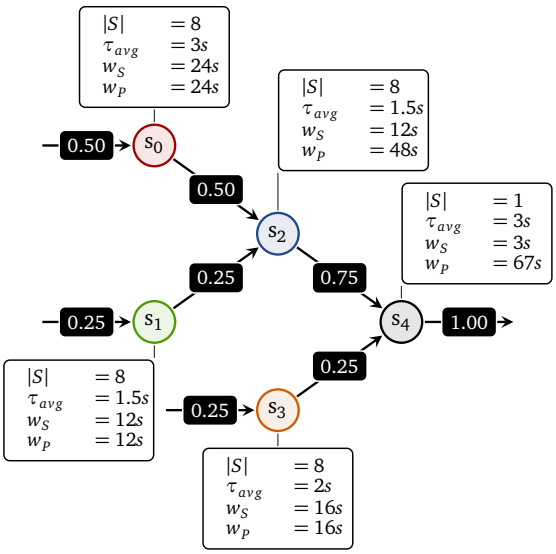
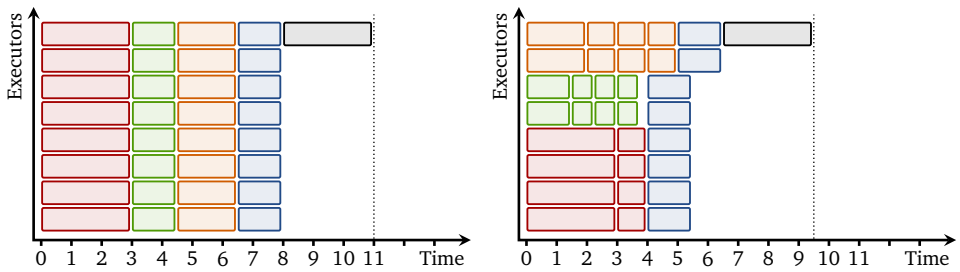


Figure 3.10: *Example of the path weights 5-stage DAG. Labels next to nodes (stages) show absolute values, labels on edges represent the relative path weight.*

An example of path weights for a DAG with 5 stages is shown in Figure 3.10. Figure 3.11b shows the corresponding stage packing schedule on eight homogeneous executors in comparison to Figure 3.11a, which shows a vanilla Spark schedule for the same DAG. In this example, all executors are assumed to be warm in the beginning.

Tasks of stage s_0 run three times faster on hot executors than on warm executors, tasks of other stages run twice as fast. Each executable stage is assigned the share of executors that corresponds to its relative path weight (rounded to the nearest integer). The executor pool size of stages is computed in descending order of their path weights. In this example, the order is s_0, followed by s_1 and s_3 (stages s_2 and s_4 are only considered once they are executable).

- Stage s_0 is assigned $\lfloor 8 \times 0.5/(0.5+0.25+0.25) \rceil = 4$ executors. 4 executors and stages s_1, s_2 remain.
- Stage s_1 is assigned $\lfloor 5 \times 0.25/(0.25+0.25) \rceil = 2$ executors. 2 executors and stage s_0 remain.
- Stage s_2 is assigned $\lfloor 2 \times 0.25/(0.25) \rceil = 2$ executors. No executors and no stages remain.



(a) Vanilla Spark **(b)** Stage packing

Figure 3.11: *Exemplary comparison of a vanilla Spark schedule (a) and a stage packing schedule (b) for the DAG shown in Figure 3.10 on 8 executors on a homogeneous cluster (to simplify the example). Application finish times are indicated by the vertical dotted line. Boxes represent tasks of stages. The width represents the task runtime. Stages are color coded.*

3.4.2.3 Executor selection

Executors are selected by the scheduler in a greedy fashion for each stage. All executable stages are served in a round robin fashion. During each round, each stage can pick one executor until it has reached its allowed share. Executors are selected according to two criteria once a stage becomes executable or when the relative path weight, and therefore the executor share of a stage, changes:

(1) Select an executor that has already executed a task of the same function before, i.e., is hot.

(2) If no such executor is available, select an executor from the most preferred node class. If no such executor is available either, proceed to the next-most preferred node class, until a free executor has been found.

The node class preference order is computed by ranking node classes in ascending order of the average predicted task runtime on each node class $n \in N$. The highest ranked node class is assumed to execute tasks of a stage the fastest.

Executors are released back into the global pool from which other stages of the same application can pick once:

- The allowed size of the executor pool of a stage is reduced, because the relative path weight changed.
- The size of the executor pool is larger than the number of unfinished tasks of a stage.

If executors have to be released, the next executor that finishes a task is released.

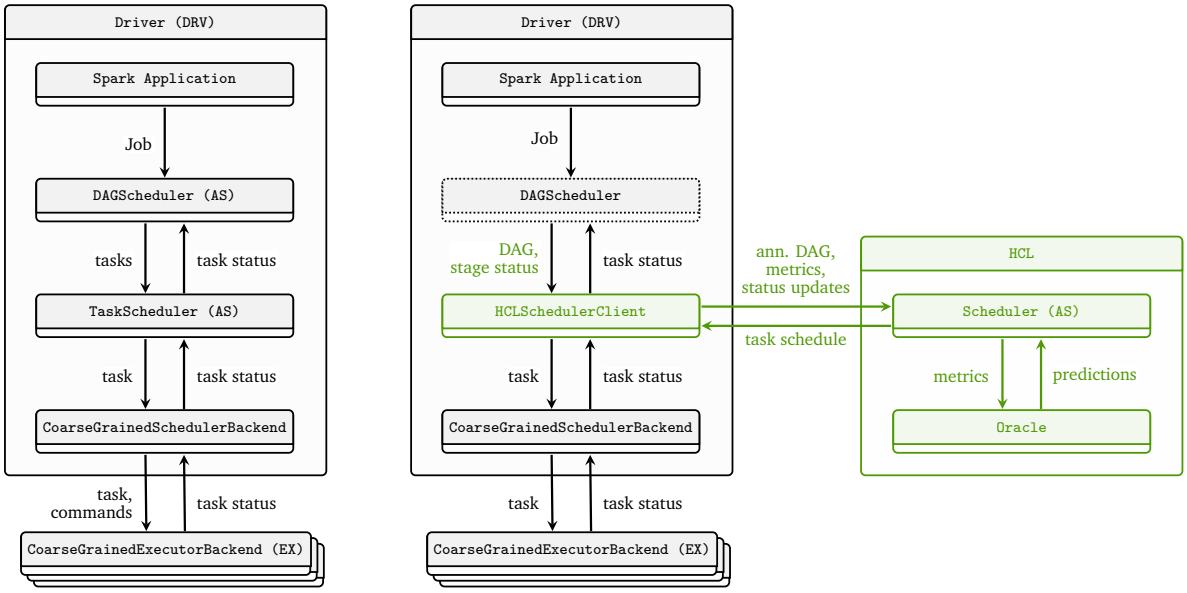
3.4.2.4 Task scheduling

Within a stage's executor pool, the task-heterogeneity-aware scheduler places tasks. HCL-SP uses a simple task placement approach where tasks are assigned to free executors in descending order of their input data size (which is provided by Spark). Similar to how stages pick executors, tasks are placed on the most preferred executors, if they are available. This strategy ensures that the longest running tasks of a stage can run on the most preferred executors. Furthermore, by scheduling shorter running tasks later, the scheduler can fill gaps in executor utilization. This assumes a positive correlation of input data size to task runtime. No further optimizations have been implemented.

3.4.3 Oracle

As other heterogeneity-aware schedulers, HCL-SP uses data from previous runs of an application to compute task runtime predictions [33, 44, 67, 66, 45, 75, 14, 3]. This is possible as many (≈40% [24]) workloads are recurring. HCL's oracle uses Catboost's GBDT implementation [113] to compute task runtime predictions. Apart from recording and storing performance metrics (see Table 3.1 for a list of recorded metrics), the oracle is simply a wrapper for Catboost. Training of the corresponding models is not part of the oracle itself but implemented in a separate Python script. This training script uses the recorded performance metrics to train a model offline.

Catboost was chosen due to practical reasons, even though it's accuracy is not the highest among the tested algorithms (see Section 3.3.2). It has a native C++ as well as a Python API and allows to exchange trained models between them. Other ML libraries with a C++ API, such as Mlpack [101] and Shogun [15] have been evaluated as well. They have not been chosen due to bugs, e.g., when storing and loading models from files, or due to the lack of a Python API, which is the preferred method to train models offline.



(a) Vanilla Spark **(b)** Spark+HCL-SP

Figure 3.12: *Depiction of HCL-SP's Spark integration showing the necessary modifications of Spark DRV the interplay with the AS. New components are highlighted in green and dotted components were significantly modified.*

3.4.4 Spark integration

HCL-SP has been integrated into Spark and its application driver (DRV) component, where Spark's native scheduling functionality resides, had to be adapted. HCL-SP takes over DAG and task scheduling functionality. In order to do that, Spark communicates the application DAG with annotations, as well as scheduling-related events (such as *stage ready* or *task finished*) to HCL-SP. In addition, task execution metrics, which are collected within Spark's execution environment (EX) and DRV components and are used by the oracle to refine and compute task runtime predictions, are communicated to HCL-SP.

Figure 3.12 shows the integration of HCL-SP into Spark (Figure 3.12b) and contrasts it to vanilla Spark (Figure 3.12a). The integration of HCL-SP is limited to Spark's DRV component which executes user-written application code as well as Spark subsystems, that are responsible for scheduling, task execution and data distribution coordination. HCL-SP's AS replaces Spark's native DAG and task scheduling functionality. Merely the DAG construction and the relaying of HCL-SP's task schedules to the `CoarseGrainedSchedulerBackend`, which is responsible for initiating task execution on long running `CoarseGrainedExecutorBackend` (executor) processes, remains.

As Spark's DAGs are isomorphic but not equivalent across application runs, a stable ID was added to each stage, in the course of the HCL-SP integration, as to make DAGs equivalent across application runs. This was necessary as HCL-SP needs to be able to reliably identify stages across application runs to select the corresponding prediction model (see Section 3.3.1.1 for details about the required modifications).

In HCL-SP, the `DAGScheduler` constructs a job DAG and forwards it to the newly added `HCLSchedulerClient` upon job submission. This new component annotates the DAG with input data locations and sizes for each task, as well as the stable stage ID and relays it to HCL-SP via HCL-SP's REST API (see Section A.1.1 for details). HCL-SP uses this information to compute a schedule for each task, which it returns to Spark via the same API. Upon completion of a task, the `HCLSchedulerClient` relays metrics (see Section 3.3.1 for a description of collected metrics) back to HCL-SP, which it feeds into its oracle. The oracle stores the data in a CSV file, which is read by the external ML training script to update the corresponding ML model.

3.4.5 Evaluation

In order to evaluate the potential benefit of stage packing as well as task runtime predictions, 90 TPC-DS queries were executed on Spark+HCL-SP and compared against vanilla Spark. All experiments use the test setup described in Section A.1.2.

This evaluation answers three questions:

(1) What is the benefit of a state- and DAG- and task-heterogeneity-aware scheduling strategy (Spark+HCL-SP) compared to an oblivious one (Vanilla Spark)?

(2) What is the benefit of adding hardware-heterogeneity-awareness to stage packing (Spark+HCL-SP (RP)) compared stage packing alone (Spark+HCL-SP) and an all-oblivious strategy (Vanilla Spark)?

(3) Can a benefit as in (2) be achieved without task runtime predictions, but with simply using a fixed node class preference order?

Each experiment was repeated ten times. For task runtime predictions, the ML model was trained on runtime metrics collected over ten prior runs without task runtime predictions.

3.4.5.1 Test setup

All experiments use the test setup described in Section A.1.2.

Test applications

For these experiments a Spark implementation of the TPC-DS benchmark suite is used. A description of this benchmark is in Section A.1.4.

3.4.5.2 Configurations

Table 3.5 lists the configurations that are compared in this evaluation.

Configuration	Description
Vanilla Spark	Vanilla Spark without HCL-SP.
Spark+HCL-SP	Spark with HCL-SP as application and task scheduler. HCL-SP assumes the same runtime for all tasks. If available, hot executors are selected. Otherwise executors are selected randomly.
Spark+HCL-SP (RP)	Spark with HCL-SP as application and task scheduler. HCL-SP uses the ML model to predict task runtimes. If available, hot executors are selected. Otherwise executors are selected from node classes, in preference order predicted by the ML model.
Spark+HCL-SP (FP)	Spark with HCL-SP as application and task scheduler. HCL-SP assumes the same runtime for all tasks. If available, hot executors are selected. Otherwise executors are selected from node classes, in a fixed (configurable) preference order. The fixed node class preference order is <3,2,1,4> (see Section A.3.3.1 for an explanation of node classes), which corresponds to the observed speed of node classes from fast to slow.

Table 3.5: *Configurations compared in this evaluation.*

3.4.5.3 Results

Figure 3.13 shows results for each query and Figure 3.14a the corresponding CDF. Results discussed in the text as well as figures do not consider the first 24 stages (0 – 23) of each query, unless noted otherwise. While part of the application, these stages are not part of the time the TPC-DS benchmark itself measures [116]. For reference, a CDF with results for the entire application is provided in Figure 3.14a. HCL-SP is able to shorten the runtime of these stages by ≈6× compared to vanilla Spark.

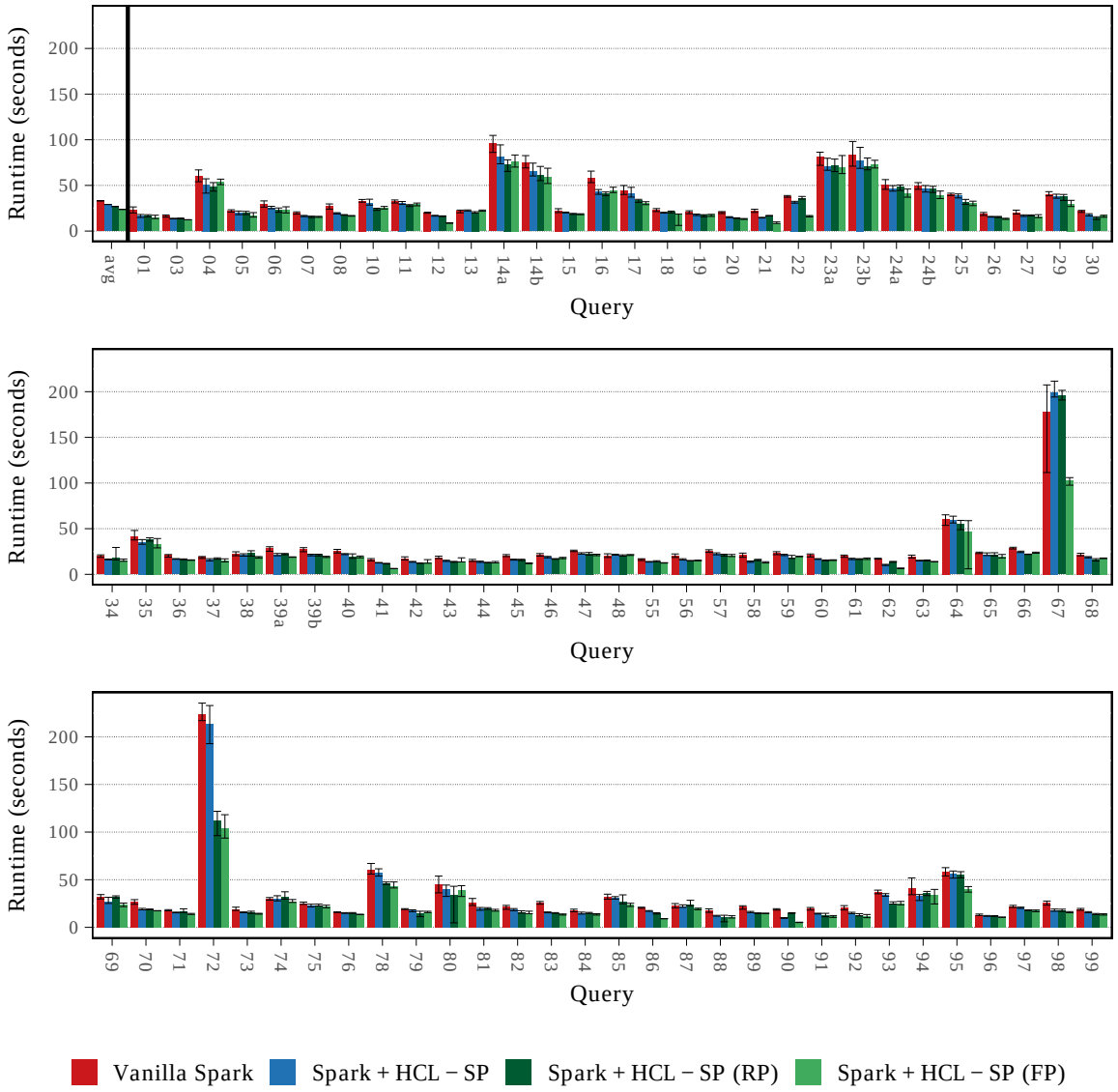
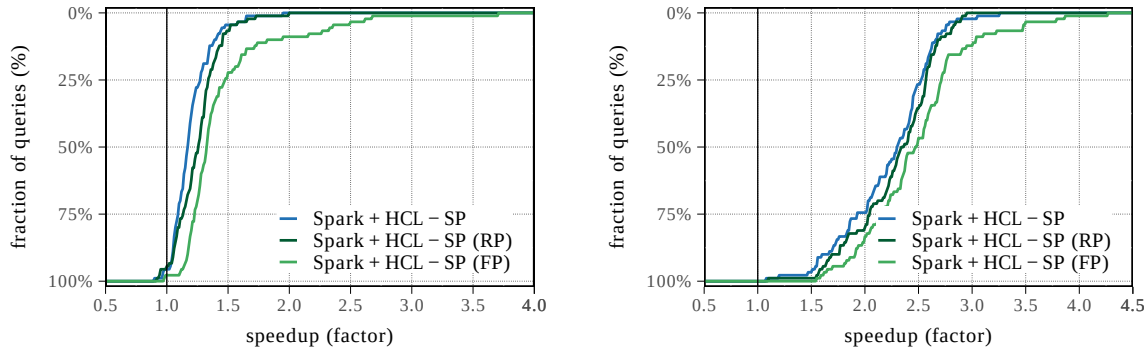


Figure 3.13: *Comparison between stage packing schedules and the baseline (Vanilla Spark).*



(a) without stages 1-23

(b) entire application

Figure 3.14: *Speedup of Spark+HCL-SP compared to vanilla Spark.*

Overall, Spark+HCL achieves runtime parity or speedup compared to vanilla Spark for 94.44% to 97.78% of all queries and a maximal speedup of 1.95× to 3.71×, depending on the used HCL configuration. Results are summarized in Table 3.6 and discussed for each HCL configuration separately in the following:

	Vanilla Spark	Spark+HCL-SP	Spark+HCL-SP (RP)	Spark+HCL-SP (FP)
Fraction of sped up queries	0.00%	95.56%	94.44%	97.78%
Mean execution speedup	1.00×	1.14×	1.23×	1.39×
Median execution speedup	1.00×	1.17×	1.26×	1.33×
Minimal execution speedup	1.00×	0.90×	0.92×	0.98×
Maximal execution speedup	1.00×	1.95×	2.00×	3.71×
Absolute resource executor busy time increase	1.00×	1.05×	1.09×	1.04×
Mean resource utilization	12.19%	27.79%	30.50%	30.64%
Mean resource utilization efficiency increase	1.00×	1.14×	1.23×	1.39×

(a) excluding stages 0 – 23

	Vanilla Spark	Spark+HCL-SP	Spark+HCL-SP (RP)	Spark+HCL-SP (FP)
Fraction of sped up queries	0.00%	100.00%	100.00%	100.00%
Mean execution speedup	1.00×	1.99×	2.11×	2.30×
Median execution speedup	1.00×	2.30×	2.34×	2.47×
Minimal execution speedup	1.00×	1.08×	1.10×	1.55×
Maximal execution speedup	1.00×	3.26×	2.95×	4.27×
Absolute resource executor busy time increase	1.00×	1.02×	1.06×	1.01×
Mean resource utilization	9.56%	23.8%	26.0%	25.9%
Mean resource utilization efficiency increase	1.00×	1.99×	2.11×	2.30×

(b) entire application

Table 3.6: *Summary of performance and efficiency metrics. Relative values are in comparison to vanilla Spark.*

3.4.5.4 Results discussion

Spark+HCL-SP. Stage packing alone achieves a mean (median) reduction of application runtime of 1.14× (1.17×). This result is within the expected range, given that the positive effect of stage packing is limited to situations where two or more stages with

more combined tasks than executors can run in parallel. Mean resource utilization more than doubles, from 12.19% to 27.79%, which indicates that stalls (due to stragglers) and executor idle times are reduced significantly. At the same time, absolute executor busy time, i.e., the accumulated time each executor was executing tasks, increased slightly, by 1.05×. The latter result was not expected, as with an increased usage of hot executors, task execution should take less time overall. A possible reason for this increase is that due to the reduced stalls and executor idle times, the fraction of executors that are busy at any point in time increases. This can also increase interference across executors, particularly on the same node, which leads to an increased task execution time. Nevertheless, mean resource utilization efficiency increases by the same factor as the mean execution speedup. This is because executors cannot be shared across applications, thus the less time is required to perform the same work (execute the application), the higher the utilization efficiency is.

Spark+HCL-SP (RP). When adding task runtime predictions to compute path weights and pick executors from preferred node classes, a mean (median) speedup of 1.23× (1.26×) over vanilla Spark was achieved. The increased speedup compared Spark+HCL-SP was expected, as with runtime prediction, the path weight becomes more accurate. Additionally, executors are less likely to be scattered across node classes as they are chosen in preference order and not at random. Picking executors from the same (or fewer) node classes is important, as it reduces the risk of any single task to become a straggler due to the usage of slower nodes.

Similarly to Spark+HCL-SP, mean resource utilization, as well as the absolute executor busy time, increases to 30.50% and by 1.09× respectively. The reasons for this are the same. Additionally, with runtime prediction, tasks of the same stage are packed onto fewer nodes than with stage packing alone, which increases interference further. This is because the scheduler preferably picks executors from one node class at a time, instead of randomly from all node classes. Yet, mean resource utilization efficiency increases by 1.23× compared to vanilla Spark, due to the further reduced application runtimes.

Spark+HCL-SP (FP). When using a fixed node class preference order, the mean (median) speedup compared to vanilla Spark is 1.39× (1.33×) and therefore exceeds that of both other HCL configurations. This result was not expected, as the assumption was that the path weight is more accurate with runtime predictions, given that averaged over a stage, the median prediction error is ≈6% (Table 3.4). A randomly selected sample of runs were manually compared to Spark (SP+RP). This comparison revealed a different node class selection pattern. Spark (SP+RP) chose slow node classes over fast node classes in some cases. This indicates that the used ML models does not reliably predict the actual node class preference order for all stages.

Mean resource utilization and absolute executor busy time increase as well to 30.64% and by 1.04× respectively. The reasons are the same as with Spark+HCL-SP (RP), except

that due to the better node class selection, absolute executor busy time decreases slightly. Due to the reduced application runtime, resource utilization efficiency increases by 1.39× compared to vanilla Spark.

The benefits of Spark (SP+FP) were achieved without the need for model training and can therefore also be used for non-recurrent applications from the very first run on. According to these results, task runtime prediction using ML models does not justify the effort necessary to do so, as the simpler method used here leads to better results. Furthermore, while the preference order has been configured manually in this evaluation, simple micro-benchmarks may be used in real-world scenarios to determine the speed of each node class.

Outliers. The data contains two outliers, queries 67 and 72. Both queries contain single tasks that run for ≈70s on the three fast node classes but take up to ≈190s on the slow node class. Vanilla Spark and Spark+HCL-SP do not know this. For query 72, Spark+HCL-SP (RP) as well as Spark+HCL-SP (FP) avoid using slow resources. For query 67, however, the predicted node class preference order is wrong and the slowest node class is selected despite the fact that executors on the other node classes were still available. However, in prior trial runs, this did not happen and Spark+HCL-SP (RP) was correctly avoiding the slow node class. All tests were performed on the same test setup and with the same code base but with different training runs. This indicates that the accuracy of predictions is unstable.

3.4.5.5 Closer look

This section examines schedules for vanilla Spark and Spark+HCL-SP (FP), as fastest HCL configuration, at the example of TPC-DS query 23a. This query has been chosen for closer examination, as it is complex enough to show the impact of stage packing and node class selection but not too complex to understand and not too large to reasonably visualize schedules.

Figures 3.16a and 3.16b show swimlane diagrams of the schedules of query 23a on vanilla Spark and on Spark+HCL-SP (FP). Larger versions of both figures and figures for both other HCL-SP configuations are provided in the appendix (Section A.1.5). Time is given on the x-axis and executors on the y-axis. Tasks are depicted as colored boxes with stage and task id labels[7]. The width represents the runtime of a task. Some tasks are very short and some stages only have a small number of tasks, such that not every stage and task can be clearly identified in the figures. Tasks of the same stage are colored identically. However, colors repeat after eight stages. Both figures have been annotated with application phases 1 – 5 (top), as well as node classes (n/c) and the preference

[7]Labels can only be identified in the PDF version of this document. Furthermore, labels are only shown if they fit inside the box.

order (p/o) used for Spark+HCL-SP (FP) (right side and with gray shaded swimlanes). A mapping of node classes to hardware descriptions is provided in Table A.13.

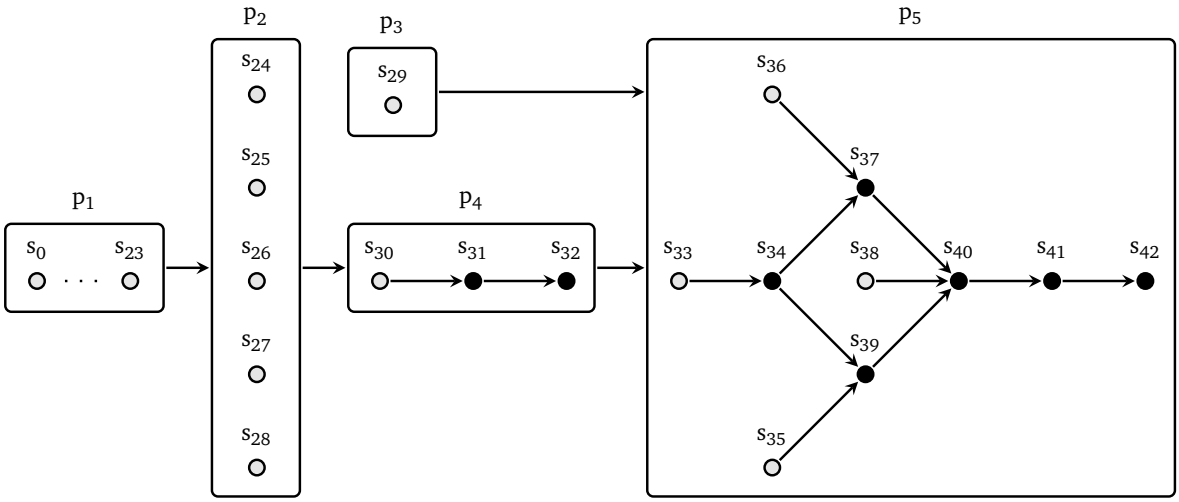
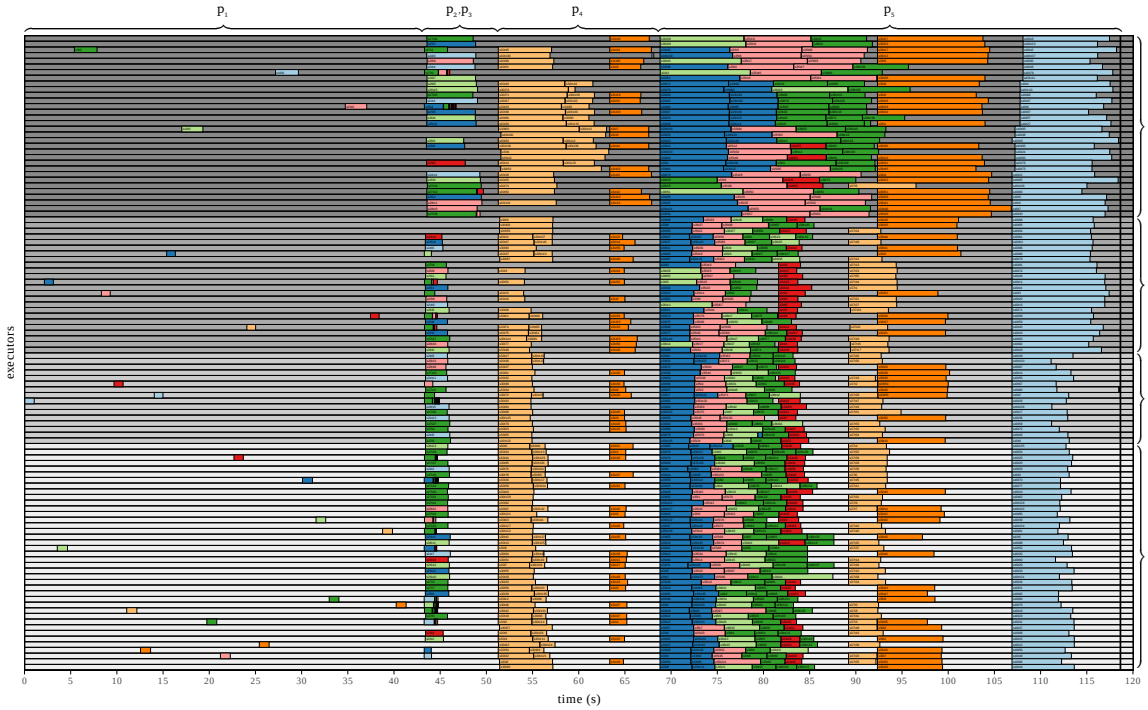
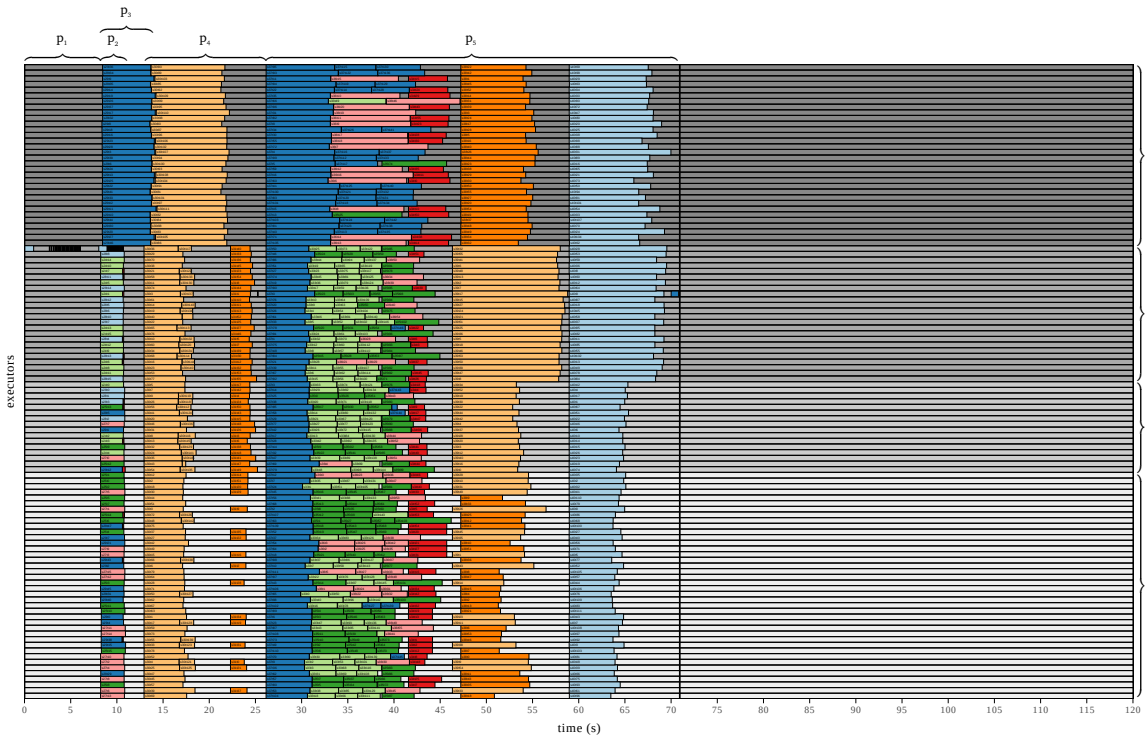


Figure 3.15: *Depiction of the DAG of TPC-DS query 23a. The application consists of five phases ($p_1 - p_5$), each consisting of multiple jobs and stages. Control dependencies between phases and data dependencies between stages are depicted by arrows. Stages that load data are depicted as empty circles, stages that process data are depicted as full circles. In total, 1126 tasks are executed across 43 stages.*

Figure 3.15 shows the corresponding DAG of query 23a. The application is divided into five phases with no data dependencies in-between. Each phase consists of one or more stages.



(a) Vanilla Spark



(b) Spark+HCL-SP (FP)

Figure 3.16: *Schedule of TPC-DS query 23a, executed on the test cluster (Section A.3.3.1) with phases (top) as well as node classes (n/c) and preference order (p/o) (right). The bold vertical bar represents the end of the application. Larger versions of both figures can be found in Section A.1.5.*

In the following, the scheduling strategies and their impact on execution are discussed for each phase separately.

(1) **Phase 1.** The application starts by executing 24 stages with a single task each. Spark schedules tasks randomly across all available executors, thus using a cold executor for each task. HCL-SP, on the other hand, schedules all tasks of all stages on the same executor, as they have the same stable ID (see Section 3.3.1.1 for details), thus maximizing the usage of hot executors. As a result, Spark's native scheduler executes the first 24 stages in ≈44s, whereas HCL-SP's executes them in ≈8s. This shows the potential benefits of stage packing in apparently trivial scheduling situations, and an issue with assuming that task runtime is independent of the executor state.

(2) **Phase 2 and 3.** Spark schedules tasks of the second phase across all node classes, whereas HCL-SP only uses the three most preferred ones, which reduces the runtime of that phase. As phase 3 becomes ready slightly later, the most preferred node classes have already been split up among stages of phase 2 and thus it has to use executors on node class 4. Vanilla Spark scatters tasks of phase 4 across all node classes with the consequence that both phases take equally long, whereas phase 2 finishes earlier with HCL-SP.

(3) **Phase 4** depends on phase 2 but not on phase 3. As phase 2 finishes earlier in the HCL-SP schedule than in the vanilla Spark schedule, it can start earlier in the former. The first stage of phase 3, stage 30 (green), has 148 tasks and as no other stage can run at the same time, it is spread across all executors in both schedules. However, neither schedule is "optimal" here: Omitting executors from node class 4 would have reduced the stage runtime. This is because it would have used more hot executors. The impact of hot executors on tasks of stage 30 is visible in the HCL-SP plot, as each 2nd task of stage 30 on an executor only requires about 1/3rd of the runtime of the 1st task. HCL-SP is not able to decide not to use an available executor.

In the vanilla Spark schedule, tasks are scheduled on the slow node class 4, despite the fact that executors on other, fast, node classes are available as well. This is due to Spark's delay scheduling strategy [18]. Per default, Spark waits up to three seconds for an executor with the most amount of input data to become available, before it schedules it on any free executor. HCL-SP does not use this strategy, as it has been shown experimentally [74, 58], that data transfer times on fast networks (e.g., 40Gbps) are neglectable compared to time spent in (de-)serialization, local data copies and object constructions. The latter steps are still necessary when data is read from HDFS, independently on whether the data originated from the same node or not. Furthermore, when exchanging intermediate results data, experiments with TPC-DS have shown that input data for the next stage is typically

scattered across all executors of the previous stage, hence only a small fraction of input data is local on any given executor anyway.

Hardware heterogeneity-awareness is beneficial for the second stage of phase 4 (blue), hence it finishes in less time in the HCL schedule. The last stage of phase 4 has only a single, sub-second task, hence it does not noticeably impact the end time of this phase.

(4) **Phase 5.** In phase 5, several stages are executed in parallel. The fraction of tasks that run on hot executors increases from 18.75% to 30.77% for Spark+HCL-SP (FP), compared to vanilla Spark. The maximal possible fraction of tasks on hot executors can only be estimated, as stages become ready at different points in time, even without unmet data or control dependencies. However, assuming that all stages that can run in parallel become ready at the same point in time, a maximum fraction can be computed. According to the DAG in Figure 3.15, two sets of stages can run in parallel:

(1) Stages s_{33}, s_{35}, s_{36}, s_{38}. These stages have 456 tasks in total.[8]

(2) Stages s_{34}, s_{37}, s_{39}. These stages have 168 tasks in total.

Independently of how many tasks belong to which stage, the first 112 tasks of each set need to *heat up* executors, while the remaining 344 and 56 tasks can run on hot executors. This results in a total fraction of tasks on hot executors of $(344+56)/(456+168) = 64.10\%$, which is significantly higher than the fraction that HCL-SP achieves. However, this estimate is idealized and overestimates the actual possible fraction. Referring to Figure 3.16b:

- The first stage of set one (dark blue) with 148 tasks becomes ready before all others. As HCL-SP does not know when other stages will become ready, it does not consider them when assigning executors, hence this stage gets all 112 executors. This reduces the estimate to $(344-112+56)/(456+168) = 46.15\%$.
- The first stage (dark red) of set two with 56 tasks also becomes ready before all others, leaving 112 tasks for the other two stages (pale and dark orange). This reduces the maximal fraction for the second set to zero and reduces the estimate to $(344-112+0)/(456+168) = 37.18\%$.

The latter number is close to the fraction that HCL-SP achieves. The remainder is due to inaccurate path weights, which, when updated, adds or removes executors from stages. This means that some stages were spread across too many executors in the beginning while other stages spread out towards the end, causing some tasks in either case to unnecessarily run on non-hot executors, compared to case where paths weights are always accurate.

[8] The number of tasks is not shown in Figure 3.15 but extracted from execution logs.

3.5 Related work

Related work is summarized in Table 3.7 and the most relevant work is discussed in the following.

- Graphene [67] schedules long tasks and those that it deems difficult to schedule first, as other (short) tasks can be placed within gaps of the schedule. An offline component computes optimized schedules for each application on a virtual space (cluster) in advance. An online component uses these pre-computed schedules to build a priority list of tasks, which it schedules on an actual cluster. Graphene assumes that resources can be shared (i.e., reassigned from on application to another) without incurring additional overheads and uses this, e.g., to ensure fairness.

- Paragon [33] and its successor Quasar [44] perform "short" (multiple seconds to a few minutes) benchmark runs of applications to determine their resource requirements and sensitivity to congestion on those resources (CPU, memory, disk, network). This information is used during execution to pack applications that are less likely to interfere with each other on the same nodes.

- TetriSched [75], an improved version of its predecessor AlSched [29], uses a Mixed Integer Linear Programming (MILP) solver to compute optimized resource schedules. TetriSched considers hardware heterogeneity as well as heterogeneity across applications when selecting resources, i.e., it is aware that different applications have different runtimes depending on the node type (e.g., with or without GPUs) and location (e.g., within the same rack or not) they're using. It does not consider DAG- or task-heterogeneity but only decides when and where resources for applications are allocated.

- Firmament [65] is a task scheduler that uses a Min-Cost Max-Flow (MCMF) solver and models the cluster as flow graph through which tasks flow. It considers task and hardware heterogeneity but not DAG-heterogeneity. Firmament is a cluster-wide task scheduler, i.e., it schedules all tasks on a cluster.

- Carbyne [66] is a DAG-heterogeneity-aware application scheduler that follows an altruistic approach when sharing resources among multiple applications. Based on knowledge of the DAG and task runtime and resource demands, Carbyne decides whether it can donate resource to other applications that are in need of them, without impacting the runtime of the donating application. It does not consider resource sharing overheads.

- HEFT [3] and CPOP [3] are DAG, task and (to a limited degree) hardware-heterogeneity-aware list scheduling algorithms, that execute tasks in ascending order of the mean execution time of the longest path to the last task across available resources. LHEFT [14] and PHEFT [40] are extensions thereof that look ahead

Scheduler	year	smallest scheduling unit	global scheduler	application heterogeneity	DAG heterogeneity	task heterogeneity	hardware heterogeneity	cluster topology	resource sharing cost	interference	uses runtime estimates	uses resource estimates
HEFT [3]	1999	task	-	+	+	+	+	+	n/a	-	+	-
CPOP [3]	1999	task	-	+	+	+	+	+	n/a	-	+	-
LHEFT [14]	2010	task	-	+	+	+	+	+	n/a	-	+	-
Spark [18]	2010	task	-	-	-	-	-	+	n/a	-	-	-
AlSched [29]	2012	app	+	+	n/a	n/a	+	+	n/a	-	-	-
YARN [39]	2013	app	+	-	n/a	n/a	-	+	n/a	-	-	-
Paragon [33]	2013	app	+	+	n/a	n/a	+	-	n/a	+	-	+
Quasar [44]	2014	app	+	+	n/a	n/a	+	-	n/a	+	-	+
PEFT [40]	2014	task	-	+	+	+	+	+	n/a	-	+	-
TetriSched [75]	2016	app	+	+	n/a	n/a	+	+	n/a	+	+	+
Firmament [65]	2016	task	+	-	-	-	-	+	-	-	-	-
Graphene [67]	2016	task	+	+	+	+	-	-	-	+	+	+
Carbyne [66]	2016	task	+	+	+	+	-	-	-	-	+	+
Decima [110]	2018	stage	+	+	+	-	-	-	+	-	-	-
HCL [84]	2017	task	-	+	+	+	+	+	+	-	+	-
HCL-SP	2018	task	-	+	+	+	+	-	+	-	+	-

Table 3.7: *Summary of related work. +/- indicate the presence/absence of a property w.r.t. heterogeneity-awareness.* n/a *indicates that a property is not applicable.*

in the DAG to assess the impact of current scheduling decisions on upcoming tasks. A greedy algorithm is used to assign tasks to resources.

- Decima [110] is a recent scheduler that uses reinforcement learning (RL) to learn the stage execution order and number of executors assigned to a job, such that an objective, e.g., job completion time, is reduced. In order train its neural network (NN), it schedules jobs in a simulated environment and learns automatically, based on feedback by the objective function. While RL is capable of automatically identifying good scheduling patterns, it requires a large number of training runs for each job to do so. Decima has been integrated in Spark.

Prior work has extensively studied and addressed various aspects of DAG-, task- and hardware-heterogeneity, albeit not always in combination. Furthermore, most schedulers are global schedulers, i.e., they schedule all applications at once, instead of only a single one, and assume that resources can be reassigned (shared) across multiple applications at no extra cost. This assumption does not always hold. For instance, Apache Spark uses long-running executor processes that require ≈1.9s to initialize[9] and additional time to start a corresponding JVM process. These executors are tied to individual application instances and cache application-specific data. In order to share the underlying resources, executors need to be shut down and restarted for another application. This, however, impairs amortization of startup and initialization costs over the course of the application execution[10].

Other scheduling strategies such as HEFT and its variants optimize schedules of individual applications. However, they have not been integrated and evaluated with cloud application frameworks, such as Apache Spark. Distributed data-processing frameworks can exhibit a complex runtime behavior. For instance, the task runtime analysis has shown the importance of the executor state on task runtimes, which they do not consider.

An important differentiation between related work is also the *smallest scheduling unit*, i.e., the unit that the lowest level scheduler of the work considers. This is either application (app), stage or task. When computing schedules, no unit smaller than the smallest scheduling unit is considered. For instance, a scheduler with applications as the smallest schedulable unit considers requirements and preferences for applications as a whole, but not for individual stages or tasks. This also impacts which scheduling challenges are faced. For instance, stragglers are only faced when scheduling individual tasks.

[9] See Section 4.2.1 for details.

[10] A detailed analysis of resource reassignment/sharing costs in Apache Spark and YARN is presented in Section 4.2.

3.6 Conclusion

This chapter presented HCL and HCL-SP as well as detailed task runtime and predictability analyses. HCL has shown that Spark applications can generally benefit from DAG-, task and hardware-heterogeneity-aware scheduling, assuming accurate task runtime predictions. An analysis of task runtimes has shown, however, that accurate task runtime prediction is unlikely to succeed, as task runtimes fluctuate even when the measured metrics are the same or similar. This also limits the achievable accuracy of any predictive model. Experiments with three ML algorithms have confirmed this. This makes the approach chosen by HCL unrealistic in practice. Nevertheless, valuable insights have been gained, namely the importance of executor state on task runtime.

Based on the gained insights, a simpler scheduling strategy has been devised: Stage packing. Stage packing exploits executor state to speed up task execution and reduce stalls during execution. Based on this new strategy, HCL-SP, a state-, DAG-, task- and hardware-heterogeneity-aware scheduler has been implemented and integrated into Spark. The scheduling algorithm of HCL-SP is computationally less complex than that of HCL, which makes it suitable for use on real systems, as shown in the evaluation. Compared to vanilla Spark on a 15 node heterogeneous test cluster, a mean execution speedup of 1.14× was achieved. Mean resource utilization efficiency was improved by the same factor. Using a simple, preference order-based executor selection strategy, the speedup was increased to 1.39×, which even surpassed the ML-based. Benefits of stage packing, as well as stage packing with preference order, can be achieved from the very first run of an application. This is in contrast to ML-based approaches, such as the ones chosen in HCL-SP and Decima. The latter requires a large number of training runs for each application [110]. Furthermore, stage packing is also usable on homogeneous clusters as well as other distributed application frameworks that exploit JIT and shared data caching on long-running executors.

Several issues remain:

- While HCL-SP's resource selection method can pick the most preferred executor from an executor pool, it is not able to determine whether it is beneficial to pick an additional executor at all, as was pointed out during the closer examination of a schedule in Section 3.4.5.5.

- Resource utilization has been increased from 12.19% for vanilla Spark, to 30.64% for Spark+HCL (SP+FP). This, however, still leaves executors idle for 69.36% of the time, while resource that they allocate (most notably memory), cannot be used by any other application. Hence, efficient resource sharing across applications is paramount.

4. Efficient resource sharing across applications at small time-scales

This chapter presents Mira, a novel resource and elastic application scheduler, that enables efficient resource sharing in homogeneous and heterogeneous clusters across applications at small, sub-second, time-scales.[1] Mira focuses on short running (less than one minute) applications, e.g., interactive data analytics as well as applications with fluctuating (changing over the course of the execution) resource demands.

Distributed application frameworks, such as Apache Spark, use long-running executors which allocate memory and compute resources for a specific application to execute short-running tasks on its behalf. Startup and initialization overheads of these executors are typically high and expected to amortize over the application runtime. In elastic scenarios, where executors are added and removed during the application runtime, amortization is less effective.

Mira allows multiple applications to sequentially share executors. This allows for startup and initialization overheads to amortize across multiple applications and thus enables more efficient sharing of resources at small, sub-second, time-scales. Furthermore, Mira adjusts executor assignments instantaneously and uses the directed acyclic graph (DAG) to estimate immediate-term resource demands. This allows it to share executors in milliseconds, instead of multiple seconds.

The main contributions of this chapter are summarized as follows:

(1) The overhead of resource allocation for Spark on top of YARN is analyzed and quantified. The resulting problem of resource sharing at small, sub-second time-scales identified.

(2) Mira, an efficient two-level resource and elastic application scheduler that reduces resource sharing delays by up to two orders of magnitude compared to Spark on YARN, and thereby enables efficient resource sharing at time-scales as low as tens of milliseconds instead of multiple seconds. Mira has been integrated into Spark. An evaluation with the TPC-DS data-analytics benchmark suite [116, 88], as representative for data-analytics workloads, shows that Mira reduces resource sharing overheads and can accelerate application execution by more than 3× in a shared environment compared to vanilla Spark on YARN.

[1] The work presented in this chapter is based on the publication "Mira: Sharing Resources for Distributed Analytics at Small Timescales" [107].

This chapter is structured as follows: First, an overview of resource sharing in Spark and YARN and the core problems is given (Section 4.1), followed by background information on the root causes of these problems and the strategies employed to mitigate their impact (Section 4.2). Subsequently, Mira is introduced (Section 4.3) and evaluated experimentally in comparison with a vanilla Spark on YARN stack (Section 4.4). Finally, related work is presented (Section 4.5) and the chapter is concluded (Section 4.7) with a discussion of Mira's concepts, the evaluation results and potential future work items (Section 4.6). Section A.2 accompanies this chapter with additional information.

4.1 Introduction

Distributed data-analytics applications are often written in data-parallel frameworks, such as MapReduce[6], Spark[18] and others [12, 96, 99, 35, 52]. Resource managers, such as YARN [39] and Mesos [21] allow multiple applications to share resources and execute at the same time. In the beginning, mostly long-running batch applications were executed on these frameworks and due to their initial success, other use-cases were added as well, such as short-running data exploration and real-time data analytics applications. [52, 99, 92, 35]. These new use-cases exhibit vastly different characteristics than traditional batch applications. First, their entire runtime can be as short as a few seconds. Second, their resource demands may fluctuate, e.g., in the case of online data analytics that need to quickly adjust their processing capacity to changes in input stream volume. As a result, resource managers are now required to manage resources at time-scales they were not designed for, i.e., seconds, instead of minutes or hours.

These smaller time-scales pose a significant challenge for resource managers, such as YARN and Mesos, and application frameworks, such as Spark. To illustrate this issue, a simple Spark application (Listing A.1), running on a YARN-managed test cluster, is sufficient (see Section A.2.2.1 and Table A.15 for details on cluster and application configurations).

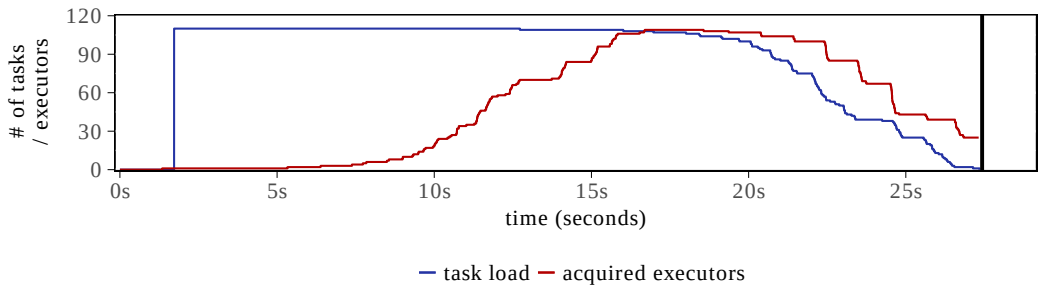


Figure 4.1: *Execution of a Spark application that spawns 110 tasks, each of which sleeps for ten seconds. The blue line shows the resource demand of the application while the red line shows the number of resources (in form of "executors") that were allocated by the application for task execution.*

Figure 4.1 shows the execution of this application. Here, 110 tasks are spawned, each of which sleeps for ten seconds and then exits. Up to 112 executors can be executed on the test cluster, hence all 110 tasks can run at the same time and the application should finish in little over ten seconds. However, as Figure 4.1 shows, its actual runtime extends to ≈2.5× of that. This highlights two issues:

(1) There is a large **delay in executor acquisition**, up to ≈15s, which is the reason for the prolonged execution.

(2) There is also a **delay in executor release** after tasks finish, despite the fact that they are idle and not needed anymore.[2]

While in long-running applications with relatively constant resource demands, both delays can be amortized, the same is not possible for short running and interactive applications with fluctuating resource demands. As will be shown later (Section 4.2.1 and Section 4.2.2), the causes for this behavior are spread across the resource manager (RM) and application scheduler (AS), hence also need to be addressed in both.

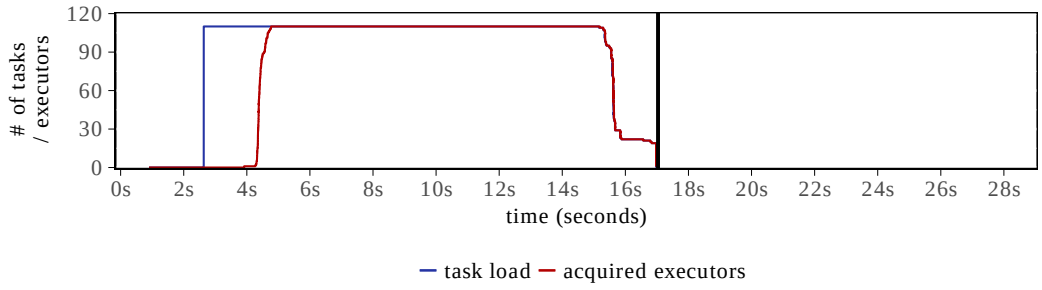
Mira addresses both issues to enable executor acquisition and release at small, sub-second time-scales. Two strategies are used:

(1) Executors are not treat as ephemeral objects that are tied to a single application, but as long-lived, shared resources. This allows Mira to amortize executor startup and initialization costs over all applications instead of just a single one and therefore to reduce recurring resource acquisition costs. As result, executors can be acquired faster.

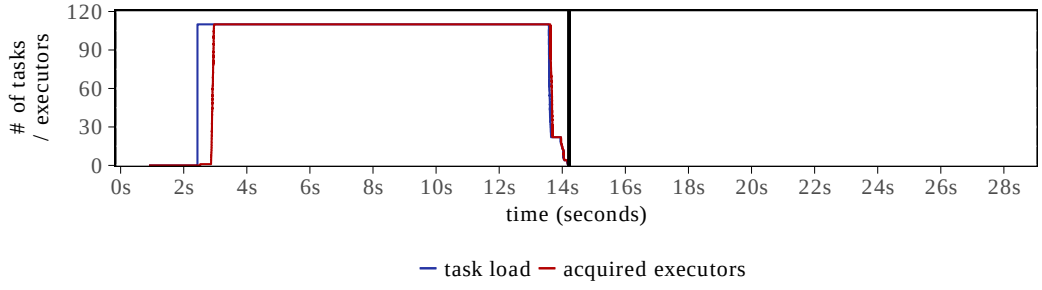
(2) Executors are released as soon as they are not needed anymore. This increases resource utilization efficiency, as idle executors are not blocked from being used by other applications. It can also reduce application runtime, as applications can acquire needed executors quicker.

The effect of both strategies can be seen in Figure 4.2a. where executors are acquired in little less than 2s (vs. up to 15s for vanilla Spark) and application runtime is reduced to 17s (vs. 28s for vanilla Spark). Furthermore, executors are released as soon as they are not needed anymore.

[2]Spark has already been configured to release executors as quickly as possible, i.e., 1s after becoming idle. Actual shut-down takes another ≈0.8s.



(a) Spark+Mira (cold)



(b) Spark+Mira (warm)

Figure 4.2: *Execution of the same Spark application as in Figure 4.1 on Mira, instead of YARN. The tails in both plots are caused by stragglers.*

Application frameworks that are written in interpreted languages, such as Spark, Flink and Tez, often employ Just-In-Time (JIT) compilation to translate often used sections into native (machine) code. Native code executes faster than interpreted code. Sharing executor across applications also allows applications to reuse native (jitted) code. This further reduces executor acquisition overheads, as parts of the Spark executor and application code has been compiled into native code. Moreover, some initialization steps have to be done only once, during executor startup. The result can be seen in Figure 4.2b.

The next section (§4.2) elaborates on the issues of delayed executor acquisition and release, and the sources thereof in the Spark and YARN application stack.

4.2 Background

Distributed application frameworks (AFs) allow developers to express algorithms using high level patterns, such as map/reduce, while the AF handles distributed execution, data distribution, task scheduling, fault handling, etc.

Figure 4.3 shows a common distributed application stack on shared clusters. Resources are managed by a resource manager (RM), such as YARN [39], Mesos [21], or others [60, 38, 65, 9, 64]. RMs enable multiple applications to coexist and share resources on the same cluster.

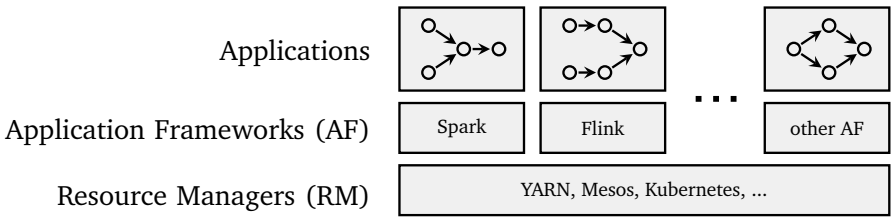


Figure 4.3: *Overview of a common distributed application stack: Applications (often represented as DAGs) execute on top of AFs that operate on a shared computing infrastructure managed by a RM.*

Generally, there are three approaches in how AFs and RMs collaborate that are defined by who controls resources and for how long.

(1) **Fine-grained resource allocation.** In this approach, used in Hadoop MapReduce [94], earlier versions of Spark [18] and others [21], control over resources stays with the RM at all times. For each application task, the RM allocates suitable resources, executes a task on behalf of the AF and releases the resources afterwards. The RM has full insight into the utilization of each resource and can share them among multiple applications. This comes at the cost of high task execution overhead, due to resource allocation and release per task, which can be prohibitive for short-running tasks.

(2) **Static coarse-grained resource allocation.** Used by Spark [18], Flink [52] and others [99, 92], this approach statically allocates a set of resources to an application for its entire runtime, giving the AF full control over them for the entire duration of the application execution. The application scheduler (AS) of an AF schedules individual tasks on the allocated resources. As resource allocation overheads can be amortized across the entire application runtime, per-task execution overheads are reduced. However, as the RM relinquishes control over these resources, it has no insight into their utilization and cannot share them with other applications, even if the owning application does not use them at all times. Consequentially, coarse-grained resource allocation is not a good fit for shared environments with fluctuating resource demands of applications.

(3) **Dynamic coarse-grained resource allocation.** This approach allows elastic expansion and contraction of the allocated set of resources of each running application. Many systems support or plan to support this (e.g., Spark [97], Flink [93], and others [100, 59]). As load increases, the AF requests additional resources from the RM and releases them once the load has decreased. Similar as in the coarse-grained mode, resource allocation costs can be amortized across multiple tasks, thus reducing the per-task execution overhead. While the RM still has no insight into resource utilization, it can regain control earlier, as AFs release resources voluntarily or upon request.

Dynamic coarse-grained resource allocation represents a compromise between execution overheads and efficient resource sharing. Nevertheless, because existing systems are not built for small time-scales and react too sluggishly to changing resource demands, they are unable to use this approach efficiently for applications with short (e.g., ten seconds) runtimes or when resource demands fluctuate on a similarly small time-scale. This is due to two fundamental issues:

(1) The **resource acquisition process is time-consuming** (see Figure 4.4), thus imposing an upper bound on the frequency with which resources can be shared.

(2) In consequence, AFs may **delay resource acquisition and hold on to resources after the need for them has ceased** (see Figure 4.4), such that they can mitigate the impact of potential future resource re-acquisition. While beneficial for an individual application, this strategy impairs overall (global) resource utilization efficiency and application performance.

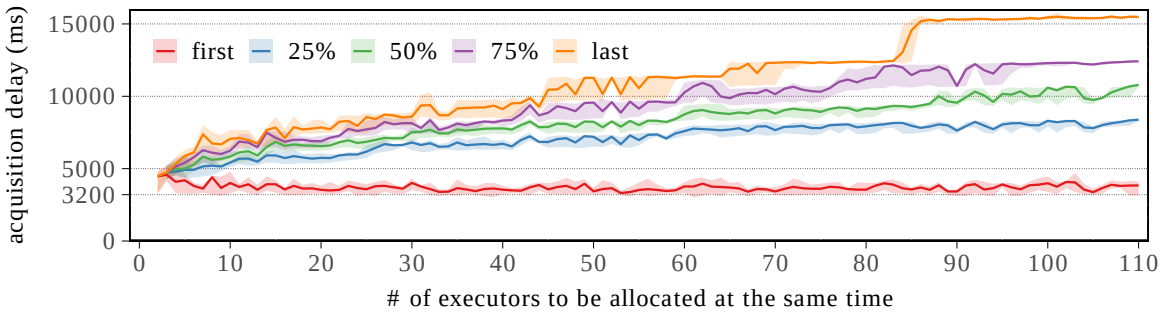
Both issues have significant impacts on interactive applications with low response time requirements and those with fluctuating resource demands. In both cases, frequent and timely executor acquisition and release are essential for high performance and efficiency. In high load scenarios where free resources are scarce, these problems get exacerbated.

In the following, executor acquisition overheads are quantified and dominating factors thereof identified. This analysis uses the popular combination of Spark as AF and YARN as RM.

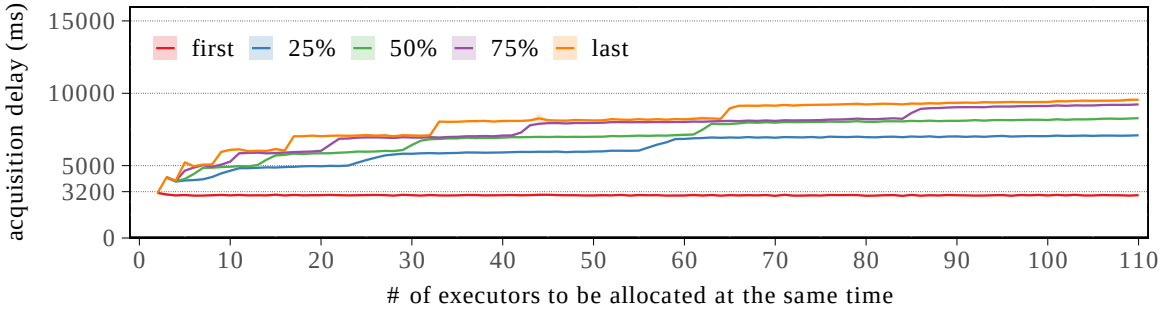
4.2.1 Executor acquisition overheads

In order to analyze executor acquisition overheads of the application stack, the test application from §4.1 is modified such that it spawns $N = 2 \ldots 110$ tasks in a loop to acquire the same number of executors. After each loop iteration, executors are released. This allows to measure the delay of acquiring N executors at a time. This delay is the overhead. For each N, the test is executed five times on the test cluster (Section A.2.2.1). The code of the test application is provided in the appendix (Listing A.2).

Results in Figure 4.4a show the average executor acquisition delay (in milliseconds) for each N for different percentiles of tasks, when executing this test with YARN as RM. To show that these overheads are not limited to YARN, the same experiment was repeated using Spark's Standalone mode [98] which uses a simple, Spark-specific resource manager. Results for the latter are shown in Figure 4.4b.



(a) Spark+YARN



(b) Spark Standalone

Figure 4.4: *Average executor acquisition delay for different percentiles. Variance is shown in both plots.*

Two results stand out:

(1) There is a base delay of ≈3.2s with both RMs to acquire the first executor. ≈1.9s can be attributed to the Spark executor startup and initialization delay. A breakdown of the individual delays is shown in Figure 4.5.

(2) There is a variable delay of up to ≈12.6s for YARN and ≈6.6s for Spark Standalone to a maximal delay of ≈15.6s and ≈9.6s respectively. The variable delay increases with N albeit not continuously but in steps. These steps are the result of Spark's executor acquisition strategy which will be discussed in the following.

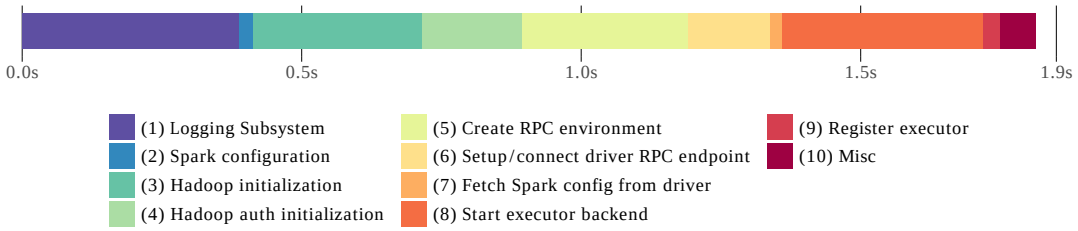


Figure 4.5: *Breakdown of Spark executor startup and initialization overheads.*

To better understand the variable fraction of the acquisition delay, Spark's executor acquisition strategy is examined in more detail. Figure 4.6 shows an excerpt of the execution of the test application for $N = 110$ executors including the number of executors Spark is requesting at each point in time (green line). In the beginning, Spark requests only a single executor. For every second that the number of executable and running tasks exceeds that of available executors, the number of requested executors is doubled, until the number of requested executors equals or exceeds the number of executable and running tasks. This strategy is responsible for ≈50% of the variable cost, as all 110 executors are requested only after ≈9s. Furthermore, it is responsible for the steps that are visible in the plot for Spark Standalone (Figure 4.4b): After each step (at 2, 4, 8, 16, ... executors), one more round – and therefore one more second – is required until the last executor has been acquired.

The same is also partially responsible for the delays seen with YARN (Figure 4.4a). YARN itself also contributes to this delay. With an increasing number of concurrent requests for executors, YARN needs longer to fulfill them. This is visible in Figure 4.6 as the widening gap between request and acquisition curves as the request curve becomes steeper, i.e., as the number of executors that are requested at the same time increases.

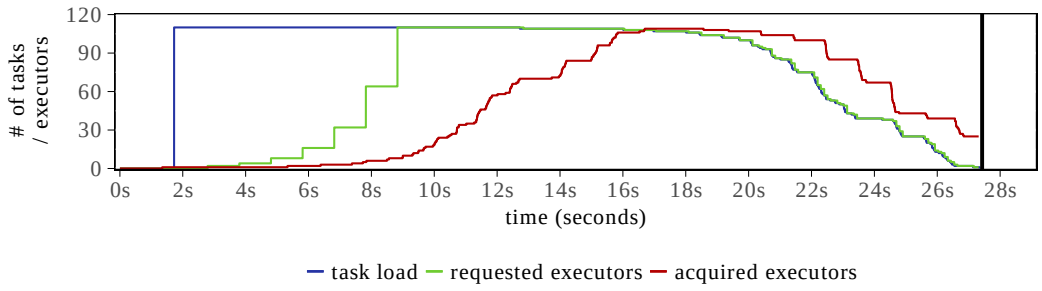


Figure 4.6: *Resource allocation in Spark+YARN.*

The observed delays are negligible for long-running applications with constant resource demands but limit the ability to adapt to workload spikes and to operate at small time-scales.

4.2.2 Resource release strategy considerations

Timely release of idle (not needed) executors is important to enable efficient resource utilization, as otherwise, resources allocated by idle executors cannot be used by other applications. When releasing executors, AFs have to make a choice, considering the high executor acquisition overheads. They can either:

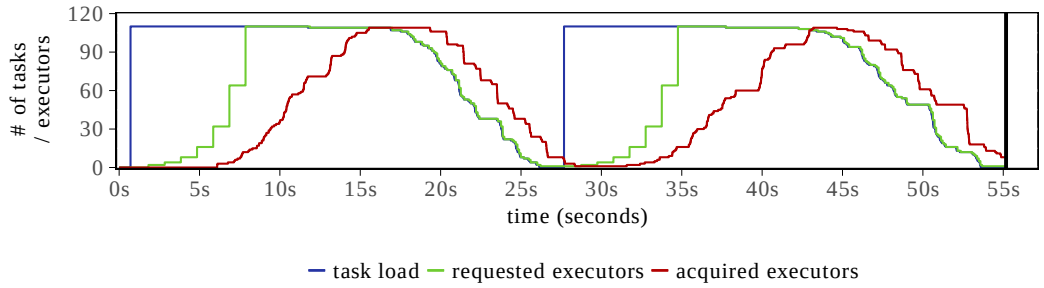
(1) Be *altruistic* and release executors quickly after they become idle such that other applications can make use of their resources. This bears the potential risk of slowing down execution of the application that releases idle executors in case they need

to re-acquire them at a later point in time. Moreover, in case executor demands of multiple applications fluctuate highly, overall resource utilization efficiency may decrease as well, as a lot of time is consumed by starting and stopping executors, during which no tasks can be executed.

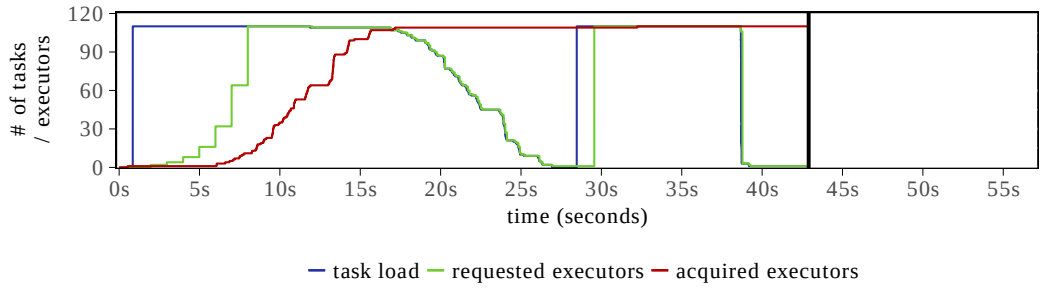
(2) Be *egoistic* and retain executors for a long time after use, thus ensuring fast execution of the current application. This comes at the cost of lower resource utilization efficiency, as resources allocated by idle executors cannot be used by other applications. Moreover, it can starve other applications for resources, thus slow down their execution.

Indeed, Spark can be configured for either approach: Idle executors can be release in as little as one second (which is used here). By default, however, they are released 60s after becoming idle, which – for short running applications – corresponds to case (2). Furthermore, executors shut down ≈0.8s after release has been initiated, hence resources of idle executors can be used by other applications no earlier than ≈1.8s after an executor becomes idle.

In order to assess the impact of either release strategy on application runtime and executor utilization, the application used in §4.2.1 is modified. Instead of spawning 110 tasks only once, it is modified to spawn another 110 tasks after the first 110 tasks have finished. In between, it waits for ten seconds, to release acquired executors when configured with a one-second release timeout. Results are shown in Figure 4.7.



(a) Release idle executors after 1s.



(b) Release idle executors after 60s.

Figure 4.7: *Consecutively allocate N=110 executors on Spark+YARN with and without release in between. The black bar represents the application end.*

If configured to release executors after one second, the number of acquired executors decreases shortly (≈1.8s) after the task load decreases, as shown in Figure 4.7a. This results in an executor utilization of 81% and a runtime of ≈28s for each stage. When configured to retain executors for 60s after they become idle (Figure 4.7b), the second stage executes in only ≈15s and therefore ≈1.9× faster than in case (1), as no recurring resource acquisition cost accrue. At the same time, executor utilization drops to 63% and the average time resources for executors are allocated, i.e. the time the executor process is running, increases by 1.22×, hence resource utilization efficiency is also reduced, despite the fact that the application executes in less time. Hence, one has to decide whether to increase resource utilization (and efficiency) at the cost of application runtime or vice versa.

The next section introduces Mira, a resource manager and elastic application scheduler that employs techniques to reduce overheads described in this chapter and furthermore implements resource management strategies that are tailored towards sharing executors at small, sub-second time-scales.

4.3 Mira

Mira is a resource manager (RM) and an application scheduler (AS), which (combined) enable resource sharing at sub-second time-scales as well as fast application execution on shared clusters. Mira is built on two key concepts:

(1) *Reusing and sequential sharing of task executors* across applications. This reduces recurring executor acquisition overheads and accelerates application code execution, by improving the exploitation of JIT compiler techniques and shared data caches.

(2) *Executors are acquired and released without delay upon changes in resource demands* of applications. This facilitates efficient executor utilization, as executor idle times are reduced.

4.3.1 Assumptions and Limitations

The concepts used in Mira are subject to the following assumptions and limitations:

- Tasks are executed on long-running executors.
- Using warm or hot executors does not increase task runtime, e.g., due to extra garbage collection when using a pre-used executor.
- Executors are stateful, e.g., contain JIT and shared data caches. Restarting an executor loses its state. Losing this state impairs performance and should be avoided.

- Executors of current AFs are exclusively associated with a single application and cannot be associated with another application without stopping and restarting the executor process.
- Executors for all applications are configured identically. Each executor can execute tasks of any application.
- Executors are started in advance. Resources allocated by executors are not needed by applications other than those using Mira as RM.
- When sharing executors across applications, data may leak across applications. Security implications of sequential executor sharing across applications are acceptable.

4.3.2 Mira system overview

Figure 4.8 shows a high-level overview of Mira's architecture along with the main communication paths. Mira consists of two major components, a resource manager (RM) (§4.3.4) and an application scheduler (AS) (§4.3.3). Multiple AS instances can coexist at the same time, each responsible for scheduling a single application.

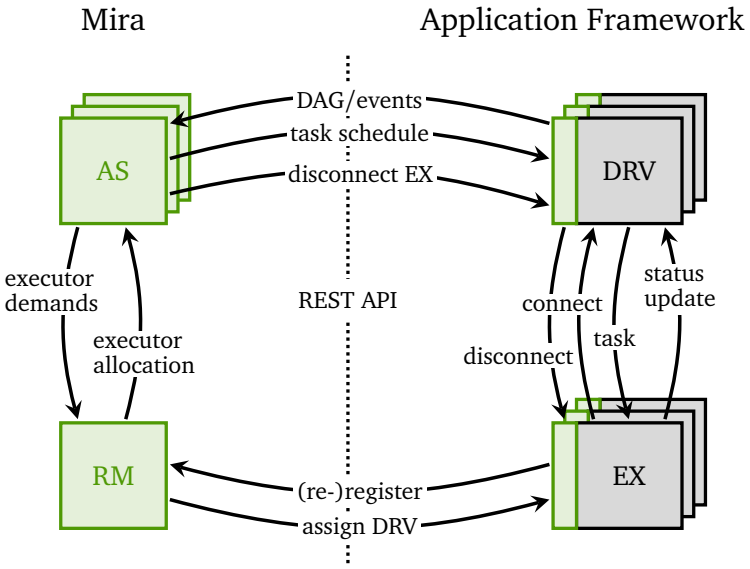


Figure 4.8: *High-level overview of the Mira architecture showing Mira (green) and application framework (AF) components (black).*

In order to integrate Mira into an AF, two of the AF's components need to be adapted.

(1) Mira provides application, DAG and task scheduling functionality for AFs. Therefore, it needs to know about the application DAG as well as scheduling-related

events (such as *stage ready* or *task finished*). This requires integration into the application driver (DRV) component[3] (Section 4.3.5).

(2) Mira provides resource management functionality for AFs and assumes execution environment (EX) reuse and sharing across applications (of the same AF). Therefore, it needs to control the EX life-cycle. This requires the AF-native EX to be managed by Mira, e.g., by wrapping it inside a Mira EX (Section 4.3.6).

Mira is based on HCL-SP (Section 3.4) and shares most its code base. Notable exceptions are its executor managing capabilities. It is implemented in ≈8k lines of C++ code, and is available at `https://github.com/zrlkau/mira`.

In the remainder of this section, all four Mira components depicted in Figure 4.8 are described. While Mira's concepts are not Spark-specific, it has been integrated into Spark as part of this work. Therefore, the integration of Mira into the DRV and EX components are described using Spark as example.

4.3.3 Application scheduler (AS)

Mira supports ASes with a strict separation of policy and mechanism. The policy, i.e., decision-making process is a component in Mira, whereas the mechanism, i.e., the execution of these decisions, is integrated into the AF's DRV component. This architecture allows Mira to support multiple scheduler implementations to coexist and the sharing of scheduler policy implementations across multiple AFs. Mira's AS for Spark is based on the stage packing scheduler in HCL-SP but does not use task runtime predictions nor node class preference orders. This scheduler is described in Section 3.4.2.1. It has been extended for dynamic executor acquisition and release.

4.3.3.1 Executor demand determination

Mira's AS explicitly requests executors from and releases them to the RM. In order to determine the executor demand d, it uses the following metrics:

(1) The total number of unfinished tasks t_{curr} in all executable stages.

(2) The total number of tasks t_{next} in stages that immediately follow executable stages.

(3) A multiplicative *resource scale-out factor* α that reduces the number of requested executors by a factor in the range $(0, 1]$ (default: 1). This factor allows to limit resource usage while remaining elastic.

[3] Here it is assumed that this functionality resides in the driver component. If this is not the case for a specific AF, the respective component needs to be adapted.

d is computed according to Equation 4.1 using the above listed metrics and the current number of allocated executors e.

$$d = \begin{cases} t_{curr} \times \alpha & \text{if } t_{curr} \times \alpha \geq e \\ \max(t_{next}, t_{curr}) \times \alpha & \text{otherwise} \end{cases} \tag{4.1}$$

The number of executors requested is therefore determined by the number of currently executable tasks (t_{curr}). As soon as the task load (adjusted by α) falls below the number of allocated executors, Mira looks ahead in the DAG and determines the immediate-term need for executors (t_{next}). All executors that are not needed now nor in the immediate-term are released. In contrast to vanilla Spark, Mira does not introduce any additional delay when requesting or releasing executors. Executor demand is recomputed upon every task/stage ready/finished event. If the executor demand has changed, the RM is immediately informed about the new demand.

The resource scale-out factor α effectively limits the number of executors that an application is allowed to acquire relative to the number of executable tasks. The reasons to impose such a limit may not be obvious at first glance and are as follows:

(1) Scaling out by a certain factor typically does not reduce the application runtime by the same factor, hence scaling out maximally, i.e., assigning only a single task per stage to each executor, is not always desirable. Reasons therefor are manifold and include increased system overheads (e.g., due to additional communication and coordination requirements) and the reduced ability to mitigate (hide) stragglers with fewer tasks per executor (see Section 2.2.4.1 for details).

(2) The task runtime analysis in Section 3.3.1.2 concluded, that the very first task of a stage that is executed on a cold or warm executor, runs significantly (up to ≈15.4×) slower than subsequent tasks (assuming otherwise comparable tasks) on then-hot executors. By reducing the number of executors per stage, the number of first tasks on cold or warm executors is decreased as well, and those running on hot executors is increased.

An alternative approach, which is possible in Spark (and virtually any other task-based distributed system), is to simply limit the total number of executors an application can use at any time, which can also force the AS to schedule more tasks per executor. This approach has two major disadvantages:

(1) It limits elasticity of the application execution as even when many tasks are executable, the number of total executors cannot exceed the limit.

(2) It does not consider that the number of executable tasks can vary greatly during the execution of an application (e.g., shown in Figures 4.14 and 4.17c). If the set limit is too low, application performance may be impaired in phases with many executable tasks. If the limit is set too high, phases with fewer executable tasks

may spread out across too many executors and reduce fraction of tasks that run on hot executors.

The resource scale-out factor determines the minimum fraction of tasks that run on hot executors, independently of the number of executable tasks, and does not set a hard limit on elasticity.

4.3.4 Resource manager (RM)

The resource manager (RM) determines the executor share of each application according to a weighted fair-share policy and assigns and revokes them to and from individual applications. Mira's RM schedules only EX instances as resources and not CPU cores and memory, such as YARN [39] or Mesos [21]. Mira maintains a pool of pre-started executors, such that application demand can be met without the need to start new executors.

Enabled by executor sharing and the resulting ability to acquire executors with sub-second delay, Mira's RM has been designed under the assumption that executors can be reassigned between applications without imposing significant overheads. Hence, Mira assigns and reclaims executors to and from applications immediately, without any waiting period, if demands and allowances change. Mira depends on cooperative ASes that release executors upon request, as soon one is or becomes idle, i.e., has finished executing a task. This facilitates efficient resource utilization, as idle executors, that are needed by other applications, are not hogged by applications that don't need them.

This is in contrast to other RMs. For instance, YARN uses multi-second timeouts before reclaiming resources from applications. Mesos uses an offer-based approach, where applications cannot actively request resources but have to wait for an offer from Mesos, which might not come soon enough when sub-second latency is desired.

4.3.4.1 Resource scheduling policy

Mira does not implement multiple submission queues to determine application priorities and therefore executor shares, as other RMs, such as YARN, but uses a simple weighted fair-share policy. Per default, all applications have the same weight (priority), i.e., all applications are entitled to the same share of available executors.

The fair-share policy distributes a set of executors R across the set of running applications A. Each application $a \in A$ with relative weight $w_a \in (0,1]$, $\sum_{a \in A} w_a = 1$ and an executor demand of d_a is assigned a subset of executors $r_a \subseteq R$ according to Equation 4.2:

$$|r_a| = \min\left(d_a, |R| \times w_a\right) \tag{4.2}$$

$|R| \times w_a$ represents the fair share of the application, i.e., the number of executors $|R|$ times the relative weight w_a of the application a. In case $d_a < |R| \times w_a$, i.e., the executor demand of an application is smaller than it's fair share, remaining executors are redistributed across all other applications by decreasing the relative weight w_a of a and

increasing the weights of all other applications a' for which the executor demand $d_{a'}$ exceeds the current assignment $|r_{a'}|$.

In case application a exceeds its current fair-share (e.g., because a new application was submitted and its fair share was therefore reduced, or because an application that has not demanded its fair share so far but does so now), the RM immediately requests the corresponding AS to release executors, until a does not exceed its fair share anymore. ASes are expected to be cooperative and release executors upon request. Mira does not currently implement any methods to enforce the release of executors, such as task preemption. Listing 4.1 shows simplified C++ code of the RM.

4.3.4.2 Scalability

The scalability of the RM is correlated to the number of concurrently running applications, as the likelihood of executor demand changes, that need to be processed by the RM, increases with the number of applications. It does not correlate to the number of tasks of applications, as no per-task decisions are made by the RM. In order to cope with a large number of executor demand change requests, simple optimizations have been implemented:

(1) Executor change events are only generated by the AS if they imply a change in executor allowance, i.e. only if executor demand crosses (i.e., was equal or higher and is lower now or was equal or lower and exceeds it now) executor allowance.

(2) Resource change events are only processed by the RM if they imply a change in executor assignments, e.g. an application requests more executors while unassigned executors remain or its fair share is not yet exhausted.

(3) The RM aggregates multiple events (line 6 in Listing 4.1) for *timeout* seconds before re-evaluating resource assignments, ensuring that the RM recomputes executor allowances as events are pending in a high-load scenario, while also ensuring quick response times in low-load scenarios.

In multi-app benchmarks (§4.4.4) a median delay of executor assignment re-evaluation of less than 100μs was measured.

```
1  void ResourceManager::handle_events() {
2    bool reschedule = false;
3
4    // Process pending events and decide whether they affect executor assignments or not
5    // until all pending events have been processed or 'timeout' has passed.
6    while (pending_events() && !timeout) {
7      Event ev = get_next_event();
8      if (impacts_executor_assignment(ev))
9        reschedule = true;
10   }
11
12   // Trigger executor assignment recomputation if necessary.
13   if (reschedule)
14     update_executor_assignment();
15 }
16
17 void ResourceManager::update_executor_assignment() {
18   // Instruct application schedulers to release executors if they have more than they are
19   // entitled to.
20   for (Application app : applications) {
21     if (app.actual_share() > app.allowed_share())
22       app.release_executors(app.actual_share() - app.allowed_share());
23   }
24
25   // Assign unassigned (free) executors as long as there are any and as long as
26   // applications demand more executors than are assigned to them.
27   while (!free_executor_pool.empty() && application_demand_exceeds_assignments()) {
28     for (Application app : applications) {
29       // Check if the application is entitled to another executor.
30       if (app.executor_demand() > app.actual_share() &&
31           app.actual_share() < app.allowed_share()) {
32         // Assign a free executor to the application.
33         Executor ex = free_executor_pool.get();
34         app.assign_executor(ex);
35       }
36
37       // Check if more executors are available and stop assignments if no more free
38       // executors are available.
39       if (free_executor_pool.empty())
40         break;
41     }
42   }
43 }
```

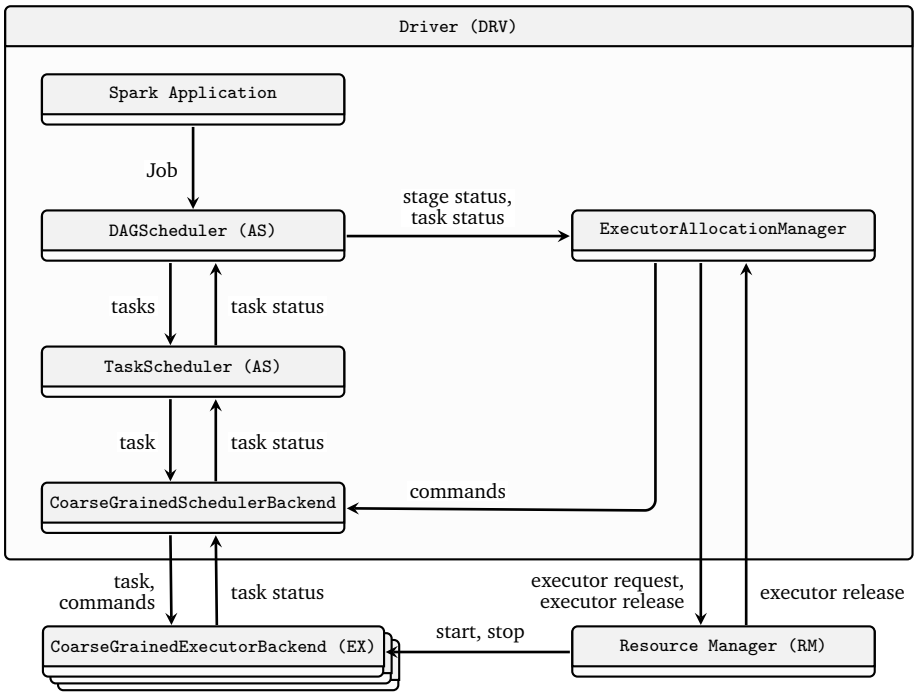
Listing 4.1: *Simplified C++ code of Mira's RM event handling and executor assignment updates*

4.3.5 Application driver (DRV)

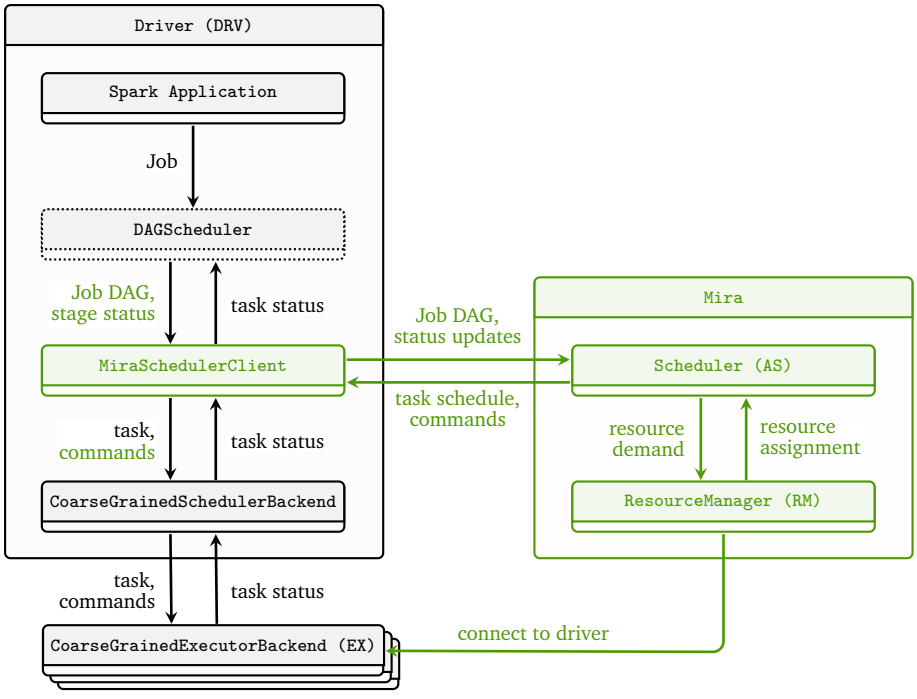
The application driver of an AF executes serial portions of the application and uses AF provided functionality to distribute execution of parallel portions across available EXes. DAG and task scheduling functionality are assumed to be part of the driver, resource demands are determined and communicated to the RM. Mira replaces large parts of this functionality and therefore needs to be integrated into the DRV. As part of this work, Mira has been integrated into Spark. For that reason, the remainder of this section describes the integration at the concrete example of Spark's application driver.

Figure 4.9 shows a schematic overview of the relevant components of the Spark driver with (Figure 4.9b) and without (Figure 4.9a) Mira integration. Information about Spark, DAGs, jobs, stages and tasks are provided in Section 2.3 of the background chapter. The following is a brief summary: A Spark application consists of a set of DAGs (jobs). Each DAG consists of one or more stages, which apply a specific function on input data to produce output data. Each stage consists of one or more tasks, each applying the specified function on a partition of the input data. The `DAGScheduler` computes a data-dependency-satisfying order in which stages are executed. Once all data dependencies have been met, a stage and all of its tasks are ready to be executed. An executable stage is submitted to the `TaskScheduler`, which assigns tasks to individual EX. Each EX is an instance of the `CoarseGrainedExecutorBackend`. Each EX is connected to a `CoarseGrainedSchedulerBackend` instance inside the driver, which is used to relay commands and events to and from the EX. The `ExecutionAllocationManager` is notified by the `DAGScheduler` about stage/task ready/finished events such that it can determine the current executor demand. If the executor demand exceeds the current executor allocation, more executors are requested, if the executor demand is lower than the current executor allocation, executors are released after a configurable timeout (see Section 4.2 for details). A schematic overview of the Spark driver is shown in Figure 4.9a.

Mira replaces large parts of the above described functionality (shown in Figure 4.9b). The `TaskScheduler` and `ExecutorAllocationManager` were removed and the `DAGScheduler` modified, such that it only creates and forwards job DAGs as well as stage status updates to the `MiraSchedulerClient`. The latter is only a relay and forwards all information, i.e., job DAGs, stage, task and executor status updates to the external Mira AS. In return, the Mira AS transmits task schedules to the `MiraSchedulerClient`, which serializes the task binary and closure, and sends it to the `CoarseGrainedSchedulerBackend` for execution on the selected EX, i.e., it takes over functionality from the original `TaskScheduler`. In contrast to vanilla Spark, where the `ExecutorAllocationManager` works transparently in the background to ensure that an appropriate number of executors is available at all times, in Mira, the AS and RM explicitly *negotiate* the number of executors that an application is entitled to. This negotiation process is described in detail in Section 4.3.3 and Section 4.3.4.



(a) Vanilla Spark DRV

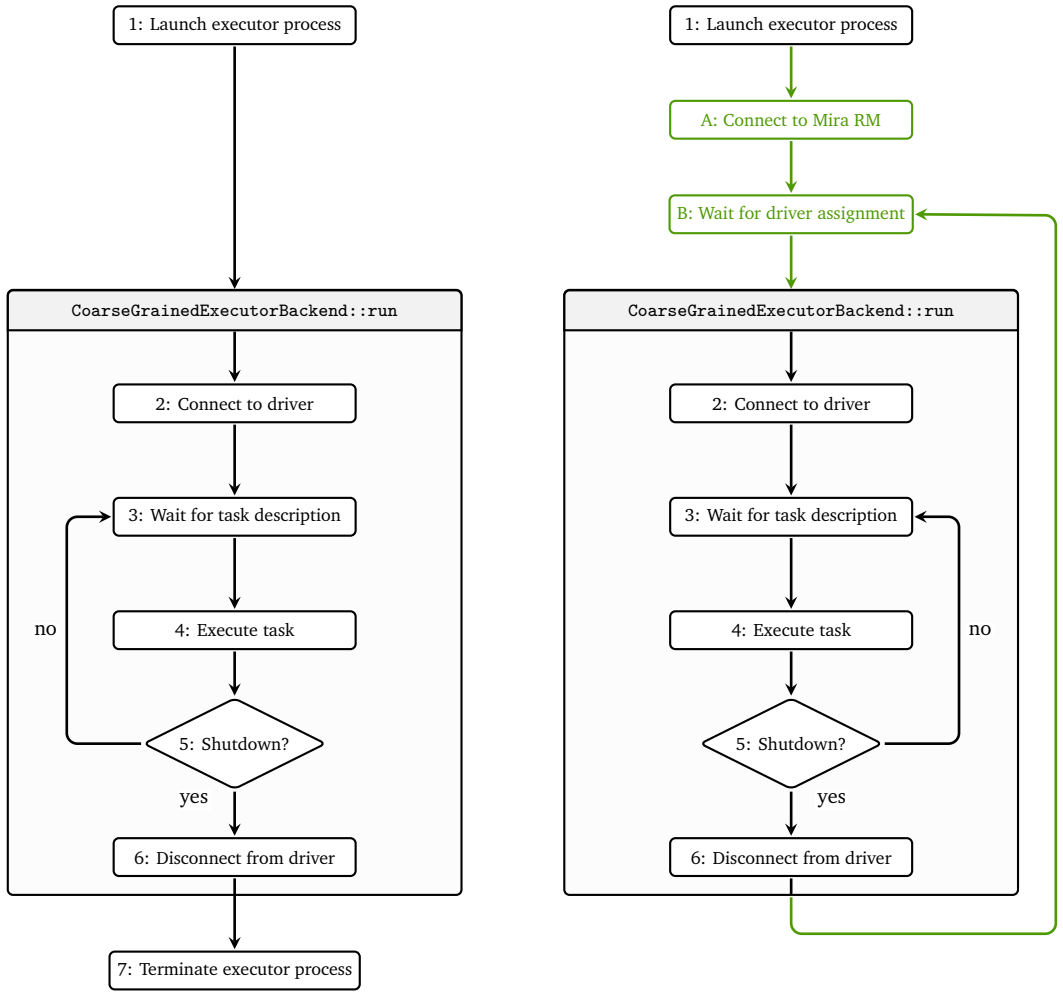


(b) Mira Spark DRV

Figure 4.9: *Overview over Spark driver (DRV) components and the integration of Mira into Spark's driver and their interplay and affiliation with RM, AS and EX. New components are highlighted in green and dotted components were significantly modified.*

4.3.6 Execution environment (EX)

Mira's EX does not replace the AF's native EX but wraps (and extends) it such that it enables the executor process to be persistent. This avoids the high recurring acquisition costs shown in §4.2.1 by making certain tasks, e.g., the repeated loading of libraries and initialization of subsystems, such as the logging subsystem, unnecessary (as shown in Figure 4.5 for Spark and also analyzed for Python-based frameworks by Oakes et al. [111]).



(a) Vanilla Spark EX **(b)** Mira Spark EX

Figure 4.10: *Conceptual difference between a vanilla Spark EX and a Spark EX, wrapped in a Mira EX. Parts marked green are added by Mira.*

For Spark and other JVM-based frameworks, persistence also allows the conservation of Just-In-Time (JIT) compiler caches. This enables parts of the application to execute as native (compiled) code earlier, instead of as interpreted code. Applications of the same AF, e.g., Spark applications, utilize AF-provided functionality, such as data and execution management as well as implementations of popular algorithms, such as for

linear algebra, machine learning (ML) and data processing and querying (e.g., Structured Query Language (SQL)). In consequence, they also share parts of their code and can therefore benefit from JIT caches that were filled by other applications.

Figure 4.10b shows Mira's EX integration with the Spark framework and contrasts it to Spark's native EX (Figure 4.10a). Mira's EX wraps Spark's native `CoarseGrainedExecutorBackend` without modifying its core functionality. The integration is explained in the following.

- **Application assignment and task execution.** Upon start (Figure 4.10b, step 1), Mira's EX wrapper connects to the Mira RM (step A) and waits until a driver has been assigned (step B). Only after Mira's RM has assigned a Spark driver to the EX, Mira's EX wrapper calls Sparks' EX `CoarseGrainedExecutorBackend::run` function, which represents the actual EX functionality and connects to the assigned Spark driver (step 2). In vanilla Spark (Figure 4.10a), said function (step 2) is called right after the executor process has been launched (step 1). After the `CoarseGrainedExecutorBackend` has connected to Spark's driver, it waits for a task description (step 3), which contains all necessary information to execute a task, i.e., the task executable code as well as its closure, and, upon reception thereof, executes the task (step 4). Once the task has completed, it confirms its completion with the Spark driver and waits for another task description, until the Spark driver instructs the `CoarseGrainedExecutorBackend` to shut down (step 5).

- **Application deassignment and disconnect.** Once a shutdown instruction has been received, the vanilla Spark EX disconnects from the Spark driver (step 6). Here, the Mira EX wrapper takes back control of the executor process and instead of terminating it (Figure 4.10a, step 7), waits for another driver assignment from Mira's RM (Figure 4.10b, step B).

- **Data retention after deassignment.** Spark stores input and intermediate results data in a distributed manner across all EX instances and exchanges data items between EXes and the DRV on demand. A central registry of the location(s) of data items (stored in so called *blocks* and identified by *block IDs*) is maintained by the `BlockManager` in the Spark application DRV. After an EX is removed (deassigned) from an application, other EXes of the same application lose access to all data blocks stored on the removed EX. Spark is partially resilient against data loss and may react in any of the following ways:

  (1) In the best case, multiple copies of the same data block exist. As long as a single copy is still accessible, data blocks can be retrieved, albeit potentially with a delay (up to 15s, per default[4]), if the first attempt to retrieve a data block was from a removed executor.

[4] `spark.shuffle.io.maxRetries = 3`, `spark.shuffle.io.retryWait = 5s`

(2) If no copy of a data block is accessible anymore, Spark will attempt to recompute it, which consumes extra time and should therefore be avoided.

(3) After several (3, per default[5]) failed attempts to retrieve data, Spark will fail and terminate the application execution.

In order to avoid the latter two cases, Spark on Mira transfers all unique data items from an EX to the DRV and updates the `BlockManager`'s registry accordingly, and only then proceeds with removing an EX from an application.

4.3.7 Component interaction

Figure 4.11 depicts the high level component interaction paths within Mira. Processes can be (roughly) divided in three categories: resource management (blue), executor life-cycle (orange) and task execution (green).

- **Resource management.** For resource management Mira's RM mainly communicates with various AS instances to receive resource demands (R1) and update resource assignments (R2). Resource assignment increases are implicit by assigning unassigned executors to an application (E4), whereas resource assignment decreases are done explicitly and rely on a cooperative AS (R3).

- **Executor life-cycle.** After launching a Mira EX (E1), it registers itself with the RM (E2) and requests a driver assignment (E3). If there is demand for this executor, the RM will assign it to an application driver (E4), which the EX will connect tor (E5). The driver will then register the new driver with the AS (E5) and start requesting tasks for execution. Once demand has ceased (R1) the RM reduces the executor share of an application (R2), the executor disconnects from the driver (E6) and requests a new driver assignment from the RM (E3).

- **Task execution.** Once an executor has been assigned to an application (E5), the driver requests tasks on its behalf from the AS (T1). As long tasks are available for the executor, they are being executed (T2).

[5] `spark.rpc.numRetries = 3`, `spark.rpc.retry.wait = 3s`

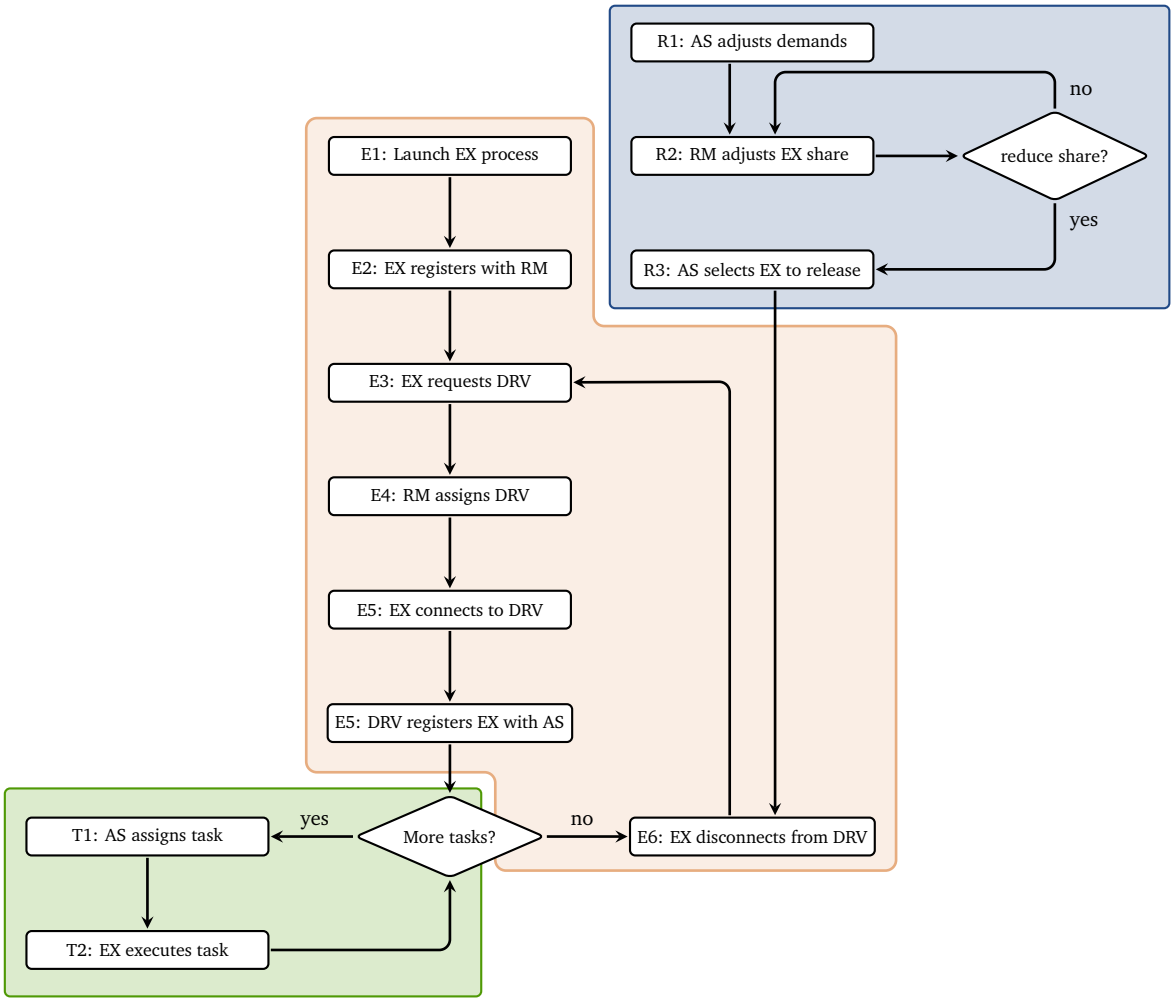


Figure 4.11: *Interaction of Mira components. Processes inside the green box represent the normal task execution. The blue box shows resource management processes and the orange box shows the executor* life cycle.

4.3.8 Mira AF Representational State Transfer (REST) interface

Mira's REST interface for AFs is based on that of HCL-SP and extends it for executor management. An interface description can be found in Section A.2.

4.4 Evaluation

This evaluation compares vanilla Spark on YARN (Spark+YARN) with the Mira-integrated version of Spark (Spark+Mira). This evaluation addresses three main questions:

(1) *Can Mira reduce the recurring resource acquisition overheads?* The experiments presented in Section 4.4.2 and Section 4.4.3, the impact of reduced system overheads for executor acquisition on application runtime is evaluated. As these experiments

show, resource acquisition overheads are reduced by up to two orders of magnitude when running Spark on Mira instead of on YARN. At the same time, the reduced overheads allow Spark to execute TPC-DS queries ≈1.5× faster on average (Section 4.4.3.3).

(2) *What impact does code execution acceleration by reusing warm executors have?* In Section 4.4.3.2 experiments compare application runtimes when using cold and warm executors. Results indicate a significant execution acceleration and reduction in application runtime by a factor of ≈2× on average.

(3) *What benefits can Mira facilitate in a multi-application setting?* In Section 4.4.4, experiments combining effects from all of Mira's features are combined in a scenario where TPC-DS queries compete for resources with a background application that exerts constant load on the cluster, showing that Spark on Mira executes applications up to 3.1× faster than on YARN (≈1.9× on average). Furthermore, the impact of immediate executor release on resource utilization efficiency is evaluated.

4.4.1 Test setup

All experiments use the test setup, software and configuration described in Section A.2.2, unless specifically noted otherwise. Furthermore, each experiment is repeated five times and average numbers are reported.

4.4.1.1 Applications

Throughout this evaluation, a Spark implementation of the TPC-DS benchmark suite is used. A description of this benchmark is in Section A.1.4. For single-application experiments and as foreground application in multi-application experiments, 90 TPC-DS queries are used. As background application in the multi-application experiments, a simple application that repeatedly executes stages of 8192 tasks, each running for one second, is used. The tasks sleep for one second and therefore use very little CPU and memory, nevertheless, they allocate (block) executors. This application generates sufficient task load to utilize all available executors of the cluster for the entire duration of each experiment. This background application represents a good case for executor sharing, as tasks finish frequently. If a background application had longer running tasks, task preemption may be necessary to retain a similar level of freedom for the RM to reassign executors at small time-scales.

4.4.1.2 Configurations

Table 4.1 lists the configurations compared in this evaluation.

Configuration	Description
Spark+YARN (1s)	This configuration refers to Spark applications running on YARN as resource manager using `spark.dynamicAllocation.executorIdleTimeout=1s` (which is the minimal possible value). This corresponds to a resource sharing optimized setting as Spark is voluntarily releasing executors only one second after they've become idle.
Spark+YARN (60s)	This configuration refers to Spark applications running on YARN as resource manager using `spark.dynamicAllocation.executorIdleTimeout=60s`. This is the default Spark setting and corresponds to a *egoistic* setting, where applications hold on to idle executors for 60s in case they are needed again. This configuration typically increases performance of individual applications at the cost of resource utilization efficiency.
Spark+Mira (cold)	This configuration refers to Spark applications running on Mira as resource manager and application scheduler using pre-started, but initially cold executors, i.e., they have never been connected with any Spark application driver and have a cold JIT cache.
Spark+Mira (warm)	This configuration refers to Spark applications running on Mira as resource manager and application scheduler using pre-started and warm executors, i.e., they have already been connected with one or more Spark application drivers before and have a warm JIT cache.

Table 4.1: *Configurations compared in this evaluation.*

4.4.2 Micro-benchmarks

This experiment evaluates the executor acquisition delay of Spark+Mira. It uses the application from Listing A.2, which spawns N=2...110 tasks at a time, to acquire N=2...110 executors. The corresponding results for Spark+YARN and Spark+Standalone can be found in Section 4.2.1. Results are shown in Figure 4.12.

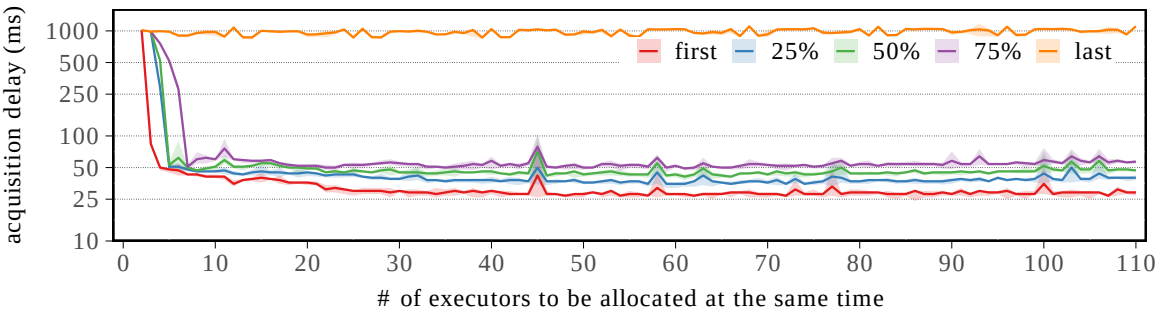


Figure 4.12: *Executor acquisition delay: Delay from increase of task load to the availability of the same number of executors, for N=2 . . . 110 tasks. Plot shows availability of first/last executor and three percentiles. The y-axis is $\log_{10}$ scaled. Ribbons show the variance.*

On average, Spark+Mira acquires the first executor after 21ms compared to 3214ms and 2884ms for Spark+YARN (Figure 4.4a) and Spark Standalone respectively, which represents a speedup of 153× and 137× (Figure 4.4b). This increases to 42ms for the 75th percentile, compared to 9408ms for Spark+YARN and 7577ms for Spark Standalone, or a factor of 224× and 180× respectively.

During each run, N is increased by one, hence a previously unused and therefore cold executor is added. While this executor has already been started, it hasn't been fully initialized yet and needs to perform all but the first initialization step (see Figure 4.5). This is why it registers later, after 1058ms on average[6], compared to 11403ms (Spark+YARN, Figure 4.4a) and 8080ms (Spark standalone, Figure 4.4b) or a factor of 10.8× and 7.6× respectively. This reduces the tail latency of executor acquisition, which can be an important factor to improve overall service performance [32]. The last warm executor is acquired after ≈100ms on average for Spark+Mira. As the usage of cold executors can be considered a rare event (each executor is cold only once and warm thereafter), it can be assumed that in most cases, all executors are warm and that the maximal acquisition delay is in the order of 100ms, further reducing tail latency.

Furthermore, Mira shows an improved scaling behavior, requiring 1.4× longer to start 101–110 executors at the same time than 2–11 executors[7], whereas Spark+YARN requires 2.1× and Spark standalone 1.8× longer.

[6]The first initialization step does not account for the entire difference between 1058ms and the average executor initialization duration of 1813ms (see Section 4.2.1 for details). A possible reason for this is that the JVM process itself performs various initialization tasks during startup, which overlap with the executor initialization in vanilla Spark and therefore prolong the initialization process. As, in the case of Mira, the full executor initialization is performed later, these processes do not overlap, hence the executor initialization can be finished quicker.

[7]The outlier for starting a single (cold) executor with Mira is not considered, as it does not accurately reflect the scaling behavior.

4.4.3 Single application benchmarks

This section presents single-application TPC-DS experiments. The objective of these experiments is to show the impact of the reduced acquisition overhead of Mira on application runtime. In a separate experiment, the impact of code execution acceleration (due to JIT-cache exploitation) is evaluated. Three types of experiments are conducted:

(1) As baseline, TPC-DS queries are executed on Spark+YARN with dynamic resource allocation enabled. Two configurations are used here: The first one uses the default resource release timeout (60s), which increases the performance of individual applications, whereas the second one uses the minimal timeout of one second, which improves resource sharing at smaller time-scales and fairness among applications.

(2) In order to evaluate the benefits of pre-started executors and Mira's executor acquisition strategy (Section 4.3.3.1), the same set of TPC-DS queries is executed on Spark+Mira and compared to the baseline. In-between each run, all Mira EXes are restarted to avoid warm-up effects across application runs.

(3) In order to evaluate the benefits of execution acceleration with warm executors, the previous experiment is repeated without restarting Mira's EXes. Before the actual benchmark runs, an initial warm-up run is performed to warm up executors. This scenario constitutes an optimal case, as the same query is executed repeatedly on warm EXes, maximizing the impact of the JIT-cache. Input data and task binaries are not cached across application runs.

Results for each query are shown in Figure 4.13 and summarized in Table 4.2.

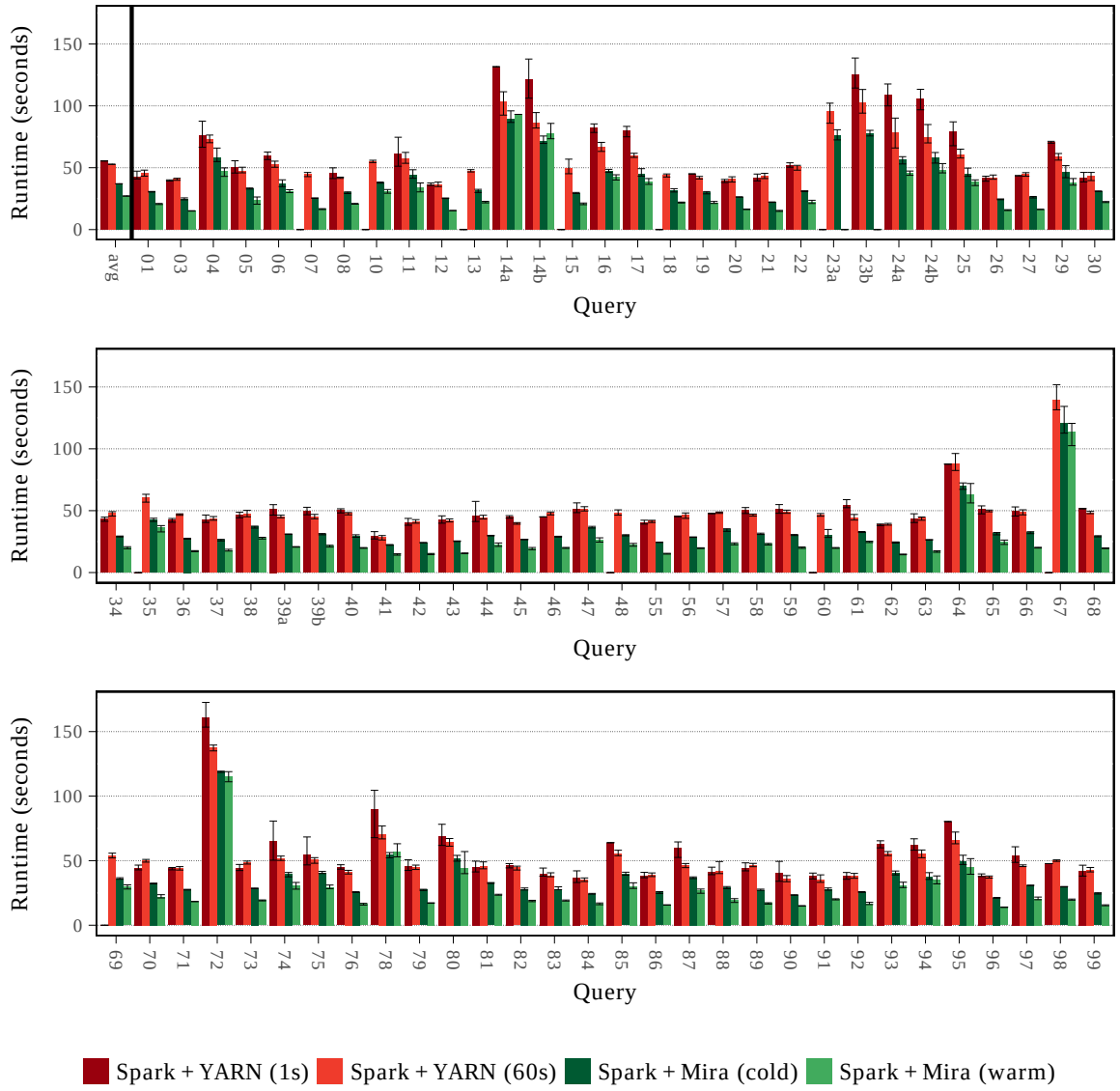


Figure 4.13: *Comparison of TPC-DS query runtime (shorter is better) between Spark+YARN and Spark+Mira on cold and warm executors in a single application setting using the default 60s executor release timeout as well as 1s. Average results are provided before the vertical line. A missing bar indicates that all test results were filtered-out due to persistent abnormal behavior.*

Figure 4.13 shows how different configurations affect query execution time. As expected, Spark+YARN with the default release duration of 60s performs better than with the sharing-optimized setting of one second, which is 1.05× slower than the former.

However, in the Spark+YARN (1s) configuration, a small number of tasks got *stuck* and prolonged the application execution by several tens of seconds before finishing (without error). This abnormal behavior was observed in ≈35% of all test runs, which were filtered out from the results presented here, as they are not representative of Spark's re-

source allocation or release strategy, but indicate the presence of a bug in Spark.[8] The same behavior was also observed in the Spark+Mira (warm) configuration, albeit only in ≈1.9% of all cases. These results were filtered out as well. All other configurations were unaffected. Causes are discussed in Section 4.4.3.4.

Impact of stage packing. As Mira's AS uses the same stage packing approach as HCL-SP, parts of the runtime prediction with cold executors is due to this approach. However, vanilla Spark's slow executor acquisition strategy increases the number of tasks on hot executors as well. In consequence, the benefit that Mira draws from stage packing compared to Spark+YARN in the elastic setting are lower than those of HCL-SP compared to vanilla Spark in a non-elastic setting. For instance, the first 24 stages (with one task each) of each TPC-DS query, which vanilla Spark scattered across all executors, are scheduled on only a single executor in the elastic setting, as Spark+YARN does not acquire more than one executor as long as the task load remains at most one. Furthermore, this one executor is never released, as the minimum number of executors is set to one. A quantification of the impact of stage packing is not possible, as Mira does not implement a scheduling strategy that resembles Spark's.

4.4.3.1 Impact of resource allocation strategy

On average, Spark+Mira (cold) is able to accelerate application execution compared to either Spark+YARN configuration. Compared to Spark+YARN (60s), a speedup of 1.43× on average can be realized. As executors are initially cold in both cases and no executor is re-acquired in the Spark+YARN (60s) case (i.e., no JIT cache is ever lost either), this speedup must be attributed to Mira's executor acquisition strategy.[9] Comparing the same Mira configuration to Spark+YARN (1s), an average speedup of 1.50× is achieved.

Spark+Mira is able to realize these performance improvements despite the fact that it acquires and releases executors about 3.0× – 4.0× more often than either Spark+YARN configuration, as shown in Table 4.2.

Configuration	Average runtime	Total number of executor acquisitions
Spark+YARN (60s)	52.8s	38815
Spark+YARN (1s)	55.5s	51181
Spark+Mira (cold)	37.0s	155712
Spark+Mira (warm)	27.1s	155197

Table 4.2: *Results summary for the single-application benchmarks.*

[8] Including filtered-out results, Spark+YARN (1s) is 1.59× slower than Spark+YARN (60s).
[9] Aside from a small impact of stage packing, as discussed earlier.

4.4.3.2 Code execution acceleration

Using Spark+Mira with warm executors, the speedup increases to 1.95× compared to Spark+YARN (60s) and 2.05× compared to Spark+YARN (1s). The difference in speedup compared to Spark+Mira (cold) reflects two aspects enabled by warm executors:

(1) Warm executors connect and initialize quicker once assigned to an application, as shown in the micro-benchmarks (Figure 4.12).

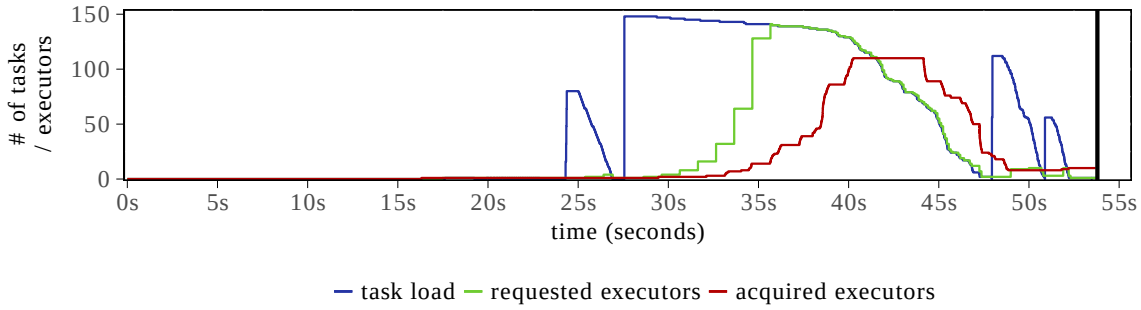
(2) Tasks can benefit from warm JIT caches from the very beginning.

The impact of the latter aspect is maximized here, as the same functions are executed repeatedly, which represents an optimal case for the JIT cache exploitation. Real-world results are likely to show a reduced benefit. Unfortunately, it is not possible to attribute fractions of the achieved speedup to either aspect. Multi-application benchmarks (Section 4.4.4) represent a non-optimal case, as here, different queries are executed back-to-back.

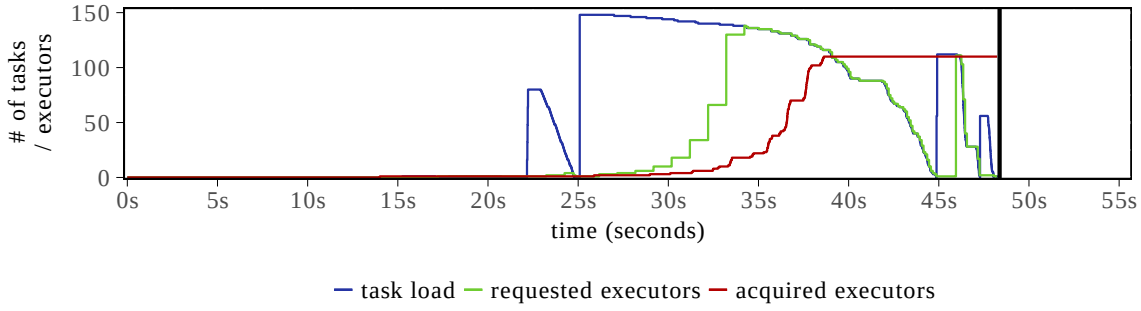
4.4.3.3 Analysis of system overheads (results breakdown)

In the following, results for TPC-DS query 59 are broken down and the impact of Mira's executor acquisition strategy, as well as the reuse of warm executors are analyzed. TPC-DS query 59 has been chosen as example, since it is not too complex for analysis, while also not being too trivial, such that none of the effects are visible in the breakdown.

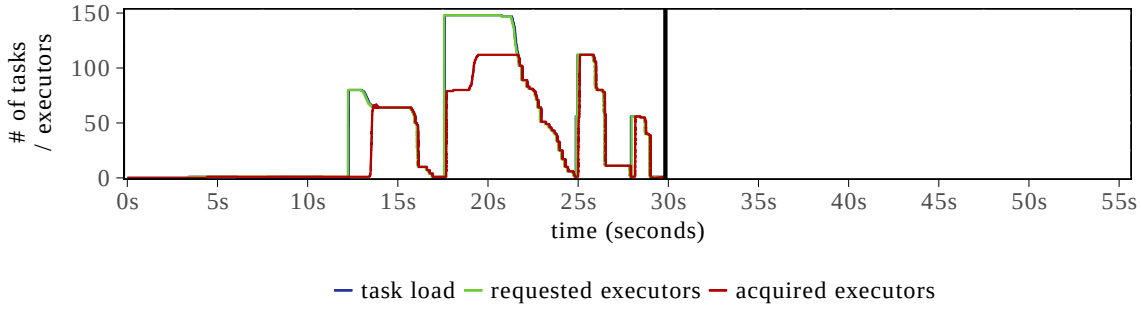
Figures 4.14a and 4.14b show the task load as well as requested and acquired executors throughout the execution of query 59 on Spark+YARN with a one second and 60s executor release timeout. Figures 4.14c and 4.14d show the same but on Spark+Mira with cold and warm executors.



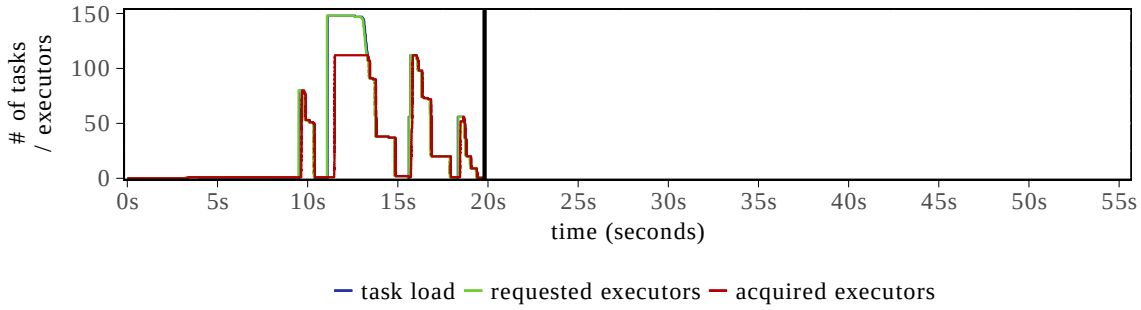
(a) Spark+YARN (1s)



(b) Spark+YARN (60s)



(c) Spark+Mira (cold)



(d) Spark+Mira (warm)

Figure 4.14: *Plot of TPC-DS query 59 (complete application execution) on 112 executors. In plots (c) and (d) the blue and green line coincide. The vertical black bar indicates the end of the execution.*

A noticeable artifact in the breakdown is that it takes several seconds (≈15s for Spark+YARN and ≈3s for Spark+Mira) before the first executor is acquired by the application driver (which is started at zero seconds). The main culprit for this delay is the admission process of the RM. While Mira is significantly faster here, it is not due to improvements within Mira but because Mira's Spark integration is based on Spark's Standalone mode, which is simpler than YARN. Spark Standalone was not considered as reference in this evaluation, however, as it is limited w.r.t. resource sharing among multiple applications, i.e., it cannot dynamically reassign allocated resources from one application to another [127].

Once the first executor has been started, a series of 24 single-task stages are executed, which requires about six seconds with any configuration. Subsequently, task load increases and runtime differences, due to conceptual differences between Spark+YARN and Spark+Mira, become apparent.

Most notably, Spark+Mira (warm) requests and acquires executors almost instantaneously, whereas Spark+YARN, in line with previous results (Section 4.2), requires ≈13.1s to acquire the maximal number of executors. Moreover, executor release is similarly faster with Spark+Mira than with Spark+YARN, even compared to the sharing optimized settings.

A comparison of Figures 4.14a and 4.14b shows why Spark+YARN with sharing optimized executor release settings performs worse than with default settings. After the second, main task set has been processed (at ≈45s), two more smaller task sets follow. With sharing optimized settings, however, Spark has already released a majority of its executors and due to its slow acquisition process, fails to increase the number of available executors in a timely manner, which results in a prolonged execution of these tasks.

In Mira, on the other hand, executor acquisition is very quick and executors can be acquired quickly with cold (Figure 4.14c) and warm (Figure 4.14d) executors, leading to an overall faster application execution. Furthermore, it also releases executors faster than either Spark+YARN configuration, which improves resource sharing with other applications.

4.4.3.4 Outliers

During the single application experiments, the Spark+YARN (1s) configuration exhibited an abnormal behavior, in that a small fraction of tasks got *stuck* for several tens of seconds and therefore prolonged the execution of the application in ≈35% of test runs. A similar behavior was observed in the Spark+Mira (warm) configuration, albeit at a much smaller scale (≈1.9%). Due to the complexity and size of the Spark code base, it was not possible to identify the root cause with certainty but it could be traced back to Spark and does not lie within Mira or its Spark-integration code changes.

A closer analysis showed that Spark executors attempts to access a shared data block, while a connection was being closed, because the remote executor, where the data block resided, was shut down. Three attempts, each five seconds apart, are made to access

the data before. However, these three attempts were repeated five times each, which caused a total delay of ≈75s for each affected task. The task itself finished successfully afterwards, indicating that Spark was able to retrieve the data from elsewhere (it is not uncommon that data blocks are replicated across multiple executors). As executors were not forcefully shut down by the resource manager (YARN), a likely explanation for this behavior is that Spark failed to reliably update its internal block location registry (which resides in the driver) or that executors relied on cached, but stale location data.

As the default Spark+YARN configuration with a 60s executor release timeout virtually never releases executors during these experiments, the issue did not show itself here. Moreover, the Mira integration modified parts of the Spark code related to this functionality and may have, as side effect, reduced the likelihood of this malfunction to occur, which is why it occurs only rarely in Spark+Mira (warm) and never in Spark+Mira (cold) experiments.

4.4.4 Multi-application experiments

This section presents the results of multi-application experiments. In said experiments, two applications are executed concurrently in order to show differences in resource sharing behavior between Spark+YARN and Spark+Mira.

- A background (BG) application (see Section 4.4.1.1 for a description), which is sufficient to fully utilize all available executors in the cluster was executed on the cluster.
- As foreground (FG) application, the same TPC-DS queries as in the preceding single-application experiments are used.

This setup forces the RM to actively balance resource assignments between both applications and demonstrates the combined effects of different resource/executor allocation and release strategies as well as the impact of more effective JIT exploitation due to executor sharing.

For these experiments, two queues, one for FG and one for BG applications, have been configured in YARN's capacity scheduler, both with a guaranteed minimum share of 50% and a maximum share of 99%[10] of all resources. Mira's resource manager uses comparable settings with equal application weights (see Section 4.3.4.1 for details).

As YARN's default settings can lead to significant delays in resource reassignment, and therefore inflated runtimes, said settings have been modified according to Table A.15 to allow YARN to react more quickly to changing resource demands. The settings in Table A.15 have been chosen as they provide the best performance for YARN in empirical tests, improving its performance in the test scenario by ≈1.14× over default YARN settings (for the Spark+YARN (1s) configuration).

[10] A maximum queue share of 100% effectively blocked a 2nd application from starting, due to lack of resources, hence, as a workaround, the limit was set to 99%.

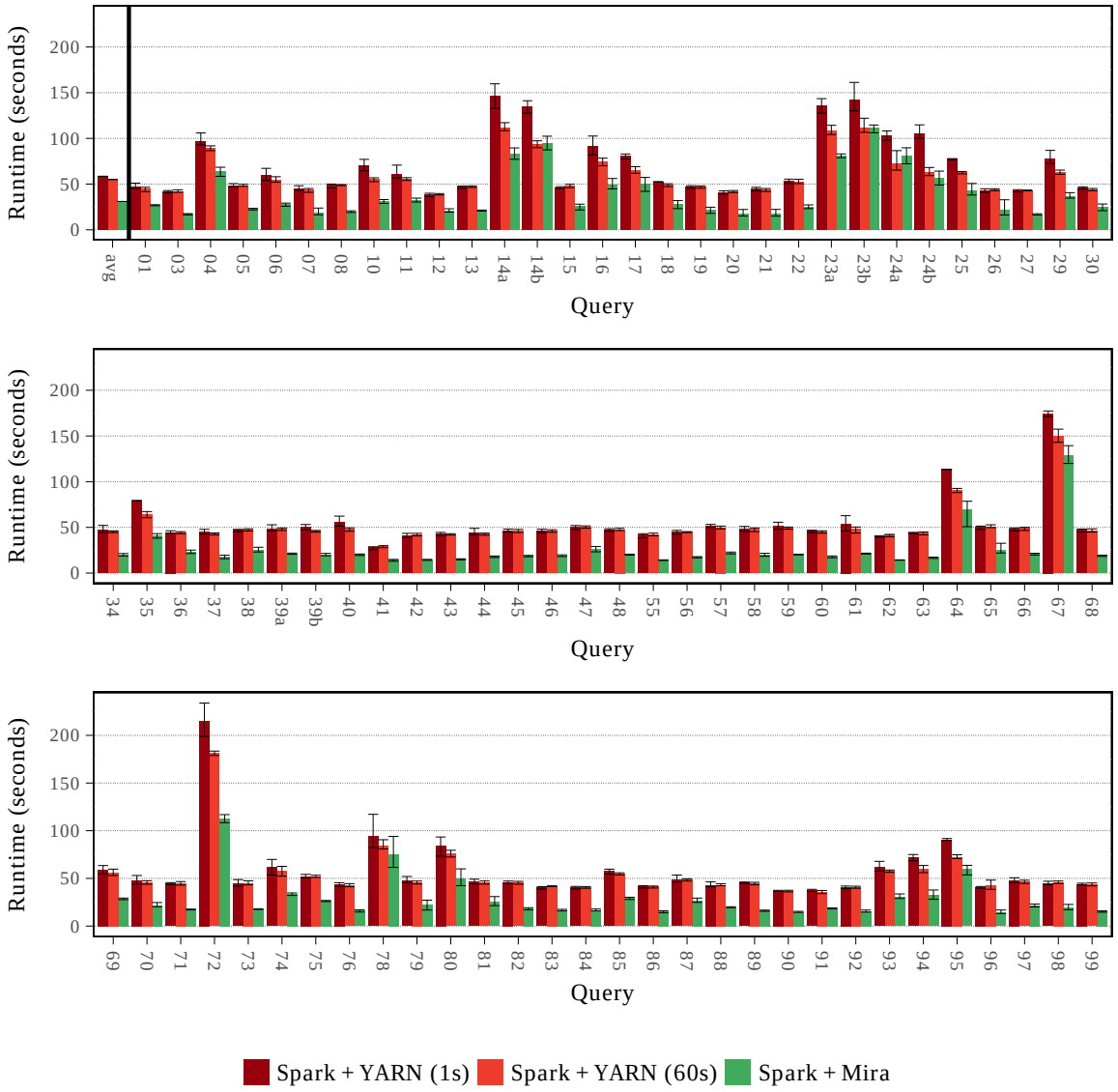
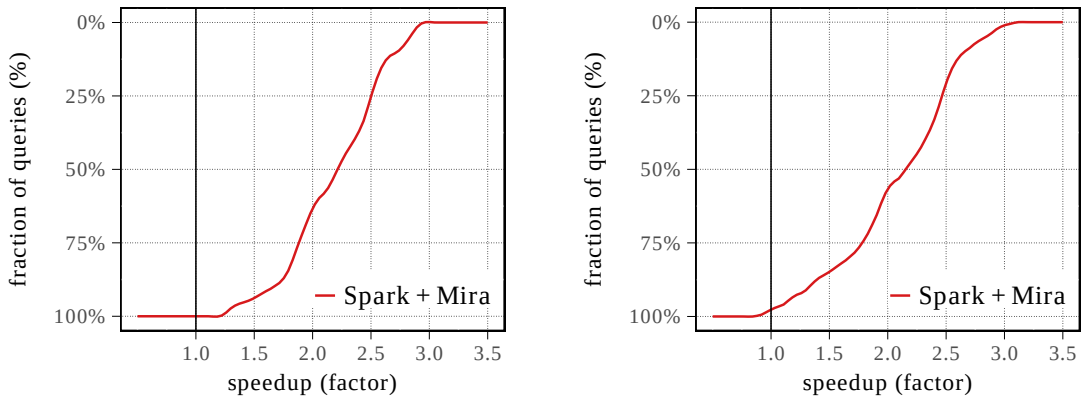


Figure 4.15: *Runtime comparison (shorter is better) between Spark+YARN and Spark+Mira with constant BG load.*



(a) Spark+Mira vs. Spark+YARN (1s)

(b) Spark+Mira vs. Spark+YARN (60s)

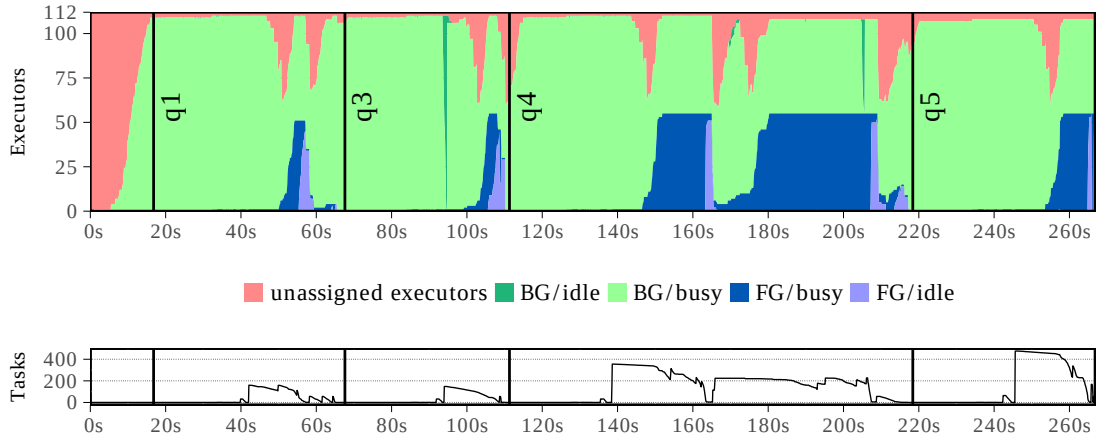
Figure 4.16: *Speedup of Spark+Mira compared to Spark+YARN with constant BG load.*

The complete results are shown in Figure 4.15 and a summary Cumulative Distribution Function (CDF) plot is shown in Figure 4.16. Spark+Mira is able to achieve an average runtime reduction of ≈1.9× compared to Spark+YARN (1s) and ≈1.8× compared to Spark+YARN (60s)[11] and a maximum speedup of ≈2.9× compared to Spark+YARN (1s) and ≈3.1× compared to Spark+YARN (60s). In the latter case, minor performance regressions for two queries were observed, which is not surprising, as in some cases, not releasing executors once they become idle can accelerate individual applications. At the same time, however, Spark+Mira achieves a higher average BG task throughput of 91 tasks per second vs. 87 for Spark+YARN (1s) and 84 for Spark+YARN (60s).[12] Results are analyzed more closely in the following.

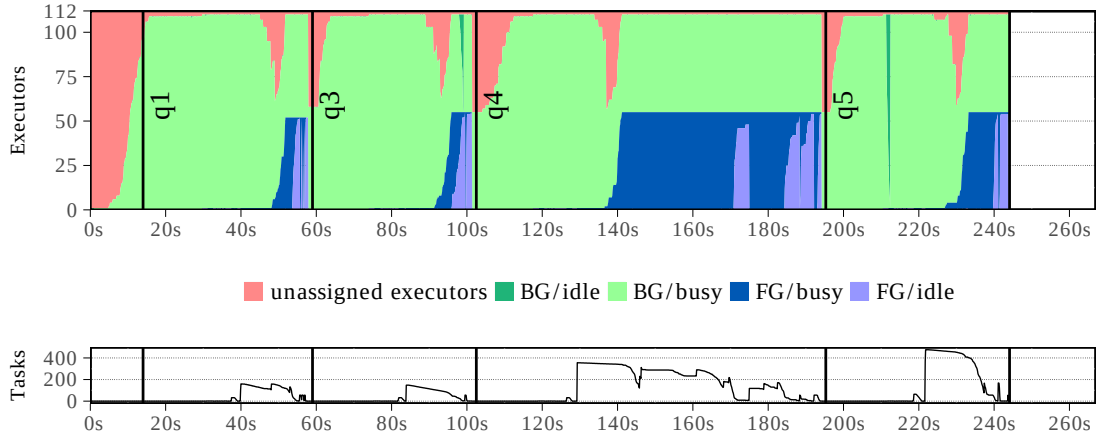
Figures 4.17a and 4.17b as well as Figure 4.17c show the allocation of resources to the BG and FG applications in a benchmark run on Spark+YARN and Spark+Mira, along with the task load at any point in time. Areas in green (blue) represent executors allocated to the BG (FG) application. Red areas represent executors that the RM has not assigned to any application or are in-between assignments. Light blue (dark green) areas represent executors that are currently allocated by the FG (BG) application but not used.

[11] Runs that exhibited the issue described in Section 4.4.3.4 have been filtered out.

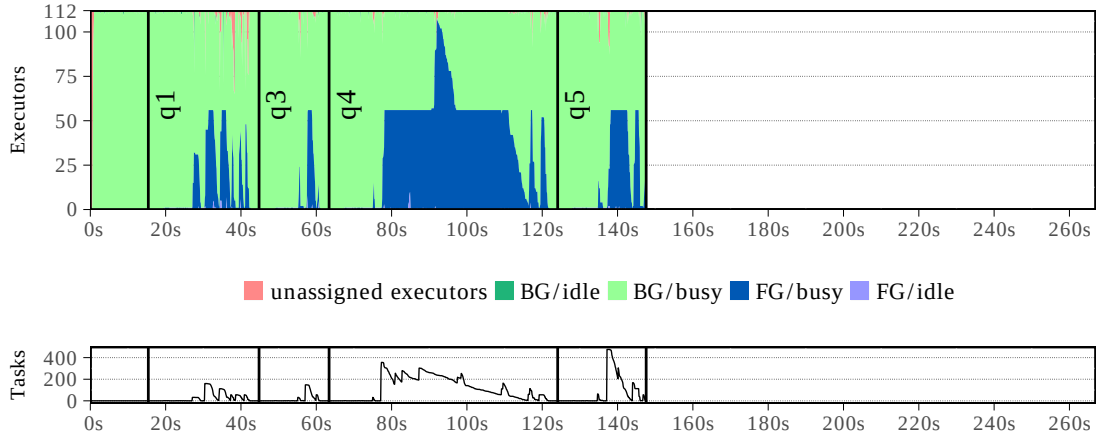
[12] Note that the background tasks' runtime was fixed at one second, thus they are not able to benefit from warm-up effects.



(a) Spark+YARN (1s)



(b) Spark+YARN (60s)



(c) Spark+Mira

Figure 4.17: *Excerpt of Spark+YARN multi-application benchmark run showing the resource sharing between BG and FG applications (top) and the corresponding task load (maximal parallelism) of the FG application (bottom). Submission of FG applications is indicated by labeled vertical black lines.*

A comparison of all three figures highlights multiple points:

- Spark+Mira is able to execute the same number of queries in 148s compared to 267s and 244s for Spark+YARN (1s) and Spark+YARN (60s) respectively, showing that it is able to execute these queries faster and, by extension, to utilize resources more efficiently. This is due to the lower executor acquisition delay for the FG application (compare delay from query start to increase of executor allocation) as well as the overall shorter task runtime (compare width of blue areas).

- There are fewer unassigned executors during transition (from BG to FG and vice versa) periods, as shown by the virtually non-existent red spikes in Figure 4.17c, compared to both Spark+YARN plots. This shows that executors spend less time in resource-management-related tasks in Spark+Mira, which increases the relative time spent executing application tasks. It can be assumed that with more concurrently running applications, and therefore a likely higher frequency of executor reassignments, Spark+YARN will fall behind Spark+Mira even further in this metric.

- Spark+YARN (60s) does not release executors but hogs them (as shown by the size of light blue areas), compared to either Spark+YARN (1s) and Spark+Mira. This allows it to execute applications slightly faster than Spark+YARN (1s), but at the cost less efficiently utilized resources.

- The high delay of executor re-acquisition for Spark+YARN (see also Figure 4.7) can be seen during the execution of query 4 in Figure 4.17a where Spark releases executors due to a short dip in task load (at ≈165s) just to reacquire them a few seconds later, with a multi-second delay.

- Spark+Mira shows a short spike in FG application resource allocation at ≈90s. As Mira's RM was configured to assign both, FG and BG applications an equal share of executors, the reason for this behavior is not obvious at first glance. The reason is, that the BG application finishes a stage which causes a short drop in executor demands. This, in turn, is immediately exploited by the FG application to acquire more executors until the BG application demands these executors back, shortly thereafter. In fact, BG application stage ends can also be observed in both Spark+YARN plots as dark green spikes from the top at ≈95s and ≈210s. Spark+YARN, however, reacts too slowly to reassign idle executors to the FG application.

The efficiency metrics in Table 4.3 show the differences more concretely. Metrics are shown in units of executor × seconds (ES) which corresponds to accumulated area of the different colors in Figure 4.17 and shows how many executors are used for what purpose, e.g., if two executors are idle for five seconds, this results in $2 \times 5 = 10$ ES.

Spark+Mira is reducing share of idle and free (unassigned) executors from 13.7% and 12.8% for Spark+YARN (60s) and Spark+YARN (1s) respectively to 3.5% or by a factor of 3.9× and 3.6×. This demonstrates Mira's ability to utilize executors more efficiently, while also executing applications faster, as shown above.

	FG Application		BG Application			
	busy	**idle**	**busy**	**idle**	**free**	**total**
Spark+Mira	49226	3243	362217	10758	1052	426496
Spark+YARN (1s)	58754	13944	551069	18506	57167	699440
Spark+YARN (60s)	58138	26415	433855	14403	37605	570416

(a) absolute executor × seconds (ES)

	FG Application		BG Application			
	busy	**idle**	**busy**	**idle**	**free**	**total**
Spark+Mira	11.54%	0.76%	84.93%	2.52%	0.25%	100%
Spark+YARN (1s)	8.40%	1.99%	78.79%	2.65%	8.17%	100%
Spark+YARN (60s)	10.19%	4.63%	76.06%	2.52%	6.59%	100%

(b) relative to the total ES

Table 4.3: *Efficiency metrics. Values are averaged accumulates over all test runs.*

	Spark+YARN (1s)	**Spark+YARN (60s)**	**Spark+Mira**
Avg. FG runtime	58.6s	55.0s	31.0s
Avg. FG speedup	1×	1.06×	1.89×
Avg. BG task throughput	86.52 tasks/s	83.76 tasks/s	90.72 tasks/s

Table 4.4: *Summary of results for the multi-application experiments. Speedup is relative to Spark+YARN (1s).*

4.4.5 Summary

The conducted experiments show that Spark+Mira can reduce runtime of the tested applications significantly, while utilizing resources more efficiently. The reason for these benefits are its reduced executor acquisition overhead, its immediate executor release, as well as the execution acceleration due to a more effective JIT-cache exploitation on warm executors. While virtually all tested TPC-DS queries benefited from Mira, shorter

queries did so to a larger degree than longer queries. This matches expectations as with the former, executor demand fluctuates on smaller time-scales than with the latter.

4.5 Related work

4.5.1 Resource managers and schedulers

YARN[39] and Mesos[21] are two of the most commonly used resource schedulers, both operating on a two-level structure (AFs/RM), but they are incapable to work efficiently on small time-scales. The issues with YARN have been demonstrated throughput this chapter. Mesos does not allow applications to request resources but makes periodic offers, which applications can accept or reject. This makes low-latency resource acquisition difficult, as the application has no control over the timing and frequency of resource offers.

There are various other approaches to cluster resource management. Kubernetes [64] is a container orchestration system based upon Google's internal Borg [60] and Omega [38] focusing on reliable, scalable and elastic deployment of service oriented workloads. Resources managers have also been used to solve various optimization problems, such as workload-specific co-scheduling [44, 33], resource packing [45] and near-term capacity planning [75] and overall optimization of scheduling decisions at scale [65, 9, 36]. These works are orthogonal to Mira, which focuses on minimizing recurring resource acquisition costs and maximizing sharing efficiency. The concept of a persistent EX, however, could also be applied to them in order to reduce resource sharing overheads and to accelerate task execution. Sparrow [36], for instance, assumes that executor acquisition and sharing among applications does not impose any overhead. Mira can therefore constitute one step towards their proposed distributed framework [37].

4.5.2 Application frameworks and schedulers

Tez [59] is a meta-framework that focuses on providing common functionality (e.g. a DAG scheduler), similarly to Mira, for application frameworks, such as Hive [12] and Pig [96]. In contrast to Mira, Tez requires YARN as resource manager and therefore suffers from similar resource acquisition and sharing shortcomings as demonstrated in this chapter.

Storm[99], Flink[52] and Apex[92] are predominantly stream processing frameworks, though the latter two can also be used for batch processing. Mira's low-latency resource sharing could be beneficial here to quickly adapt the application resource pool to changes in load.

4.5.3 Execution environment optimizations

Lion et al. [72] have done an extensive study of the impact and sources of JVM cold start costs across several distributed frameworks and applications, such as Spark, Hive and HDFS, and conclude that the warm-up time is a common bottleneck for short-running jobs. They present a transparent client/server JVM replacement that enables a persistent, warmed-up Java Virtual Machine (JVM) process to be reused repeatedly and offers a large speedup for certain workloads. Mira applies the same optimizations, but it achieves it by reusing executors. Both approaches have advantages. While their work promises to not impair security and is transparent to the AF, the approach Mira chose is potentially more efficient, as it can also reduce recurring initialization cost. Furthermore, as shown in Section 4.2 and Section 4.4, adapting scheduling strategies can further improve performance and resource utilization efficiency, which is beyond the scope of their work.

Similar optimizations exist for other environments than JVM. Oakes et al. [111] have studied overheads of Python and containers in serverless frameworks and unveil significant overheads for cold-starting Docker containers and the Python runtime. They present approaches on how to reduce both, e.g. by forking new Python runtimes from existing ones. A similar approach is used in Sand [91].

These optimization techniques may be combined with Mira's resource acquisition and release strategy as alternative to executor sharing. This would partially alleviate security concerns at the cost of increased executor sharing overheads, as executors would need to be initialized during resource transfer from on application to another.

4.6 Discussion & future work

This section discusses disadvantages of Mira's approach, provides potential solutions to them, and motivates possible future work. Furthermore, an unexpected discovery made during evaluation is discussed.

4.6.1 Security implications of executor sharing

Executor sharing is a major source of Mira's performance advantage over other systems, as it allows Mira to reduce executor acquisition overheads as well as code execution acceleration. It is, however, also its greatest potential weakness w.r.t. deployment of Mira in a production environment, as security is a major concern in environments where resources are shared with potentially untrusted third parties. Strict separation of data is therefore important.

Mira's executor sharing, in its current form, weakens this separation, as executor processes may be shared with other applications and users. While executors are only shared sequentially, i.e., there is a temporary, exclusive association of an executor with an application, residual state may still exist once an executor is reassigned to another application.

Upon acquiring an executor, a subsequent application may scan accessible memory, file descriptors and network sockets, that were not properly closed by the previous application, for classified information. Conversely, an application may inject malicious code into the executor, e.g., by loading a manipulated version of a library, which will then be used by all subsequent applications on the same executor. Both attack methods have to be prevented in an environment where trust among sharing parties has not been established.

While there are many methods to reduce the attack surface, e.g., by ensuring file descriptors and sockets have been closed before reassigning executors to other applications, security will most likely not be equivalent to systems where executors are not shared. For this reason, *executor pools*, as discussed in §4.6.3 in more detail, provide a viable tradeoff between performance and security. Sharing can be limited to executors from certain pools. Access to executor pools can be limited using a permission system similar to that of POSIX file systems, such that only trusted users and applications can share executors among each other.

Another potential solution is to restrict executor sharing to *safe* applications, e.g., such where the user itself is not allowed to execute arbitrary code on an executor but has to assemble functionality from a predefined set of *safe* function blocks. This, approach may be applicable to machine learning training and inference, where a few standardized function blocks can be used to assemble applications. Another use-case are data-analytics tasks, similar to those evaluated in this chapter, where users only provide data which they can query using a higher-level language, such as SQL.

4.6.2 Non-uniform resource demands

The current *one-size-fits-all* approach, where all applications and tasks share uniformly configured executors, falls short in cases where applications have non-uniform resource demands, e.g., w.r.t. the maximal JVM heap size. Using *executor pools* (§4.6.3) can also provide a viable solution to this issue, as they allow using differently configured executors.

As additional optimization that exceeds capabilities of current systems, Mira may select executors from different pools not only on a per application, but also on a per stage (or even task) basis. As applications consist of multiple, heterogeneous stages that execute different functions on different (amounts of) data and may or may not benefit from accelerators (such as GPUs), tasks of different stages likely have varying ideal resource preferences, which can be expressed as set of executor parameters.

While the determination of said preferences and the corresponding executor parameters is a non-trivial challenge (as the work on HCL has demonstrated), if limited to a small set of parameters, e.g., only the heap size, it might be possible to learn optimal parameters over time. Mira already monitors heap utilization of each task, which could be used to train a simple ML model that predicts the ideal (e.g., minimal) heap size parameter for executors depending on the executed task (or stage). The benefits from such

a capability is an increased resource utilization, as more executors can be placed on a single node and/or the frequency of garbage collection can be reduced, if memory-heavy tasks can run on executors with larger heaps than memory-light tasks.

4.6.3 Executor pools

Executor pools address issues where executors cannot or should not be shared among different applications or users, or need to be discriminated, e.g., due to different settings or resource requirements. The basic idea is that each pool can have different access requirements and/or properties, which are shared by all executors within a pool. By providing a limited set of pools, Mira could address the above mentioned issues (Sections 4.6.1 and 4.6.2).

The drawback of executor pools is that they limit the potential for executor sharing and thus the freedom and flexibility of the RM which narrows the advantage of the executor sharing concept. However, as mitigation strategy, dynamic executor pools may be used. As Mira – being based on HCL – is aware of the application DAG and therefore upcoming stages and tasks, the AS can register near-future demands of an application with the RM which can then start corresponding executors in advance.

As executors in dynamic pools will be cold initially, a short (system-provided) warm-up application may be executed on them, such that they connect quickly to actual application drivers. This can help to reduce tail latencies, as cold executors require significantly longer to connect than warm executors (see Section 4.2.1 for details).

4.6.4 Lookahead-scheduling benefits

Mira, being based on HCL-SP, has the ability to look ahead in the application DAG and anticipate near-term resource demands. This ability could be used to pre-acquire and/or warm-up cold executors ahead of time, such that the acquisition delay and tail latencies, that the application might experience otherwise, are reduced.

4.6.5 Scale-out performance implications

During the evaluation of Mira, a unexpected behavior was observed, namely, that reducing the number of executors an application can use, speeds up overall execution. This behavior has several causes, among them are:

- *More effective straggler mitigation.* The largest issue with stragglers is, that they may force an entire application to stall in front of a barrier. This is most likely to happen, if a straggler occurs within the last wave of tasks that is executed in front of a barrier. Since with fewer executors, the number of task waves increase while the number of tasks per wave decrease, the likelihood of a task in the last wave to become a straggler is smaller too.

- Shared data caching and JIT-cache exploitation are more effective if fewer executors are used.
- Peak communication volumes are reduced, as fewer tasks start/finish at the same time.

Based on this observation, a resource scale-out factor (RSF) α was added to Mira. α the number of executors a set of tasks (e.g. a stage) is executed on. Per default, $\alpha = 1.0$, i.e., the AS tries to acquire one executor per executable task. This is also the policy of vanilla Spark. With a lower value e.g., $\alpha = 0.5$, the AS only acquires 0.5 as many executors as executable tasks, hence α constitutes a relative resource usage limit, not an absolute one. As vanilla Spark does not support a relative resource usage limit, evaluation of α has been omitted from the main evaluation section of this chapter. As the effort to evaluate the impacts of the resource scale-out factor is ten times[13] as large as for the single-application experiments presented in the main evaluation section (Section 4.4), here, only a subset of TPC-DS queries have been evaluated for $\alpha = 0.0 \ldots 1.0$ (with otherwise identical setup and configuration parameters as in Section 4.4.3) on a small subset of TPC-DS queries have been conducted in an attempt to quantify this effect. The results of these experiments are shown in Figure 4.18.

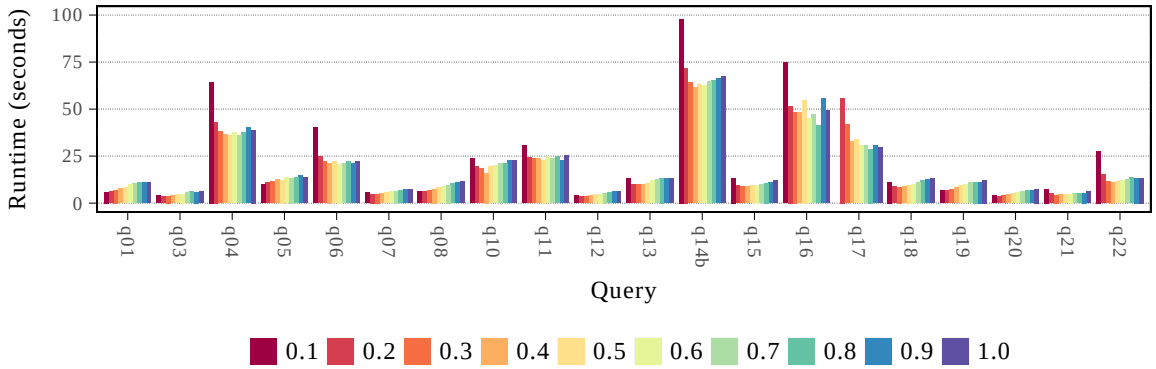


Figure 4.18: *Impact of the resource scale-out factor (RSF) on application runtime using Spark+Mira (warm). Results for Spark+Mira (cold) are similar.*

The results indicate that using the maximal number of executors possible is rarely the best choice and that a reduced set of executors can deliver better performance, while also reducing resource usage. Unfortunately, the ideal α (from a runtime perspective) varies across queries but is typically on the lower end ($\approx$0.3). However, the conducted experiments are not sufficient to draw firm conclusions. For instance, as the number of tasks frequently exceeds the number of available executors, a *natural* upper bound for parallelism is set.

[13] For each value of α a complete set of benchmarks had to be performed.

4.7 Conclusion

This chapter presented Mira, a resource manager and elastic application scheduler that enables efficient resource sharing for micro-task-based data-analytics frameworks at small time-scales. The evaluation showed a significant performance and efficiency improvements over existing systems, accelerating analytics workloads by up to 3.1× in a shared environment while reducing executor idle time by up to 3.9× to 3.5%.

Mira is an orthogonal approach to HCL-SP to increase resource utilization efficiency that works across multiple applications instead of within individual applications. However, Mira has shortcomings as well, which, along with possible solutions, have been discussed in §4.6. Furthermore, due to its reliance on micro-tasks, Mira cannot be applied to some important algorithms, namely distributed iterative-convergent machine learning algorithms, as discussed in the next chapter.

5. Tackling scheduling challenges for distributed machine learning

This chapter presents uni-tasks, a novel execution model that enables efficient, elastic and load balanced execution of iterative-convergent distributed machine learning (ML) training algorithms in shared heterogeneous environments.[1]

Uni-tasks addresses a major issue of *micro-tasks*, the standard method to implement elasticity and load balancing in distributed systems. Balancing load and scaling elastically with micro-tasks requires the usage of a large number of tasks. This can have detrimental effects on the convergence behavior of state-of-the-art distributed ML training algorithms [124]. On the other hand, distributed ML training algorithms have special properties that allow alternative approaches to tackle these scheduling issues: They iteratively converge towards a goal while being resilient to bounded errors in the computation. These properties enable uni-tasks. In contrast to micro-tasks, uni-tasks schedules small (stateful) data chunks, not tasks. This allows systems based on uni-tasks to only execute a single task per node at all times, without impairing its ability to scale elastically and balance load.

Uni-tasks has been implemented in Chicle, a distributed, elastic ML training framework and two distributed ML algorithms, Communication-efficient distributed dual Coordinate Ascent (CoCoA) and local Stochastic Gradient Descent (SGD) (a recent variant of mini-batch SGD) have been implemented. These algorithms cover the training of generalized linear models (GLMs), neural networks (NNs), deep neural networks (DNNs) and convolutional neural networks (CNNs). Uni-tasks and Chicle have been evaluated in an elastic, heterogeneous setup.

The main contributions of this chapter are summarized as follows:

(1) Uni-tasks, a novel execution model that enables fine-granular elastic and load balanced scheduling with only a single task per node. This allows efficient, elastic execution of distributed iterative-convergent ML training algorithms in shared and heterogeneous environments, without impairing convergence.

(2) Chicle, a prototypical implementation of a distributed ML training framework based on the uni-tasks execution model.

[1] The work presented in this chapter is based on the publication "Elastic CoCoA: Scaling In to Improve Convergence" [106].

(3) Novel optimizations techniques for the CoCoA algorithm that are enabled by uni-tasks and can accelerate training while, at the same time, reduce the resource consumption.

This chapter is structured as follows: The problem of elastic and load-balanced scheduling is introduced in Section 5.1, followed by a brief overview over the workings and important properties of distributed ML applications (Section 5.2). Section 5.2 elaborates thereon and provides background information on the scheduling challenges that distributed ML applications face, as well as how they are currently addressed.

Sections 5.3 and 5.4 introduce uni-tasks and the implementation thereof in Chicle. The following three sections elaborate on the concepts and implementations used to address the three scheduling challenges that are considered in this chapter: Load balancing in heterogeneous clusters (Section 5.5), elasticity (Section 5.6) and straggler mitigation (Section 5.7). Each section also presents an evaluation.

Subsequently, a comparison of Chicle with two state-of-the-art distributed ML training frameworks is presented (Section 5.8) before two CoCoA-specific optimizations enabled by uni-tasks are presented (Section 5.9). This chapter is concluded with an overview of related work (Section 5.10) and a discussion (Section 5.11).

Section A.3 accompanies this chapter with additional information. Test applications used to evaluate uni-tasks and Chicle are described in Section A.3.1.

5.1 Introduction

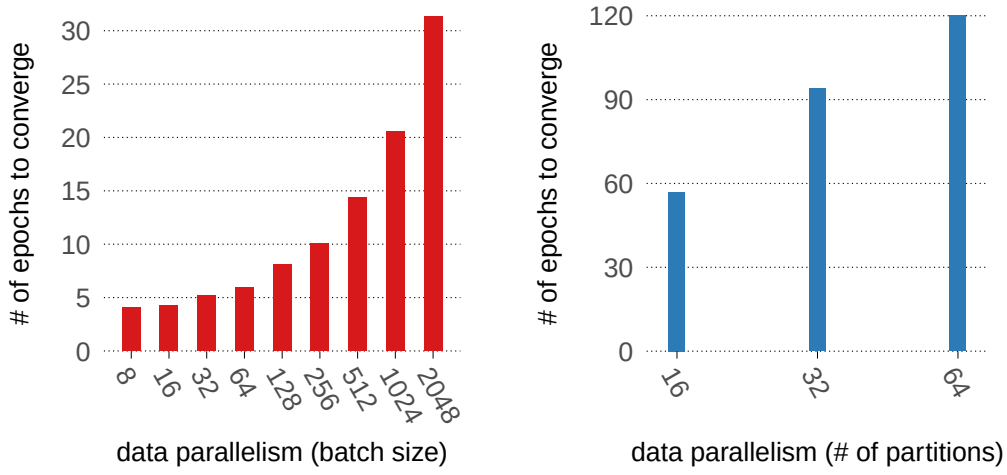
Machine learning (ML) has become one of the most important areas of research in recent years and novel services such as automatic object identification, recommendation systems, navigation systems, language translation and others have emerged from it. Many advances in ML have been fueled by ever growing collections of data that ML algorithms can be trained on and distributed training has become the standard way to do so in a reasonable amount of time.

As many other distributed applications, distributed ML training often runs on infrastructure that is shared among multiple applications and users. As shown in the previous chapter (Chapter 4), elastic execution is key to efficient resource sharing in such scenarios. In general purpose frameworks, such as Spark [18], elastic, load balanced execution is usually implemented with micro-tasks. In a micro-tasks system, elastic scaling is done by distributing a given number of tasks across more or fewer nodes and not by adjusting the number of tasks. Similarly, load balancing is done by scheduling more or fewer tasks per node. The number of tasks constitutes an upper bound for the scaling and determines the granularity with which load can be balanced, hence the number of tasks should be high (e.g., up to 2–3× as many tasks as CPUs [118]).

Due to the importance of ML and peculiarities of ML training algorithms (Section 5.2.2), several specialized distributed ML training frameworks have emerged over

the last couple of years [62, 85, 103]. Most of them, though, are rigid, i.e., do not support elasticity nor load balancing. The few that do [114, 90], also follow the basic micro-task approach. While this approach works well for generic distributed applications, it is not an ideal fit for many distributed ML training algorithms, such as Mini-batch SGD (mSGD)/glslsgd and CoCoA.

The reason for this is, that the number of tasks used during each iteration of the training process constitutes a lower bound for the level of data parallelism, i.e., the number of independently processed training samples, as each task has to process at least a single training sample per iteration. The level of data parallelism, however, affects the convergence behavior of training algorithms: The larger it is, the lower the convergence rate per processed training sample becomes and the more epochs (passes through all training samples) are needed to converge. Figures 5.1a and 5.1b depict this correlation for the mSGD and CoCoA training algorithms. An extensive study of this relationship for mSGD is presented in Shallue et al. [124].



(a) CIFAR-10 (mSGD) **(b)** Criteo (CoCoA)

Figure 5.1: *Example of the correlation between data parallelism and the number of epochs needed to converge (lower is better) for the mSGD and the CoCoA training algorithms. See Sections A.3.2.1 and A.3.2.2 for details about this experiment.*

This constitutes a conflict between *scheduling efficiency*, where more tasks allow the scheduler to better balance load and elastically scale out further, and *training efficiency*, where fewer tasks generally improve convergence rate per training sample.

For instance, in the example from Figure 5.1a, a CNN is trained using the mSGD algorithm on the CIFAR-10 dataset. In mSGD, the batch size corresponds to the level of data parallelism, i.e., the number of samples that are processed independently.

If this CNN were to be trained on a shared 16 node cluster, at least 16 tasks would be required to utilize all nodes. If each task trained on one sample per iteration, a batch size of $16 \times 1 = 16$ would be required. According to the data from Figure 5.1a, this

would require approximately four epochs to converge. Due to constant system overheads, one training sample per task and iteration would most likely incur unacceptable relative system overheads. To mitigate these overheads, multiple samples, e.g., eight, need to be processed per task and iteration. With the resulting batch size of $16 \times 8 = 128$ training samples, approximately eight epochs would be required to converge.

Assuming that one epoch takes one minute on all 16 nodes, training would finish in about eight minutes. If, however, only eight nodes, instead of the expected 16, were actually available, training with 16 tasks would take twice[2] as long, as two tasks per node need to be executed back-to-back in each iteration. However, in a shared system, the number of available nodes is generally not known in advance.

Had the user chosen to only use eight nodes (and thus eight tasks) from the beginning, the resulting smaller batch size of $8 \times 8 = 64$ would only require about 6 epochs and therefore $2 \times 6 = 12$, instead of 16 minutes. Hence, the user has two options:

(1) Speculate that all 16 nodes will be available during training, accept the increased number of epochs to converge, and risk a prolonged training in case they're not.

(2) Assume that only eight nodes will be available during training, benefit from a lower number of epochs to converge, but risk not exploiting all available resources, if all 16 nodes will be available after all.

This example highlights the main issue of micro-tasks for elastic distributed ML training: The number of tasks is generally only optimal, if all of them can be executed in parallel and sub-optimal otherwise. When considering the additional tasks needed to balance load in a heterogeneous cluster, the problem is aggravated further. Figure 5.1b shows a similar behavior for the CoCoA algorithm, were a higher level of data parallelism also leads to a larger number of epochs needed to converge.

5.2 Background

This section complements Chapter 2 with additional background information, specific to ML, and provides information on the general workings of distributed iterative-convergent ML training algorithms as well as relevant properties that are used to address the scheduling challenges described in Section 2.2.

5.2.1 General workings of distributed ML training algorithms

ML training algorithms identify correlations between input data (training samples) and expected output data (labels) and gradually adjust the parameters of an ML model in an iterative fashion to reflect these correlations ever more closely, i.e., during the training process, an ML model iteratively converges towards a state where it can predict the

[2] Ignoring scaling overheads.

correct outputs for a given input data set with increasing accuracy. While the details of this training process vary across concrete training algorithms, the general **iterative-convergent** pattern remains the same.

Next, this training process is formalized: A randomly initialized model $\mathbf{m}$ is iteratively refined during the training process on a training dataset D such that $\mathbf{m}$ converges towards a state that minimizes (or maximizes) an objective function (e.g. mean square error (MSE)). Here D contains a plurality of training samples and label pairs, e.g., an image of a cat and the label "cat", which represent the *ground truth*.

During each iteration i, an updated model $\mathbf{m}^{(i)}$ is computed on a randomly chosen subset $\widehat{D} \subseteq D$:

$$\mathbf{m}^{(i)} = \mathbf{m}^{(i-1)} + f_{\Delta}(\mathbf{m}^{(i-1)}, \widehat{D}) \tag{5.1}$$

The update function f_{Δ} is computed in a data parallel manner across K nodes by splitting up $\widehat{D}$ into K disjoint partitions $\widehat{D}_k \subseteq \widehat{D}$:

$$f_{\Delta}(\mathbf{m}^{(i-1)}, \widehat{D}) = \sum_{k=1}^{K} \Big(\frac{|\widehat{D}_k|}{|\widehat{D}|} \times f_{\Delta,k}(\mathbf{m}^{(i-1)}, \widehat{D}_k) \Big) \tag{5.2}$$

Each $f_{\Delta,k}$ is weighted by the fraction $\frac{|\widehat{D}_k|}{|\widehat{D}|}$ of the training samples it processed in iteration i to account for imbalances in the size $|\widehat{D}_k|$ of each partition $\widehat{D}_k$. In case all partitions have the same size, this degrades to $\frac{1}{K}$.

The computation of f_{Δ} is auto-correcting in face of bounded errors, an important property discussed further in Section 5.2.2.

The general structure of the algorithms that are considered here is depicted in Figure 5.2: During each iteration i, K tasks each process $H \times L$ training samples based on the model of the previous iteration $\mathbf{m}^{(i-1)}$ to compute $f_{\Delta,k}$, whereas H sets of L independent samples are processed sequentially by each task. L samples are processed in parallel to take advantage of local hardware-parallelism. After each set of L samples have been processed, a local model update is performed, such that learning on subsequent samples within the same iteration can exploit knowledge gained from previous samples of the same iteration and task. Afterwards, all tasks combine their results (Equation 5.2) to update a global model (Equation 5.1), which forms the basis of the next iteration. Values for L and H depend on the specific algorithm and configuration.

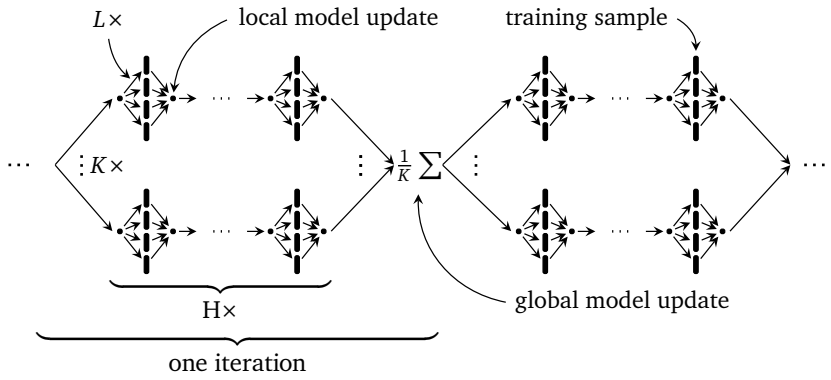


Figure 5.2: *General structure of distributed ML training algorithms considered in this chapter.*

5.2.2 Important properties of distributed ML training algorithms

The distributed ML training algorithms considered here have five important properties that can be exploited by ML training frameworks to optimize the training process. These properties are:

(1) The model is trained in an **iterative** fashion, i.e., the update function f_Δ is computed repeatedly, generally on the same number of training samples in each iteration. This property is exploited for load balancing (Section 5.5) and elasticity (Section 5.6).

(2) The model gradually **converges** towards an optimum. For generalized linear models (GLMs) this optimum is always global, while no such guarantee can be made for neural networks (NNs). Here, the algorithm may converge towards a local optimum. The convergence rate generally changes during the training process, starting at a high rate and approaching zero towards the end. This property is exploited for auto-scale-in (Section 5.9.1).

(3) The computation of f_Δ is **auto-correcting**, i.e., it can tolerate bounded errors, should they occur, and correct them in subsequent iterations. This property is exploited for straggler mitigation (Section 5.7).

(4) The training process itself is **stochastic**, i.e., training samples can be processed in any order and a random sample selection typically increases the convergence rate. This property is a exploited for load balancing (Section 5.5), elasticity (Section 5.6) and straggler mitigation (Section 5.7).

(5) The computation of f_Δ by each task is **interruptible** at any time, i.e., before all training samples of the current iteration have been processed. After an interruption, f_Δ represents a valid update for the so-far processed training samples. This property is exploited for straggler mitigation (Section 5.7).

These properties are exploited by many distributed ML training frameworks to varying degrees and can be used for many different purposes, such as automatic hyper-parameter[3] tuning. Here, however, the focus is on how these properties can be exploited to improve scheduling, which will be elaborated on in the next section.

Beyond these five properties, there is one additional important aspect to consider when devising scheduling techniques for distributed ML algorithms, namely that their **convergence rate per training sample and the ability of the resulting model to generalize to previously unseen data degrades with an increasing level of data parallelism**, as already indicated in Section 5.1. Generalization refers to the ability of a model to make predictions for previously unseen data. While no formal explanation for this correlation can be given here, intuitively, it exists because of the stochastic nature of ML training algorithms where the ideal is to update the model after each sample. The more samples are processed independently (in parallel), the further the algorithm diverges from this ideal and the more contradictory the updates from each sample can become.

Two recent works examine this correlation in more detail for the widely-used mSGD algorithm: Keskar et al. [71] focus on the issue of the ability of a model to generalize from the training data to data encountered during the usage of the model. Shallue et al. [124] extensively study this aspect w.r.t. the convergence rate per processed training sample.

5.3 The *uni-tasks* execution model

This section introduces uni-tasks, a novel execution model that enables efficient, elastic and load balanced execution of iterative-convergent distributed ML training algorithms in shared heterogeneous environments.

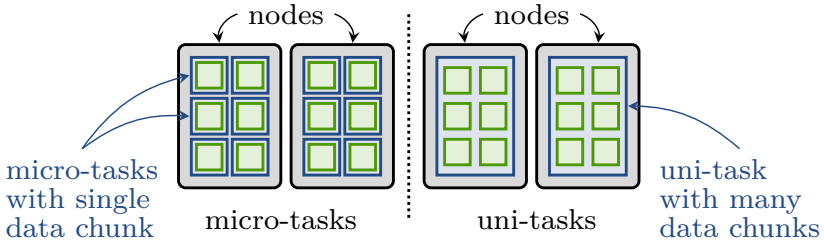


Figure 5.3: *Conceptual difference between micro-tasks (left) and uni-tasks (right). Tasks are represented as blue, and data chunks as green boxes.*

Figure 5.3 depicts conceptual differences between uni-tasks and micro-tasks which are explained in the following. In uni-tasks, only a single execution context (henceforth referred to as "task") is executed per node. A task can use multiple threads to exploit hardware-level parallelism. The schedulable unit of work is a data chunk. A data chunk

[3]The term *hyper-parameter* refers to a configuration parameter of an ML training algorithms.

contains a set of data items (training samples) that can be processed independently by the application-specific algorithm that is executed by the task. One or more data chunks can be assigned to each task. Each task has full, random access to all data items across all data chunks that were assigned to it.

While scheduling data chunks may appear similar to scheduling tasks and having a single task per node may appear similar to how distributed memory parallel programming concepts are used (e.g., for Message Passing Interface (MPI) applications), there are important differences to both.

- When scheduling tasks to balance load and to scale elastically, the number of tasks determines the scheduling granularity and therefore needs to be a multiple of the number of nodes. As explained previously, a large number of tasks has detrimental effects on the convergence behavior of distributed ML training algorithms. In uni-tasks, the number data chunks, not tasks, determines the scheduling granularity and therefore needs to be a multiple of the number of nodes. As only a single task per node is used, which has full, random access to all data chunks on the node, the number of nodes, not the much larger number of data chunks, constitutes a lower bound for the level of data parallelism of the application. In consequence, the latter is a multiple smaller in uni-tasks than in micro-tasks. Systems that fall in this category are Spark [18] and Litz [114].
- In other distributed memory parallel programming concepts that use a single task per node (e.g., as many MPI applications), the level of data parallelism can be as low as with uni-tasks. In contrast to uni-tasks, however, no load balancing or elastic execution is defined by the concept. Systems that fall in this category are Snap ML [103], TensorFlow [62] and PyTorch [85]

Uni-tasks combines the scheduling flexibility of micro-tasks with the low lower bound for the level of data parallelism of MPI applications.

5.3.1 Assumptions

Uni-tasks is based on the following assumptions:

- An increased level of data-parallelism impairs convergence rate or maximal test accuracy. This justifies the uni-tasks approach.
- Training algorithms are iterative. This is required as uni-tasks implementation is expected to move data chunks only in-between iterations.
- Training samples can be processed independently, in any order and combination. Furthermore, the number of training samples can vary across tasks without impairing maximal test accuracy and convergence rate. This ensures that data chunks can freely moved across tasks.

- Task runtime increases with the number of training samples processed during an iteration. This ensures that load can be balanced by adjusting the number of data chunks on each task.
- The level of data parallelism can be changed during training without impairing maximal test accuracy and convergence rate. This allows the system to add and remove tasks during training.
- Local solver algorithms do not access training data in-between iterations and do not retain state that depends on individual training samples and is not kept in data chunks. This ensures that data chunks can be moved between tasks in-between iterations.
- Local solver algorithms can be interrupted frequently during each iteration. At each interruption point, a valid result (model update) is available. This is used to mitigate stragglers via task preemption. The more often a task can be interrupted, the smaller the runtime variations across all tasks can be.

5.3.2 Task contract

In order to provide simple rules to application developers, uni-tasks establishes a simple two-clause **task contract** between the scheduler (system) and the application (task):

(1) During an iteration, a task has full, random access to all training samples across all data chunks that the scheduler assigned to it. The scheduler may not add or remove any data chunks.

(2) In-between iterations, i.e., when tasks synchronize with each other, the scheduler may add or remove data chunks to and from tasks. Tasks cannot access training samples during this period. The scheduler notifies the task about any changes in their data chunk assignment.

This contract ensures that training algorithms executed by tasks can treat all training samples from all data chunks as if they came from a single set while allowing the scheduler to shift load between tasks frequently and at well-defined points in time.

5.3.3 Scheduling in uni-tasks

As there is only a single task per node in uni-tasks, task scheduling cannot be used to shift load between nodes. Instead, the scheduler assigns one or more *data chunks* to each task. A data chunk is a fixed-sized container for training (input) data, and for state that depends on training data samples. The number and size of data chunks determines the scheduling granularity and should therefore be at least an order of magnitude larger than the expected number of nodes used during training. Similar recommendations for the

number of tasks exist for micro-tasks [118, 37, 36]. In contrast to micro-tasks, however, a large number of small data chunks does not incur significant runtime overheads (see Section 5.3.4 for details) nor does it impair convergence.

The benefits of uni-tasks over the micro-task concept can only be realized if all three scheduling challenges (as laid out in Section 2.2.4) are addressed. Sections 5.5 through 5.7 detail and evaluate the concepts and implementations used to address these challenges.

5.3.4 Uni-tasks overheads

In contrast to micro-tasks, the number and size of data chunks does not correlate to system runtime overheads. This is due to the way data chunks can be handled by tasks: Each task has full random access to all data chunks in its memory space. A data chunk appears as a fixed sized, contiguous memory region that contains a dense or sparse training sample matrix, hence the localization and access of a training sample in a task's memory only requires the de-referencing of the data chunk's base pointer and an access into the contained matrix.

Data chunks can incur memory overheads as they can only store complete training samples. If the combined size of all training samples stored in a data chunk does not match the capacity of the data chunk, allocated, but unused space remains. In case where the number of training samples N is in the tens or hundreds, this overhead is generally very small, in the order of $1/N$. If, however, only few training samples fit in a data chunk, it can become significant. In that case, increasing the data chunk size may alleviate this issue.

5.4 Chicle design and Implementation

This section introduces Chicle[4], a distributed, elastic ML training framework. In contrast to other frameworks, such as TensorFlow [62], PyTorch [85] and Snap ML [103], Chicle combines the ability to elastically scale execution, balance load and mitigate stragglers in homogeneous as well as heterogeneous environments without compromising convergence behavior that micro-task-based systems [114] suffer from.

The main objective of Chicle is to demonstrate the viability and benefits of the uni-tasks concept in elastic, heterogeneous scenarios while also demonstrating that uni-tasks does not result in significant drawbacks for applications in rigid, homogeneous scenarios. The purpose of Chicle is therefore to serve as a proof of concept for uni-tasks, rather than a production-ready distributed ML training framework. For this reason the implementation of advanced optimization methods, such as online hyper-parameter tuning [86, 102], that can be found in major ML training frameworks, have been omitted.

[4] Chicle is the Mexican-Spanish word for latex from the sapodilla tree that is used as basis for chewing gum and a reference to Chicle's ability to elastically *stretch* (scale) computation.

Furthermore, the decision to implement Chicle from scratch, instead of adapting an existing framework to uni-tasks, was made due to the expected complications when modifying a central concept (such as the execution model) of a highly optimized (and therefore presumably less adaptable) existing ML training framework. In fact, initially, Spark was investigated w.r.t. the possibility of implementing uni-tasks. However, serious limitations were identified, with the low data (de-)serialization performance being the most severe one. As chunks may be moved across nodes frequently, the overhead for doing so needs to be low. It was concluded that this is not possible within the Spark framework, and that an implementation from scratch is warranted.

5.4.1 Assumptions

This section complements the concept-specific assumptions in Section 5.3.1 with Chicle-specific ones.

- Data chunks are stored in memory in a way that does not require serialization (but may require deserialization) before being transferred between tasks. This enables transfer via RDMA read operations without prior notification of the remote task. If this assumption does not hold, overhead for data chunk movement during load balancing and elastic scaling actions increases.
- Node performance changes slowly (excluding stragglers), over the course of at least tens of iterations. This ensures that the load balancing algorithm can learn task runtimes to balance load.

5.4.2 Overview

The high level architecture of Chicle is depicted in Figure 5.4. Chicle is implemented as a synchronous driver/worker design with a central driver and multiple workers. Communication uses an RDMA-based RPC framework. Chicle is implemented in ≈7k lines of C++ code.

Both, driver and workers, consist of multiple modules which are tied together by a local event bus for communication and synchronization. The driver executes the **trainer** module, which, in tandem with multiple policy modules, is responsible for the coordination of the training process. Scheduling-related algorithms, e.g., load balancing and task preemption, are implemented as policies. Applications can also make use of the policy framework to implement custom non-scheduling related optimizations (see Section 5.9).

Worker execute **solver** modules, where application-specific algorithms, such as SGD or stochastic coordinate descent (SCD), are implemented (examples of solver modules can be found in Section A.3.1). As per uni-tasks, only a single (multi-threaded) worker (task) is executed per node. Solvers are controlled by the trainer and policy modules in the driver (examples of trainer modules can be found in Section A.3.1). Solvers report

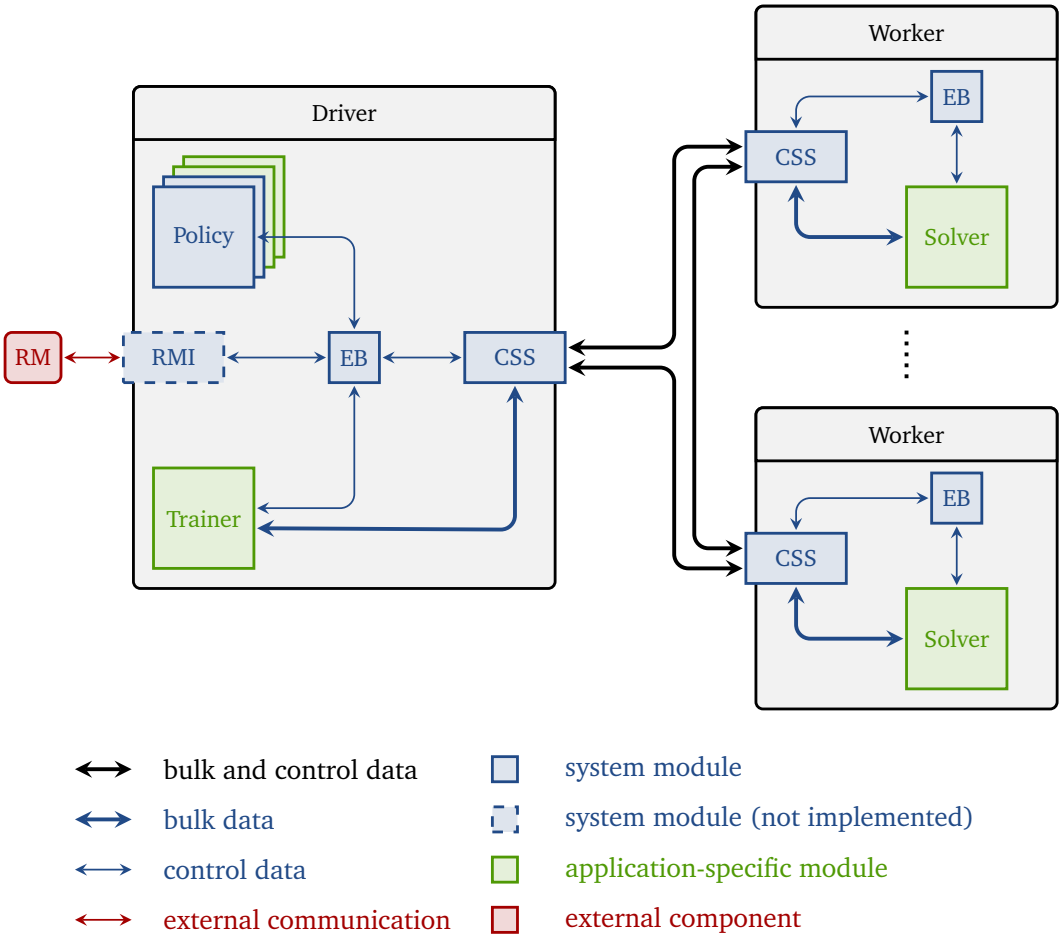


Figure 5.4: *High-level architecture of Chicle with driver and worker components, as well as their respective trainer, policy, event bus (EB), communication subsystem (CSS) and solver modules. The resource manager (RM) is connected to the driver via a RM interface (RMI).*

metrics to the driver, such as runtimes, number of processed samples, etc., to them which are used to make scheduling decisions. Furthermore, solvers periodically (typically at the end of each iteration) synchronize model updates with each other via the trainer.

Chicle applications consist of, and need to implement, a trainer and solver module as well as optionally, application-specific policy modules. Applications can also make use of policy modules that Chicle provides natively. Two examples of Chicle applications are presented in Section A.3.1.

In the remainder of this section, each system module as well as the communication subsystem and data chunk format of Chicle is described. Application-specific trainer and solver modules are described in Section A.3.1.

5.4.3 Communication facilities

This section describes the implementation of the event relay and communication facilities of Chicle. While important for building a functional system, their specific implementation is independent of the uni-tasks concept.

5.4.3.1 Event Bus (EB)

A means to relay events between various system modules is a basic necessity for any complex (distributed) system. Chicle implements a simple asynchronous event bus based on the Boost Signals2 library [119] that serves as central hub for global and local events to meet this necessity. It allows modules to subscribe to specific event types and emit events. Important event types (e.g., start iteration, iteration finished) as well as their senders and receivers are listed in Table A.20 and Figure 5.4 additionally shows the global communication paths in Chicle.

5.4.3.2 Communication Subsystem (CSS)

The efficiency of its communication subsystem substantially contributes to the performance of a distributed system. For distributed ML training, that requires frequent synchronization between multiple distributed tasks and communication of large amounts of data, it is vital. For instance, for the KDDA dataset (see Table A.18), K × 154MiB of model update data (f_Δ) have to be communicated per iteration between workers and the trainer and vice versa.

Therefore, a communication subsystem based on remote direct memory access (RDMA) has been implemented to enable zero-copy data transfers with minimal CPU overhead. RDMA-based communication is also available in other distributed ML training frameworks, such as TensorFlow [62], Snap ML [103] and PyTorch [85].

Chicle's communication subsystem is based on the OFED Verbs API as this allows for the greatest implementation flexibility and performance. Basic communication primitives, such as one-to-one, one-to-many and many-to-one communication patterns have been implemented, the latter two via wrappers based on one-to-one primitives. Moreover, Chicle uses one-sided RDMA read primitives for bulk data transfer (model updates, data chunks) to avoid unnecessary memory copies. Remote procedure calls (RPCs), to relay events from the event bus across nodes are transmitted using two-sided RDMA send and receive primitives. For each peer, an instance of a communication endpoint is constructed.

An overview of the most important methods provided by Chicle's communication subsystem is given in Table A.21.

5.4.4 In-memory data chunk format

Chicle provides the `Chunk` class to store and transmit mutable data across nodes in fixed-sized contiguous memory chunks. Applications can utilize the `Chunk` class to store and transmit training data as well as state that corresponds to training samples. Applications have to derive a specialized class from the provided generic `Chunk` class. Each derived chunk class needs to implement the methods listed in Table 5.1 but can add additional methods, e.g., to access individual training samples, as needed.

Method	Description
`int idx()`	Returns the global index of this data chunk.
`int worker_id()`	Returns the id of the worker this chunk currently resides on. This is mostly used by the trainer which retains *empty* chunks (i.e., without allocating any memory for training samples) locally, to keep track of their location.
`struct ibv_mr* mr()`	Returns an IB Verbs memory region data structure that contains necessary information for remote one-sided access to this chunk.
`char* data()`	Returns a pointer to the raw data that is backing this data chunk.
`size_t size()`	Returns the size of the memory that backs this data chunk.
`void restore()`	Restores (de-serializes) this data chunk after is has been read from a remote node.
`int num_samples()`	Returns the number of training samples contained in this data chunk.

Table 5.1: *API of the generic `Chunk` class.*

The uni-tasks concept assumes that data chunks can be moved and processed independently of each other, i.e., training samples must not span multiple data chunks. Furthermore, Chicle's communication subsystem requires that data chunks can be transferred using one-sided RDMA read primitives and must be transferable without serialization, as the source task is not notified of the transfer ahead of time. This also limits the data structures that can be stored inside a chunk object to ones that do not rely on the validity of pointers after a transfer (i.e., if pointers are used, they need to be restorable by the receiving task). Fortunately, most training data is stored in form of dense or sparse vectors, matrices or simply byte arrays, which are not affected by these restrictions.

De-serialization steps (via the `restore` method) are allowed (e.g. to restore pointers), as long as they do not modify the data in a way that would require serialization for further transfers. Chicle assumes that the in-memory layout of `Chunk` objects is identical on all

nodes, which prohibits the use of mixed-endianess clusters as well as (to some degree) different compilers or compiler versions[5]. Examples of application-specific chunk classes can be found in Section A.3.1.1 and Section A.3.1.2.

5.4.5 System policies

System policy modules are a vital part of Chicle. All scheduling functionality is implemented as system policies. Additionally, various support functions, such as the configuration of communication paths, are implemented as policy modules.

Policy modules are executed in separate threads on the driver and multiple policy modules can be executed at the same time. They can coordinate with each other, the trainer and solver via the event and communication subsystem of Chicle. Furthermore, they need to coordinate with the trainer as data chunk reassignment actions that can be initiated by policies must be completed before the next iteration starts. Table 5.2 lists all implemented system policies.

Policy	Description
Rebalance	This policy balances workload in heterogeneous clusters by shifting data chunks from slow nodes to fast nodes. See Section 5.5.1 for details.
Elasticity	This policy redistributes chunks upon addition or removal of nodes. See Section 5.6.1 for details.
Preemption	This policy is responsible for straggler mitigation and preempts straggling tasks. See Section 5.7.1 for details.
Initial chunk assignment	This policy randomly assigns data chunks to workers after they are added such that the work load (number of chunks) is balanced.
Statistics	This policy module collects and reports metrics during each iteration necessary to evaluate Chicle.

Table 5.2: *System policies of Chicle. The first three policies implement scheduling functions while the rest implements support functionality.*

In addition to the system policies, additional application-specific policies have been implemented. They are presented and evaluated in Section 5.9.

[5]The in-memory layout of data structures is not fully defined by the C/C++ ISO standard and may differ across compilers or compiler versions.

5.5 Load balancing

Load balancing ensures that the load on each node corresponds to the performance of said node. Faster nodes can perform more work than slower nodes in the same amount of time.

Load balancing in uni-tasks works by shifting data chunks from workers (henceforth referred to as tasks) on slower nodes to tasks on faster nodes, until the relative number of data chunks on each node corresponds to their relative performance to each other and task runtimes per iteration align.

The task runtime per iteration does not only depend on the number of data chunks assigned to it but on the number of training samples processed. Therefore, in addition to shifting data chunks, tasks also need to know how many training samples they are supposed to process in each iteration, e.g. one sample per data chunk. Uni-tasks does not prescribe a specific policy but assumes a positive correlation between the number of training samples per task and the number of training samples processed by that task per iteration.

As result of the task contract (Section 5.3.2), the system is not allowed to add or remove data chunks to and from tasks once an iteration has started. This prevents the scheduler from reactively balancing load as a micro-task system could do. Instead, the scheduler has to pro-actively balance load before an iteration starts. Uni-tasks exploits the iterative nature of ML training algorithms. In each iteration, the same function is executed on the same (amount) of data on each task. Assuming a constant or only slowly (over tens of iterations) changing node performance, task runtimes are relatively constant. This enables the scheduler to predict task runtimes and to correlate them with the workload (number of data chunks or training samples) of each task. Based on this information, the scheduler can estimate task runtimes for the next iteration as well as the impact of adding or removing a chunk from a task. Chunks can be gradually shifted from tasks on slow nodes to those on fast nodes, until task runtimes are balanced across all nodes. Figure 5.5 shows an example of this gradual process to balance runtimes of tasks on differently fast nodes.

This approach can balance load when node performance is static, e.g., caused by different generations of hardware, as well as when node performance changes infrequently or slowly, e.g., due to long-running background applications in shared clusters. It cannot mitigate the impact of stragglers, as those affect the perceived node performance intermittently and unpredictably.

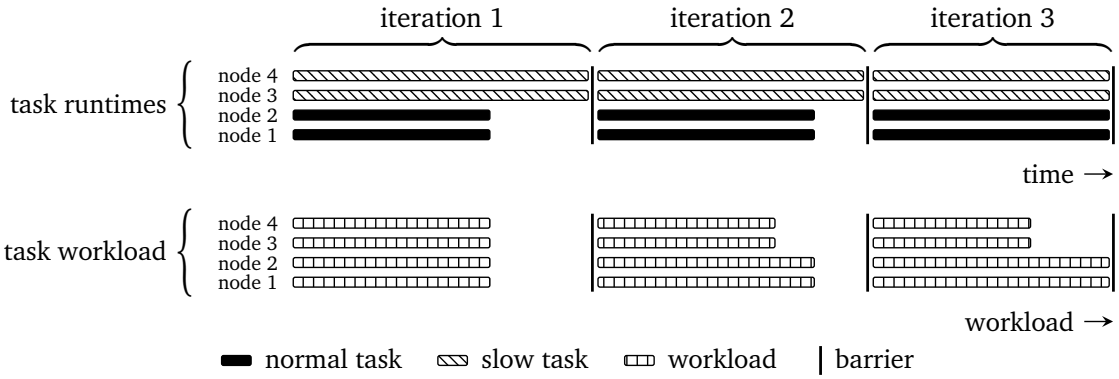


Figure 5.5: *Conceptual depiction of the workload rebalancing process using hardware-heterogeneity-aware load-balancing with uni-tasks. This simplified depiction assumes zero data transfer times.*

The basic idea is similar to HCL's initial approach to use task runtime predictions to balance load in heterogeneous clusters, without the use of many small tasks (Chapter 3). An important difference is, however, that HCL cannot make any assumptions about the iterativeness of an application, whereas systems based on uni-tasks can rely on this property of ML training algorithms. This significantly simplifies reliable task runtime predictions.

5.5.1 Load-balancing in Chicle: Rebalance policy

The rebalance policy implements the hardware-heterogeneity-aware load-balancing strategy described above. It uses task runtime predictions to estimate future runtimes and moves chunks based on those predictions across K nodes. Therefore, task t_k on node n_k with $0 \leq k < K$, reports the runtime τ'_k of the last iteration of task t_k. Furthermore, it reports the fraction of planned samples it processed (σ_k) during the last iteration. This information is necessary, as a task may have been preempted, in which case τ'_k refers to fewer samples than planned. This needs to be considered when predicting task runtimes. Using the aforementioned information, it computes the normalized task runtime during the last iteration τ_k.

$$\tau_k = \frac{\tau'_k}{\sigma_k} \tag{5.3}$$

As task runtimes are not steady but vary slightly in-between iterations, a moving median $\overline{\tau_k}$ over all τ_k of the last I iterations[6] is computed. $\overline{\tau_k}$ represents the fundamental task runtime.

All tasks are then sorted in ascending order of $\overline{\tau_k}$. Let t_f and t_s be the fastest and slowest tasks respectively. Further, let c be a randomly selected chunk of t_s with $|c|$ training samples. Let $|t_f|$ and $|t_s|$ be the total number of training samples on tasks t_f and

[6]The moving median is computed across I iterations only if I iterations have already passed and fewer otherwise.

t_s before the chunk rebalancing. Then, $\Delta\tau_f$ and $\Delta\tau_s$ is the predicted runtime change for either task, after chunk c is moved from t_s to t_f.

$$\Delta\tau_f = \frac{|c|}{|t_f|} \times \overline{\tau_f} \tag{5.4}$$

$$\Delta\tau_s = -\frac{|c|}{|t_s|} \times \overline{\tau_s} \tag{5.5}$$

Using this information, the rebalance policy decides whether the iteration runtime can be reduced by moving chunk c, i.e., whether the runtime of the fastest task t_f is not increased beyond that of the slowest task t_s. In order to prevent cyclic chunk movements, an additional damping factor α is used. Chunk c is moved, iff Equation 5.6 is true.

$$\overline{\tau_f} + \Delta\tau_f \times \alpha < \overline{\tau_s} \tag{5.6}$$

Once a chunk has been moved, τ_f and τ_s are adjusted by $\Delta\tau_f$ and $\Delta\tau_s$ and the process is repeated, until Equation 5.6 is not true anymore or until a maximum of r_{max} chunks have been moved. r_{max} has been introduced to avoid over-correction due to misprediction of task runtimes. Listing 5.1 shows the corresponding code of the `rebalance` function.

Smaller values for α and I, as well as larger values for r_{max} allow this policy to balance load quicker, but come at the cost of potentially less stable (and thus more frequent) chunk (re-)assignments. Table 5.3 lists all parameters of the rebalance policy.

Parameter	Description
α	Damping (safety) factor that ensures that data chunks are only moved if the runtime gain is *significant*, e.g. for $\alpha = 1.1$, the runtime gain has to be more than 10% of the estimated runtime of a data chunk.
r_{max}	The maximal fraction of data chunks to be moved in each iteration.
I	The size of the sliding window to compute the moving average $\overline{\tau_k}$ over.

Table 5.3: *Parameters of the rebalance policy parameters used here.*

```
void rebalance() {
  // 'numChunks' is the total number of chunks in the dataset.
  for (int moved = 0; moved < r_max * numChunks; moved++) {
    // 'workers' is a list with all currently active workers.  Sort this list in ascending
    // order of their median runtime τ_k.
    sort(workers);

    Worker* fastest = workers.first; // worker with the shortest median task runtime
    Worker* slowest = workers.last;  // worker with the longest median task runtime

    // Determine runtime for a chunk that is to be moved from the 'slowest' to the
    // 'fastest' worker.
    Chunk* chunk = slowest->chunks.first;

    float fastestChunkRuntime = chunk->numSamples / fastest->numTotalSamples *
                                fastest->medianTaskRuntime;
    float slowestChunkRuntime = chunk->numSamples / slowest->numTotalSamples *
                                slowest->medianTaskRuntime;

    // Make sure the 'fastest' task does not become slower than the currently 'slowest'
    // task, if the chunk would be moved.  Otherwise, moving the chunk does not reduce
    // iteration runtime.
    if (!(fastest->medianTaskRuntime + (fastestChunkRuntime * α) <
          slowest->medianTaskRuntime))
        break;

    // Move 'chunk' from 'slowest' to 'fastest' worker.  Chunk data movement is performed
    // in the background, hence this call is non-blocking.
    moveChunk(chunk, slowest, fastest);

    // Adjust task runtime predictions based on the predicted task runtime change
    fastest->adjustRuntime(fastestChunkRuntime);
    slowest->adjustRuntime(-slowestChunkRuntime);
  }

  // wait for all chunk data movements to complete
}
```

Listing 5.1: *Simplified C++ code of the rebalance policy's chunk moving algorithm. Additional functions are described in Section A.3.4.3*

5.5.2 Evaluation

This section presents evaluation results of the load-balancing experiments performed on Chicle to compare uni-tasks with micro-tasks with 16, 32, 48 and 64 tasks (the number of tasks is henceforth noted in parentheses). The evaluation method and metrics described here are also used for evaluations of elasticity (Section 5.6) and straggler mitigation (Section 5.7). ML-specific terms used through the evaluation are defined in Table 2.6. Two distributed ML training algorithms have implemented on Chicle:

- Communication-efficient distributed dual Coordinate Ascent (CoCoA) [46, 115] for the training of generalized linear models (GLMs).
- Local SGD (lSGD) [109], a recent variant of Mini-batch SGD (mSGD), for the training of neural networks (NNs), deep neural networks (DNNs), convolutional neural networks (CNNs) and others.

Both algorithms and their implementations are described in Section A.3.1.

Each CoCoA experiment was repeated five times and each lSGD experiment was repeated ten times. lSGD experiments are executed more often because their generally shorter task runtimes exhibit higher relative variance than those of CoCoA. Due to the large number of tests (315 individual training runs for this section), a test time limit of 360 seconds[7] has been set, after which training is terminated.

No direct comparison between uni-tasks and another load-balancing ML training framework can be presented here, as no competitive framework with this ability is publicly available. Spark, even though it is based on a micro-task concept and allows elastic and load balanced execution, was not considered here. Its performance for CoCoA is up to one order of magnitude lower than that of Snap ML [80], to which Chicle is compared later. A set of baseline tests of the reference implementation of CoCoA on Spark confirm these findings (Section A.3.2.3). Spark requires between 6.8× and 23.0× more time per epoch than Chicle (see for details). Spark's performance for other ML training algorithms falls also short [62, 77]. This does not allow a fair comparison between uni-tasks and micro-tasks.

All experiments use the test setup and datasets described in Section A.3.3.

5.5.2.1 Evaluation metrics

The following metrics and training target values are used throughout the evaluation:

- CoCoA solves a convex optimization problem with one exact solution. Training progress is measured by the **duality-gap**. The *duality-gap* is a metric for the distance of the current solution to the exact solution and represents "an accuracy certificate and stopping criterion" [115] and also used in the original CoCoA publications [46, 115]. The smallest duality-gap, rounded to the next larger 10^{-i}, that is reached by the micro-tasks (16) case within the test time limit is used as training target.
- LSGD solves a non-convex optimization problem, hence no metric such as the duality-gap exists. Instead, the **test accuracy**, i.e., the fraction of correct predictions on a test dataset, is used as metric to measure training progress. The highest test accuracy, rounded to the nearest $\frac{1}{100}$, that is reached by all runs is selected as target.

[7] This time limit does not include the time needed to load the training data from the file system.

To measure the amount of work and time necessary for training, the following metrics are used:

- **Number of epochs to converge** (lower is better), which correspond to the total amount of work necessary to converge. One epoch represents one pass over all training samples. For micro-tasks, this metric only depends on the global batch size (lSGD) and number of equally sized partitions (CoCoA), not on the location or performance of nodes on which tasks are executed. Chicle is used to emulate a micro-tasks system by using a fixed batch size/partitioning and task count.
- **Projected schedule length** (lower is better). Schedule lengths are projected as due to the lack of publicly available, competitive micro-task (or otherwise load-balancing) ML training frameworks, no wall time comparison measurements can be produced here. The projection is based on an idealized iteration duration (in units of time) and the number of iterations needed to converge.
- **Iteration duration** (lower is better). For uni-tasks, the length of iterations is measured on the test cluster with four slowed down nodes. Iteration durations are measured for uni-tasks with and without the rebalance policy as well as in the baseline case without any slowed down nodes. For the above mentioned reasons, no comparable measurements for micro-tasks could be produced.

Wall time measurements for CoCoA include the computation of the duality-gap, as it is computed in distributed fashion, similar to how a real (production) system would do it. Wall time measurements for lSGD do not include the computation of the test accuracy, as the latter is performed on the trainer instead of distributed on workers, which does not resemble a real system. For this reason, it is not included in the time measurements. This also applies to subsequent evaluations. Duality-gap and test accuracy computation does not impact convergence per epoch.

For each dataset, the number of epochs/schedule length to reach the training target is used to compare performance of uni-tasks and micro-tasks. Training target values are noted in parentheses with the results.

5.5.2.2 Parameters

Table 5.4 lists the parameters for the rebalance policy used throughout this evaluation. Tables 5.5 and 5.6 list the parameters used for CoCoA and lSGD training algorithms.

Parameter	Value	Description
α	2	Damping (safety) factor. The selected value is high enough to filter out task runtime jitter and prevent cyclic chunk movements.

r_{max}	0.5/0.05	Maximal fraction of chunks to move in one iteration. The first value is for CoCoA, the second for lSGD. The latter value is smaller, as iterations in lSGD are much shorter than in CoCoA and therefore more sensitive to background noise.
I	10	Size of the sliding window to compute the median task runtime. The selected value is high enough to filter out task runtime jitter and to prevent cyclic chunk movements.

Table 5.4: *Used rebalance policy configuration parameters.*

Parameter	Value	Description
H	variable	H is the number of independent training sample sets to be processed by each task during an iteration (see Figure A.7). It is set to a value that corresponds to all training samples that reside on a node. A smaller value of H results in a higher convergence rate per epoch but requires more iterations (rounds of communication) to converge. The optimal number of H depends on the system (compute vs. communication speed), dataset (number of features vs. number of samples) as well as the training stage (similar to Section 5.9.1). The used setting has been chosen for simplicity reasons and works for all datasets.
λ	0.01	Regularization parameter (similar to the learning rate in mSGD/lSGD). If λ is chosen too small (big), the model can over-fit (under-fit). A smaller value for λ also leads to a slower convergence of the duality-gap per epoch. The optimal value is dataset dependent. A single value has been chosen for simplicity reasons and works for all datasets.
chunk size	2MiB	Chicle chunk size. This value has been chosen as it provides a sufficiently fine scheduling granularity.

Table 5.5: *Parameters for CoCoA and the SCD solver used in this evaluation.*

Parameter	Value	Description
L	8	L is the number of independent training samples per task and iteration (see Figure 5.2 for details).
H	16	H is the average number of independent training samples sets that are processed sequentially by each task per iteration.
Global batch size	$K \times L \times H$	Global batch size for K tasks.
Learning rate	0.002	Learning rate, normalized for a batch size of 128. In line with best practice [124, 81], the learning rate is scaled by $\times\sqrt{n}$ when changing the batch size by $\times n$.
Momentum	0.9	Stochastic Gradient Descent (SGD) momentum parameter.
Chunk size	256KiB	Chicle chunk size.

Table 5.6: *Parameters for lSGD used in this evaluation.*

For lSGD, a limited set (≈2 – 8) of values for each parameter have been tested in a homogeneous scenario on 16 nodes using 16 tasks. The selected values have provided the best performance in terms of number of epochs to converge and maximal test accuracy. As no comprehensive parameter sweep has been performed, no claim of optimality w.r.t. the selected parameters is made.

5.5.2.3 Experiments

Two types of load-balancing experiments are performed:

(1) **Assumed node performance to project schedule length:** The rebalance policy is configured not to measure task runtimes on each node to determine relative node performance, but to assume a configured relative node performance. This is done to allow the projection of schedules for uni-tasks and micro-tasks and compute their total length in order to estimate *time* differences. As no other (micro-task based) load-balancing distributed ML framework exists, time differences cannot be measured directly.

(2) **Automatically determine node performance and balance load:** The rebalance policy is configured to measure task runtimes on each node to determine relative node performance. This is done to show that the rebalance policy can automatically determine node performance, balance load and reduce iteration duration.

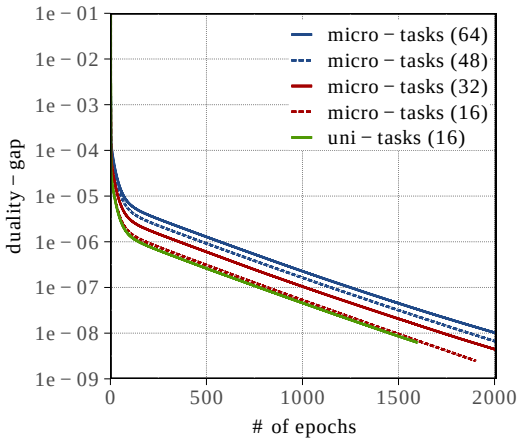
5.5.2.4 Experiment 1: Assumed node performance to project schedule length

In this experiment, the number of epochs to converge is measured for uni-tasks as well as for all four micro-task scenarios. For each scenario, schedule lengths (in units of time) are projected and compared. Furthermore, it shows the impact of uni-tasks and micro-tasks on the convergence behavior.

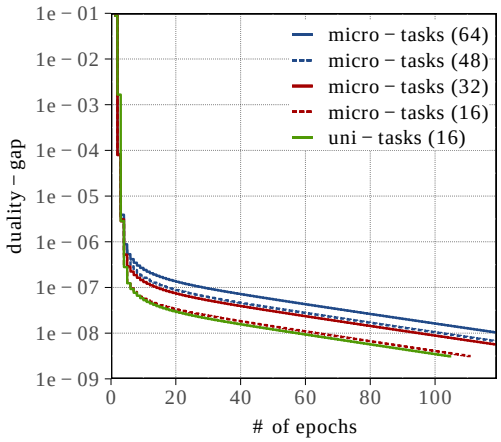
(1) **Uni-tasks scenario:** The rebalance policy uses a predefined (not measured) *node performance profile* with eight fast and eight slow nodes. Slow nodes require 1.5× as long to execute a task than fast nodes. The rebalance policy balances load according to this profile, such that fast nodes get ≈1.5× as many data chunks as slow nodes and tasks on fast nodes process ≈1.5× as many training samples per iteration than tasks on slow nodes. The total number of training samples processed during each iteration remains constant.

(2) **Micro-tasks scenarios:** Chicle is executed without the rebalance policy enabled and with 16, 32, 48 and 64 tasks to emulate a micro-tasks-based system. All tasks process the same number of training samples per iteration. The number of epochs needed to converge is independent of the quantity and speed of the used nodes.

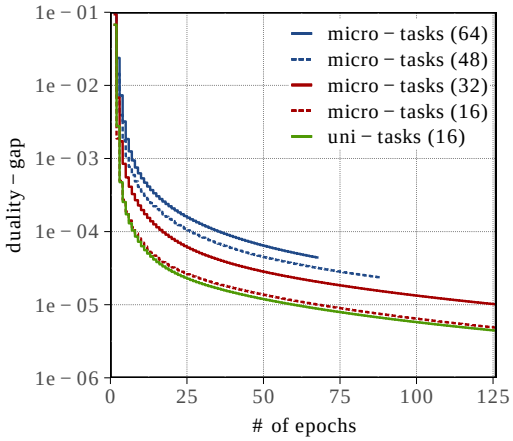
Epochs to converge. Figure 5.6 shows convergence plots for the CoCoA and lSGD algorithms. Tables 5.7a and 5.7b list the absolute and relative differences of the number of epochs needed to converge between uni-tasks and micro-tasks. Additionally, Table 5.7 lists the maximal achieved test accuracy for lSGD, which typically peaks or levels off within the test time limit. For CoCoA, the duality-gap does not level off within the test time limit, hence no minimal value can be provided.



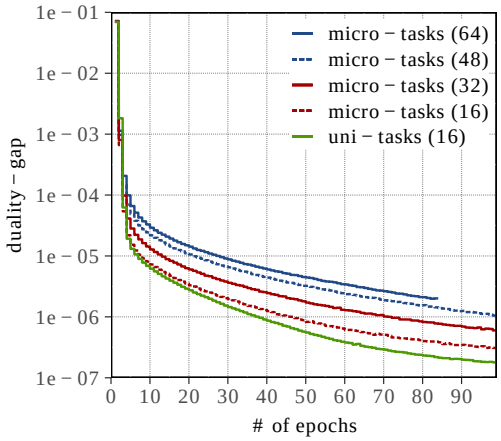
(a) Higgs (CoCoA)



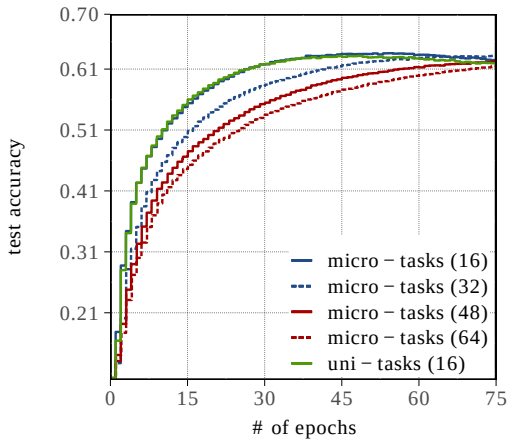
(b) Criteo (CoCoA)



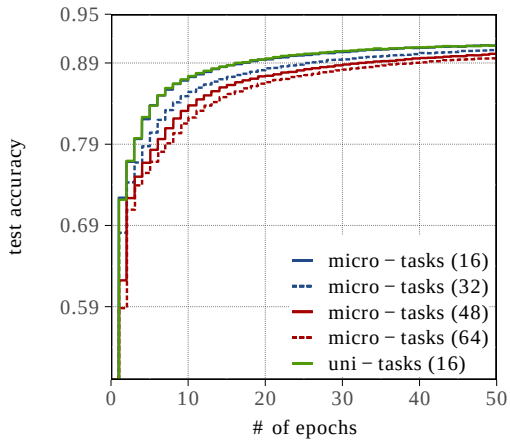
(c) KDDA (CoCoA)



(d) Webspam (CoCoA)



(e) CIFAR-10 (lSGD)



(f) Fashion-MNIST (lSGD)

Figure 5.6: *Convergence of the duality-gap (lower is better) and test accuracy (higher is better)* ***over epochs*** *in a heterogeneous scenario. This plot shows the impact of the number of tasks on the convergence behavior of micro-tasks compared to uni-tasks with the given node performance profile. The number of tasks is given in parentheses.*

Some plot lines end in Figure 5.6 earlier than others. The reason for this is that the number of epochs that are possible within the test time limit varies. For instance, uni-tasks was executed assuming the node performance profile provided above, which does not correspond to the actual node performance of the test cluster, which slowed down training and thus allowed for fewer epochs within the test time limit. The length of plot lines does not allow drawing conclusions about the training *time*, since the test is not designed to measure time.

CoCoA. Figures 5.6a through 5.6d show the convergence of the duality-gap for CoCoA. In particular, it shows that with an increasing number of micro-tasks, the number of epochs needed to reach a certain duality-gap increases as well. The reason for this is that due to the smaller partition size and the reduced training context of each task, the ability to identify correlations locally is reduced. This is visible by the location of the *knee* (the region where the convergence plot lines flatten) on the y-axis. For fewer tasks, it occurs at lower y-axis (duality-gap) values than for more tasks. After this point, the number of tasks is irrelevant to the convergence rate, as all identify correlation information only via the global state. For CoCoA, uni-tasks require as many epochs to converge as micro-tasks (16) and fewer than any other micro-tasks scenario. This is expected, as the number of data partitions is the same for micro-tasks (16) and lower otherwise. It also shows that load balancing, as it is done in uni-tasks, does not impair the convergence rate per epoch. Table 5.7a summarizes the results.

lSGD. The convergence behavior of lSGD on uni-tasks and micro-tasks (16), shown in Figures 5.6e and 5.6f, are also similar for the two tested datasets. This is expected behavior, as both use the same global batch size and only vary in the batch partitioning across tasks. For uni-tasks and micro-tasks (16), the test accuracy for the CIFAR-10 (Figure 5.6e) dataset degrades after reaching a maximum. This general behavior is common for mSGD (which is closely related to lSGD), especially with smaller batch sizes, and has also been observed in the PyTorch reference framework (see Section 5.8.2). Mitigation strategies for this behavior exist, such as reducing the learning rate or increasing the batch size in response to a falling convergence rate. Both are considered advanced optimization methods and have not been implemented yet. Moreover, in a production setting, the training would be terminated after no increase in accuracy with a test data set is achieved within a certain amount of time, at which time the model state, for which the highest accuracy was achieved, is used as final training result.

	uni-tasks	micro-tasks			
Dataset	**16**	**16**	**32**	**48**	**64**
CIFAR-10	63.43%	<u>63.90%</u>	63.34%	62.34%	61.48%
Fashion-MNIST	<u>91.28%</u>	<u>91.28%</u>	90.66%	90.20%	89.71%

Table 5.7: *Average maximal test accuracy (higher is better) for lSGD on uni-tasks and micro-tasks. The highest test accuracy for each dataset is underlined.*

The maximal test accuracy (higher is better) of uni-tasks and micro-tasks (16) is similar (≈0.74% lower) but degrades with an increasing number of tasks (Table 5.7). As the required test accuracy is use-case dependent, no final verdict on the impact of the measured differences can be made.

Summary. Table 5.8 summarizes the results. Small differences in the convergence behavior and (for lSGD) maximal test accuracy of micro-tasks (16) and uni-tasks can be observed. These are caused by the difference in data partitioning and the weight assigned to each model update during each iteration. The weight of a model update $f_{\Delta,k}$ from task k is computed as the ratio $|\widehat{D}_k|/|\widehat{D}|$ of number of samples $|\widehat{D}_k|$ processed on the aforementioned task to the total number of samples $|\widehat{D}|$ processed during an iteration (see Equation 5.2). This assumes that updates from tasks that process $n\times$ as many samples are $n\times$ as *accurate* as other tasks. This, however, does not consider that due to the local model updates, that occur on each task in-between global model updates, updates from tasks that process more samples (and perform more local model updates), are overproportionally more accurate than updates of tasks with fewer samples. This is due to the fact that each local model update improves the basis for training on subsequent samples. In consequence, the weights assigned to model updates are only an approximation of their actual weight, which can impact convergence.

Overall, micro-tasks (16) and uni-tasks behave similarly w.r.t. the number of epochs needed to converge. However, micro-tasks (16) does not allow for any load-balancing, as the number of tasks executed on each node is already minimal (one task per node), whereas uni-tasks can balance load with one task per node. Hence, even though a similar number of epochs are needed to converge in both cases, the micro-task (16) schedule is gated by tasks on the eight slow nodes.

In order to enable any load-balancing with micro-tasks, at least the next larger scenario with 32 tasks has to be used. This, however already requires up to 2.08× as many epochs as uni-tasks for CoCoA and 1.56× for lSGD (Table 5.8). The average maximal test accuracy for the latter is also slightly lower than for uni-tasks. In general, the more tasks are being used, the more epochs are needed to converge and, in the case of lSGD, the lower the average maximal test accuracy is. In all but one case (CIFAR-10, compared

to micro-tasks (16)), uni-tasks converges in fewer epochs than any micro-tasks scenario and, in the case of lSGD, also achieves the same or higher test accuracies.

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
Higgs (1e-8)	1394	1474	1724	1864	2007	1.06×	1.24×	1.34×	1.44×
Criteo (1e-8)	54	57	94	100	120	1.06×	1.74×	1.85×	2.22×
KDDA (1e-5)	60	68	125	–	–	1.13×	2.08×	–	–
Webspam (1e-6)	37	46	71	86	–	1.24×	1.92×	2.32×	–

(a) CoCoA

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
CIFAR-10 (61%)	27.3	27.1	41.7	56.5	67.9	0.99×	1.53×	2.07×	2.49×
F.-MNIST (89%)	16.5	17.1	25.8	32.0	38.8	1.04×	1.56×	1.94×	2.35×

(b) lSGD

Table 5.8: *Absolute and relative (compared to uni-tasks) number of epochs to converge to the target duality-gap and test accuracy respectively. "–" indicates that the target was not reached within the test time limit.*

5.5.2.5 Schedule length projection

Given the measured number of epochs needed to converge, one can determine the corresponding number of iterations (rounds of global communication) needed to converge. Using the number of iterations to converge × the duration of each iteration, the length of the entire schedule and thus the duration (in units of time) for the entire training process can be projected.

Number of iterations. For CoCoA, one epoch equals one iteration. For lSGD, the number of iterations per epoch is $\frac{\text{size of dataset}}{\text{batch size}}$. The batch size is $K \times L \times H$ with fixed $L = 8$ and $H = 16$ and a variable number of tasks K. Table 5.9 lists the number of iterations per epoch for lSGD.

	uni-tasks	micro-tasks			
Dataset	**16**	**16**	**32**	**48**	**64**
CIFAR-10	24.4	24.4	12.2	8.1	6.1
Fashion-MNIST	29.3	29.3	14.6	9.8	7.3

Table 5.9: *Number of iterations per epoch for lSGD. Dataset sizes are given in Table A.19.*

Iteration duration. The duration of each iteration depends on

- Task runtime, which depends on the number of training samples processed by a task and whether it is being executed on a fast or a slow node.
- Number of back-to-back tasks per node.

All task runtimes are normalized, such that when using 16 micro-tasks, each processing 1/16th of the workload (training samples), one task runs for one time unit on a fast and 1.5 time units on a slow node. This corresponds to the node performance profile used in the above experiments.

For CoCoA, the projected task runtimes and iteration durations are shown in Table 5.9a. To balance the workload in uni-tasks such that tasks on fast nodes process 1.5× as much data as tasks on slow nodes, the former needs to process $1/13.\bar{3}$th of the workload and the latter 1/20th. This results in $16/13.\bar{3} \times 1 = 1.2$ time units for uni-tasks on fast nodes and $16/20 \times 1.5 = 1.2$ time units on slow nodes. The task schedule of each iteration is depicted in Figure 5.7a. For micro-tasks, the workload is distributed evenly across all tasks, hence a normalized task runtime of 16/number of tasks is used. The task schedule of each iteration is depicted in Figures 5.7b through 5.7e.

For lSGD, the projected task runtimes and iteration durations are shown in Table 5.9b. The difference to CoCoA is that in lSGD, the number of training samples depends on the number of tasks, whereas in CoCoA, it is constant. For uni-tasks and micro-tasks (16), this is the same as for CoCoA. For 32, 48 and 64 micro-tasks, the number of training samples increases. As result, the task runtime is constant independently of the number of used micro-tasks but the iteration duration increases, as with more micro-tasks, more training samples are processed per iteration.

	Task runtimes			
# of tasks	**Fast node**	**Slow node**	**Iteration duration**	**Used for**
16	1.000	1.500	1.500	micro-tasks (16)
32	0.500	0.750	1.500	micro-tasks (32)
48	0.333	0.500	1.333	micro-tasks (48)
64	0.250	0.375	1.250	micro-tasks (64)
16	1.200	1.200	1.200	uni-tasks

(a) CoCoA

	Task runtimes			
# of tasks	**Fast node**	**Slow node**	**Iteration duration**	**Used for**
16	1.0	1.5	1.5	micro-tasks (16)
32	1.0	1.5	3.0	micro-tasks (32)
48	1.0	1.5	4.0	micro-tasks (48)
64	1.0	1.5	5.0	micro-tasks (64)
16	1.2	1.2	1.2	uni-tasks

(b) lSGD

Table 5.10: *Assumed task runtimes (in time units) for each micro-tasks and uni-tasks scenario. The schedules that correspond to the iteration durations are depicted in Figures 5.7 and 5.8.*

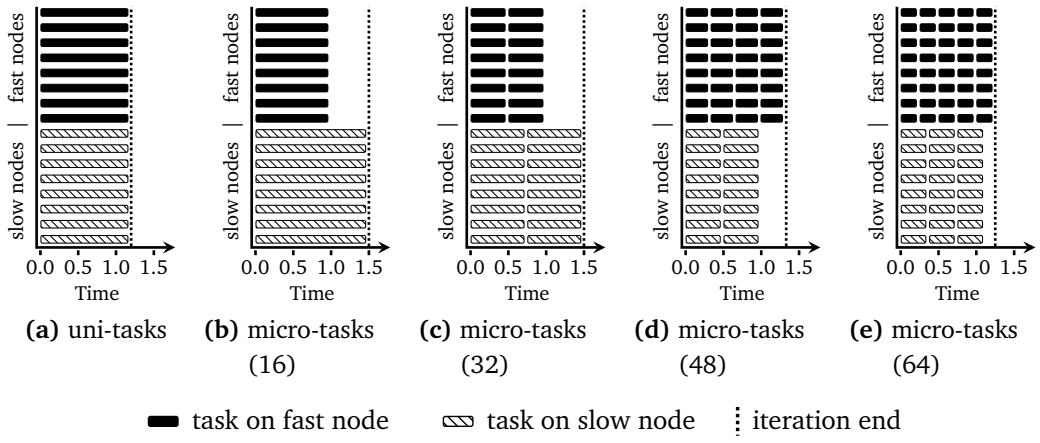
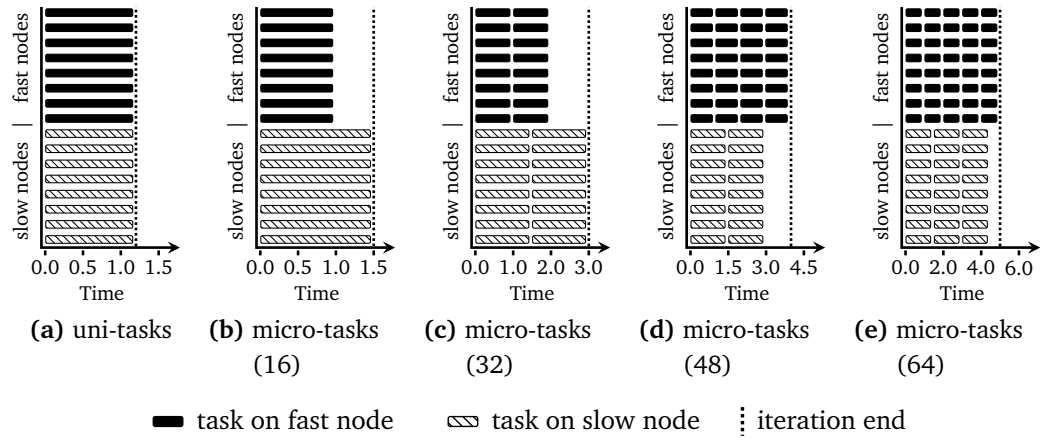


Figure 5.7: *CoCoA minimal iteration schedules for uni-tasks and each micro-tasks scenario. Runtimes of tasks correspond to those listed in Table 5.9a.*



(a) uni-tasks **(b)** micro-tasks (16) **(c)** micro-tasks (32) **(d)** micro-tasks (48) **(e)** micro-tasks (64)

Figure 5.8: *LSGD minimal iteration schedules for uni-tasks and each micro-tasks scenario.* ***Notice the different x-axis values compared to Figure 5.7****. Runtimes of tasks correspond to those listed in Table 5.9b.*

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
Higgs (1e-8)	1673	2211	2586	2485	2509	1.32	1.55	1.49	1.50
Criteo (1e-8)	65	86	141	133	150	1.32	2.17	2.05	2.31
KDDA (1e-5)	72	102	188	–	–	1.42	2.61	–	–
Webspam (1e-6)	44	69	106	115	–	1.57	2.41	2.61	–

(a) CoCoA

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
CIFAR-10 (61%)	799	992	1526	1831	2071	1.24	1.91	2.29	2.59
F.-MNIST (89%)	580	752	1130	1254	1416	1.30	1.95	2.16	2.44

(b) lSGD

Table 5.11: *Projected schedule lengths for CoCoA and lSGD. "–" indicates that the target was not reached within the test time limit.*

Projection. Based on these task runtimes, minimal iteration schedules are computed. Let e be the number of epochs needed to converge, $|e|$ the number of iterations per epoch[8] and τ the iteration duration. The projected schedule length $T = e \times |e| \times \tau$. For instance, for micro-tasks(16) and the Fashion-MNIST dataset with lSGD, $e = 17.1$, $|e| = 29.3$ and

[8] $|e| = 1.0$ in all CoCoA experiments as here, one epoch equals one iteration.

$\tau = 1.5$, the schedule length $T = 17.1 \times 29.3 \times 1.5 = 752$ time units. All projected schedule lengths are listed in Table 5.11. Projections assume zero system (scheduling, communication, task launch, ...) overheads.

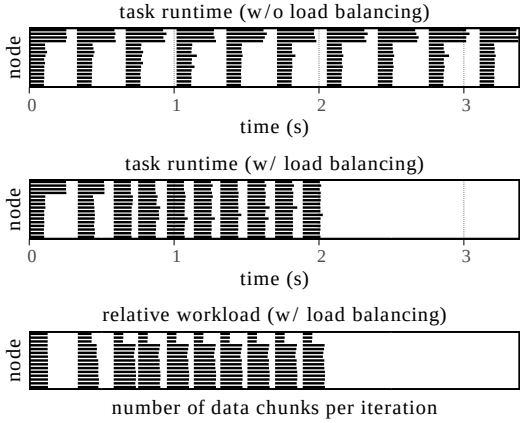
Results show, that the increased scheduling granularity, by using more micro-tasks, cannot compensate for the increase in the number of epochs needed to converge. In all micro-task scenarios, 16 tasks result in the shortest schedule lengths, even though no load-balancing is possible. This shows the ineffectiveness of micro-tasks in balancing load for both training algorithms. In general, micro-tasks need at least 24% to 57% longer than uni-tasks under the assumed node performance profile.

5.5.2.6 Experiment 2: Automatically determine node performance and balance load

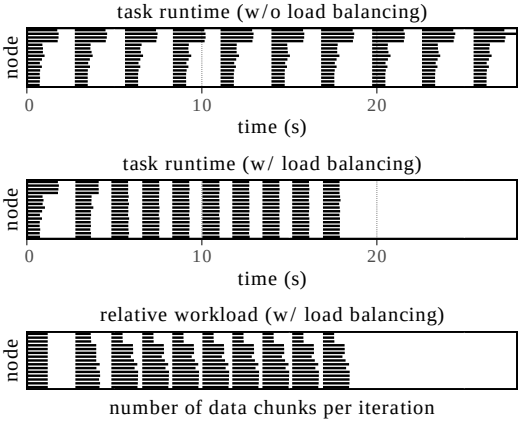
This experiment evaluates the rebalance policy's ability to determine task runtimes and balance workload accordingly. To that end, uni-tasks with load-balancing is compared to micro-tasks (16) without load-balancing. Due to the lack of another load-balancing distributed ML training framework, no other comparisons can be made here. All tests are executed with the test setup described in Section A.3.3. Four nodes of node class three (Table A.17) have been slowed down to 1.20GHz to increase the heterogeneity of the test cluster. As baseline, micro-tasks (16) is also evaluated with no slowed down nodes.

The swimlane diagrams in Figure 5.9 show the task runtimes during each iteration during the initial load-balancing process on the heterogeneous cluster. For comparison, task runtimes per iteration are also shown without load balancing enabled. For the case with load-balancing, the workload on each node during each iteration is provided in Figure 5.9 (bottom plots) to visualize how Chicle shifts load from slow to fast nodes and how this affects task runtimes. The x-axis of the workload plot shows the relative workload and not time. Each iteration of the workload plot corresponds to the same iteration in the task runtime plot with load balancing. Section 2.5.2.1 provides a detailed explanation of these figures. Table 5.12 summarizes the most important results of this experiment.

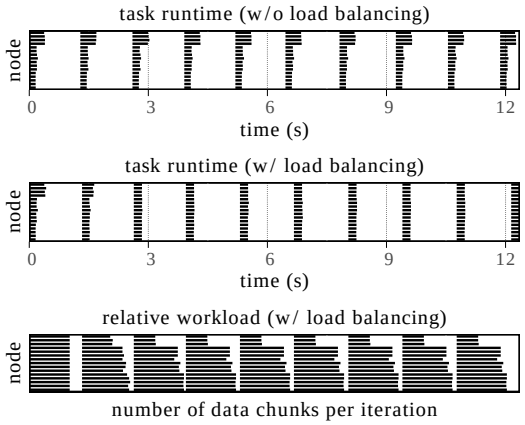
Results. The execution plots in Figure 5.9 show that task runtimes with load-balancing start unequal, whereas the workload starts out equal across all nodes (shown by the equally long bars during the first iteration). Over the course of 2 – 3 (CoCoA) and ≈50 iterations (lSGD, >1000 iterations in total), the rebalance policy learns to predict task runtimes in correspondence to the workload and balances the workload by shifting data chunks from slow nodes to fast nodes. As results, task runtimes equalize across all nodes, which is also reflected in the task runtime standard deviation listed in Table 5.12. The latter is reduced by 12.28× on average with load-balancing compared to without load-balancing. For CoCoA (Figures 5.9a through 5.9d), this process requires fewer iterations,



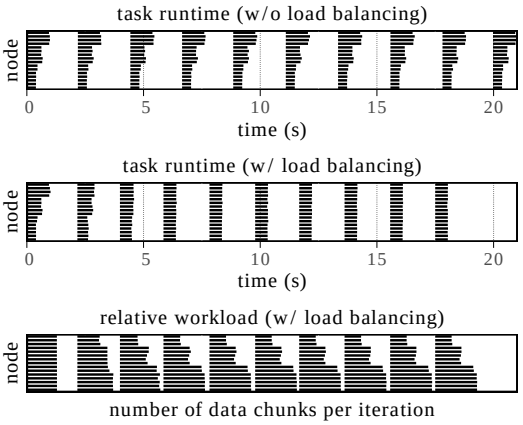
(a) Higgs (CoCoA)



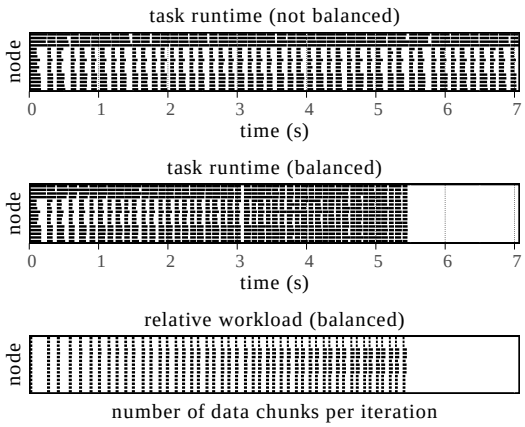
(b) Criteo (CoCoA)



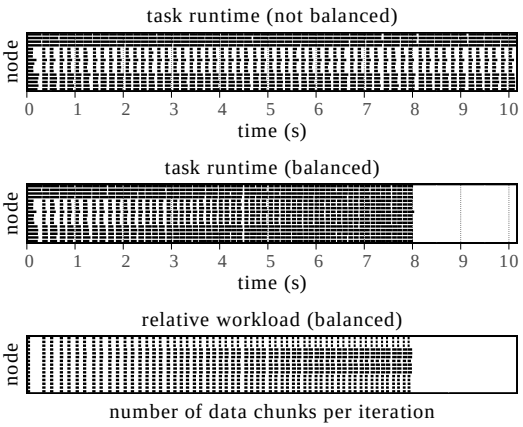
(c) KDDA (CoCoA)



(d) Webspam (CoCoA)



(e) CIFAR-10 (lSGD)



(f) Fashion-MNIST (lSGD)

Figure 5.9: *Swimlane diagram of the load balancing process (first 10 iterations for CoCoA and first 50 iterations for lSGD). CPUs of the top four nodes run at a reduced frequency. This type of diagram is described in Section 2.5.2.1.*

as r_{max} is set to 0.5 here, whereas for lSGD (Figures 5.9e and 5.9f), with $r_{max} = 0.05$, it takes more time.

However, Figures 5.9c and 5.9d also show that even though task runtimes are equalized after the workload has been balanced, iteration times do not necessarily decrease. The corresponding datasets, KDDA and Webspam, have a large number of features (20M and 17M), whereas all other datasets have at most 1M features. The size of the model update vector depends on the number of features. As a result, this vector is large: 154MiB for KDDA and 127MiB for Webspam. During each iteration, the model update vector has to be transmitted from all 16 nodes to the trainer, where it is reduced and broadcast back to each node before the next iteration can start.

Data transfer for broadcast is started at the same time on all nodes and takes ≈610ms for KDDA and ≈500ms for Webspam with and without load balancing. Reduce times are not measured separately by Chicle, however it can be assumed that with load-balancing enabled, all tasks start data transfer at the same time and reduce times are similar to broadcast times. Without load-balancing, data transfer is staggered and reduce times per node are lower than broadcast times. This allows hiding delays in computation of tasks on slow nodes with data transfer from tasks on fast nodes.

As task runtimes in KDDA are significantly shorter than data transfer times (≈150ms vs. ≈610ms), slow tasks can be completely hidden, and no benefit from load-balancing can be achieved. For Webspam, task runtimes are slightly longer than data transfer times (≈540ms vs. ≈500ms), hence slow tasks cannot be hidden completely and iteration runtimes can be reduced. In order to realize benefits from workload-balancing for datasets such as KDDA and Webspam, Chicle needs to implement more efficient reduce and broadcast operations, e.g., using a tree.

However, for all other datasets, data transfer times are neglectable compared to task runtimes, due to the smaller number of features and therefore model update vector size. Here, load-balancing reduces iteration time by 1.56× on average and is even 3.3% faster than the baseline case, were no nodes have been slowed down. The latter is possible because even in the baseline case, small performance imbalances exist between nodes.

5.5.2.7 Evaluation summary

In all tested scenarios, uni-tasks reduces the number of epochs necessary to converge with only minimal impairment of the average maximal test accuracy for CIFAR-10. However, the only micro-tasks case that achieves a higher maximal test accuracy uses 16 tasks, which does not allow any load-balancing on the 16 node test cluster. Furthermore, the schedule projection shows the conflict between schedule efficiency, i.e., the ability to balance load, and the training efficiency, i.e., the number of epochs needed to converge. This conflict does not exist in uni-tasks.

	Higgs			Criteo		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	186ms	349ms	186ms	1834ms	2905ms	1903ms
mean task time	113ms	149ms	132ms	863ms	1120ms	1009ms
task time std. dev.	15ms	69ms	7ms	92ms	453ms	66ms

	KDDA			Webspam		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	1336ms	1362ms	1398ms	1804ms	2245ms	1965ms
mean task time	131ms	173ms	150ms	481ms	621ms	540ms
task time std. dev.	14ms	80ms	4ms	110ms	246ms	10ms

(a) CoCoA

	CIFAR-10			Fashion-MNIST		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	136ms	181ms	127ms	166ms	207ms	147ms
mean task time	73ms	92ms	81ms	88ms	108ms	98ms
task time std. dev.	17ms	30ms	5ms	24ms	38ms	6ms

(b) lSGD

Table 5.12: *Summary of results (mean iteration and task runtime, as well as standard deviation of the latter) for the load-balancing experiments. Each dataset was trained with (w/) and without (w/o) rebalancing and compared to the baseline, without slowed-down nodes.*

5.6 Elasticity

Elastic execution is a special case of load balancing, where node performance changes from *full* to *none* (if a node is removed) and vice versa (if a node is added) and can therefore be addressed using the same data chunk moving mechanisms as described in Section 5.5, albeit with a much simpler policy (i.e., without the need for task runtime predictions).

An example of elastic scale-out is depicted in Figure 5.10. Here, a node is added after iteration i finishes and parts of the workload (in form of data chunks) of nodes 1 and 2 is moved to the newly added node 3. As per the task contract, data chunks cannot be moved during an iteration, hence this move occurs in-between iterations i and $i+1$. Due to the reduced workload per node, the runtime of iteration $i+1$ is reduced. This process is repeated after iteration $i+1$ where node 4 is added.

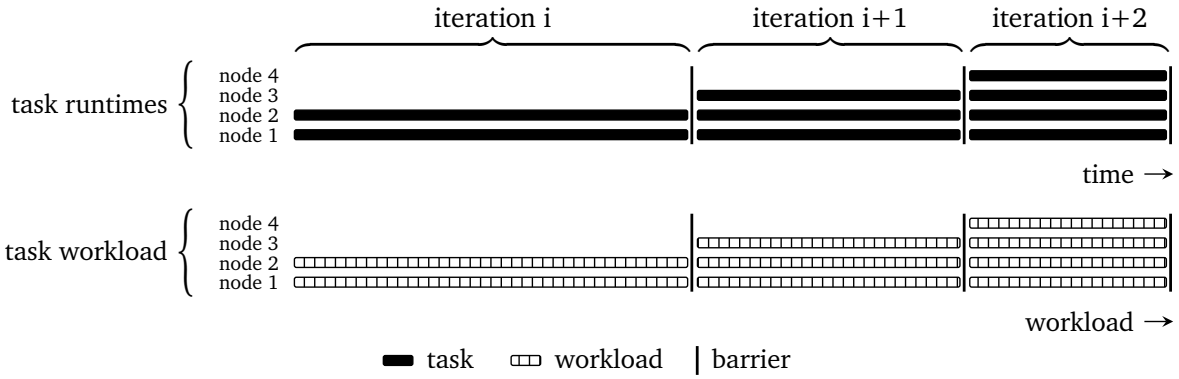


Figure 5.10: *Conceptual depiction of the workload distribution process during elastic scale-out with uni-tasks. Necessary data transfer times are not shown.*

An example of elastic scale-in is depicted in Figure 5.11. Here, node 4 is removed after iteration i finishes and the total workload of node 4 is spread across the remaining nodes 1–3 in-between iterations i and $i+1$. Due to the increased workload on the remaining nodes, runtime of iteration $i+1$ increases. This process is repeated after iteration $i+1$ where node 3 is removed. As per the task contract, nodes can only be removed in-between iterations, hence the resource manager needs to announce the removal of a node ahead of time.

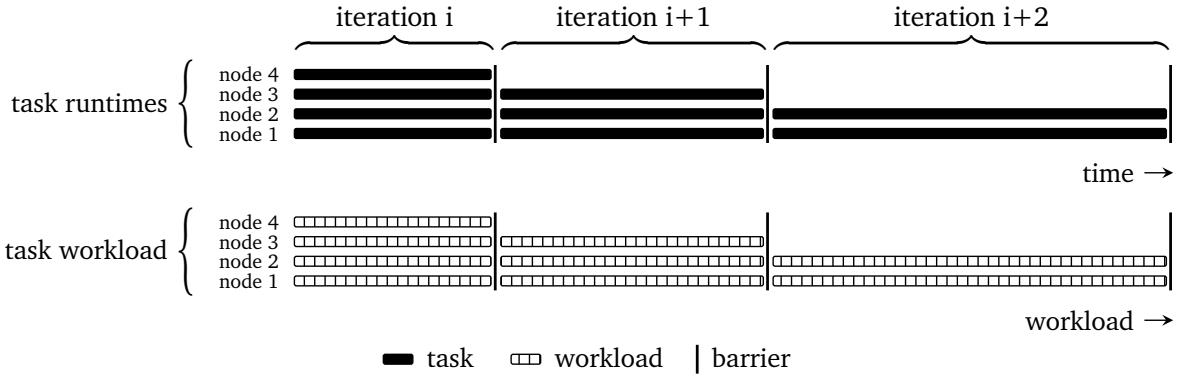


Figure 5.11: *Conceptual depiction of the workload concentration process during elastic scale-in with uni-tasks. Necessary data transfer times are not shown.*

5.6.1 Load balancing in Chicle: Elasticity policy

The elasticity policy implements the method described above. It provides the ability of Chicle applications to elastically scale in and out depending on resource availability and application demands. It subscribes to *worker registered* and *remove worker* events which are emitted by the trainer and the resource manager respectively. However, as Chicle is currently not integrated into a resource manager such as Mira or YARN, these events are

generated internally, according to a configurable schedule. This is sufficient to evaluate the behavior of Chicle applications in elastic scenarios.

Upon receiving a *worker registered* event, this policy calls the `scaleout` function that moves data chunks from existing to newly registered worker(s) in a round robin fashion, until the number of chunks assigned to them is equal to the average number of chunks across all workers. Afterwards it emits the *worker added* event to notify the trainer about the new worker(s). The code for the `scaleout` function is shown in Listing 5.2.

```
1  void scaleout(list<Worker*> workers)
2  {
3    do {
4      // 'workers' is a list with all currently active workers, including the newly added
5      // ones.  Sort this list in ascending order of the number of chunks on each worker.
6      sort(workers);
7
8      Worker* leastLoaded = workers.first; // worker with the fewest data chunks
9      Worker* mostLoaded  = workers.last;  // worker with the most data chunks
10
11     Chunk* chunk = mostLoaded->chunks.first;
12
13     // Move 'chunk' from 'mostLoaded' to 'leastLoaded' worker.  Chunk data movement is
14     // performed in the background, hence this call is non-blocking.
15     move(chunk, mostLoaded, leastLoaded);
16   } while (leastLoaded->chunks.size + 1 < mostLoaded->chunks.size)
17
18   // wait for all chunk data movements to complete and emit worker added event
19 }
```

Listing 5.2: *Simplified C++ code of the elasticity policy's scale-out algorithm.*

Similarly, upon receiving the *remove worker* event, the `scalein` function moves all data chunks from the to-be-removed worker(s) onto the remaining workers in a round robin fashion. After all chunks have been moved, it emits the *worker removed* event to notify the trainer about the removed worker(s), as well as to confirm the removal to the resource manager. The code for the `scalein` function is shown in Listing 5.3.

```
void scalein(list<Worker*> workers, list<Worker*> remove)
{
  // 'workers' is a list with all currently active workers, excluding the to-be removed
  // ones.  Shuffle the list to avoid concentrating chunks on a few workers in case this
  // function is called repeatedly.
  shuffle(workers);

  // 'remove' is a list with workers that should be removed.
  while (!remove.empty()) {
    Worker *removeWorker = remove.first;
    while(removeWorker.chunks.size > 0) {
      Chunk*  chunk           = removeWorker.chunks.first;
      Worker* remainingWorker = workers.next;

      // Move chunk from a to-be-removed worker to a remaining one.  Chunk data movement
      // is performed in the background, hence this call is non-blocking.
      move(chunk, removeWorker, remainingWorker);
    }

    // Remove the worker as no more chunks are left.
    removeWorker.pop_first;
  }

  // wait for all chunk data movements to complete and emit worker removed event
}
```

Listing 5.3: *Simplified C++ code of the elasticity policy's scale-in algorithm.*

In neither case does this policy consider knowledge about node performance when shifting data chunks, but relies on the rebalance policy to reestablish task runtime balance. A possible improvement to this policy is to exploit this information, as far as node performance is known, to reduce the need for subsequent rebalance actions.

5.6.2 Evaluation

This section presents evaluation results for elastic scaling experiments performed on Chicle to compare uni-tasks with micro-tasks using CoCoA and lSGD (see Section A.3.1 for details). The purpose of these experiments is to show the advantage of uni-tasks over micro-tasks in terms of convergence per epoch and time, in scenarios where elastic scaling is necessary or beneficial.

Experiments. Two scenarios with the following node availability profiles have been evaluated:

- Scaling in from 16 to two nodes by removing two nodes every 20s.

- Scaling out from two to 16 nodes by adding two nodes every 20s.

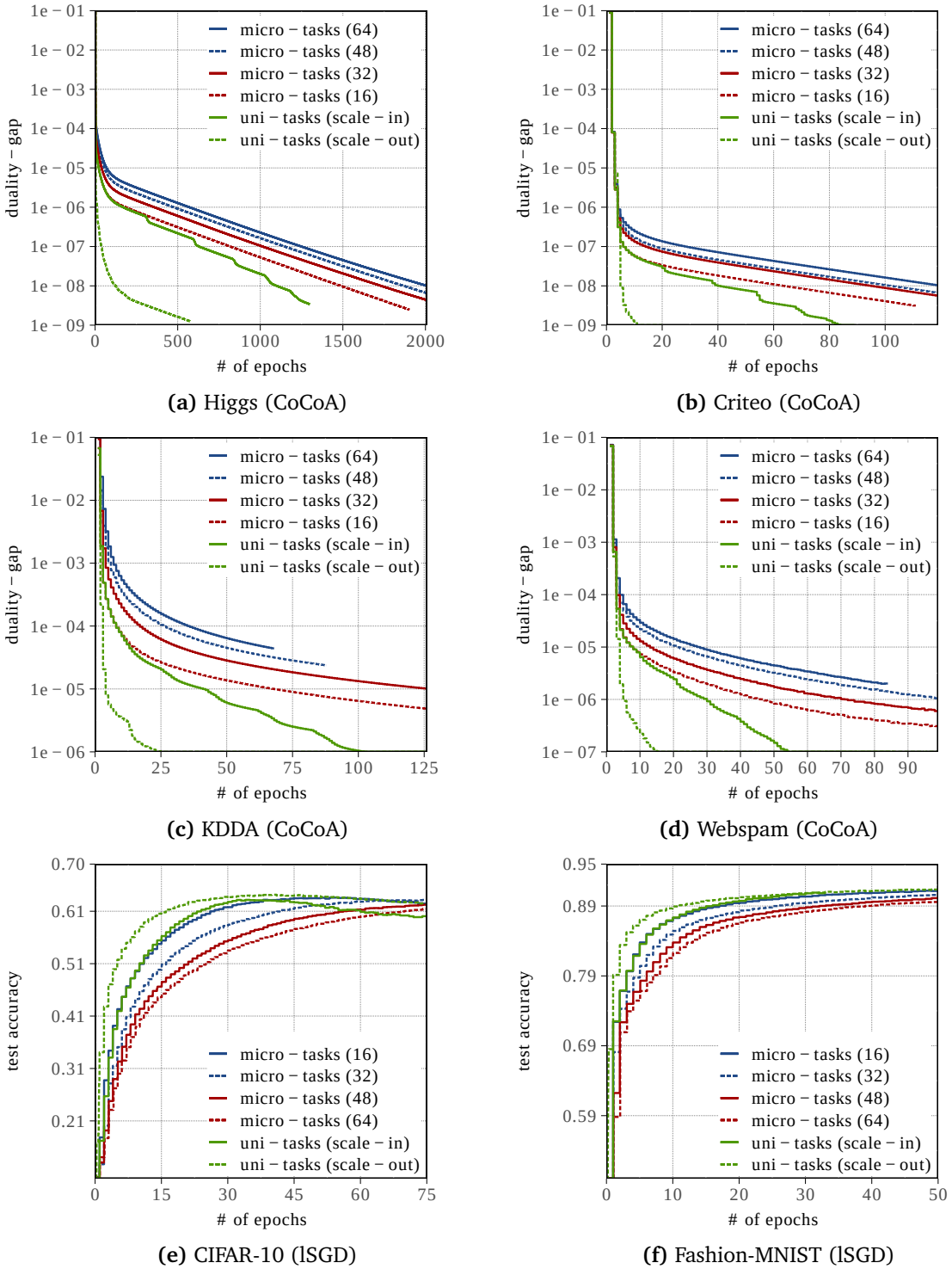
In each scenario, the metrics described in Section 5.5.2.1 are collected, i.e., duality-gap and test accuracy to measure training progress as well as number of epochs to converge. Iteration durations are computed based on these measurements.

All experiments use the test setup and datasets described in Section A.3.3. For uni-tasks, Chicle's elasticity policy has been configured according to the given node availability profiles and the number of epochs to converge is measured. Parameters for CoCoA and lSGD are the same as in Section 5.5.2.2. For micro-tasks, the node availability profile does not have an impact on the number of epochs needed to converge. Instead, it depends on the number of tasks (and therefore data partitions) used to execute training. As this is the same as with the load balancing experiments, measurements for the number of epochs to converge are reused for micro-tasks.

Similarly, to the load balancing experiments, the lack of publicly available competitive elastic ML training frameworks prevents a direct comparison of uni-tasks with micro-tasks. Instead, the length of a schedule is projected to estimate time savings of uni-tasks over micro-tasks. This projection is based on the measured number of epochs to converge and the node availability profile.

5.6.2.1 Epochs to converge

Figure 5.12 shows convergence plots for CoCoA and lSGD. Tables 5.13 and 5.14 list the corresponding absolute and relative differences of the number of epochs needed to converge between uni-tasks and micro-tasks for elastic scale-in and scale-out. Table 5.15 lists the average maximal achieved test accuracy for lSGD.



(a) Higgs (CoCoA) **(b)** Criteo (CoCoA)

(c) KDDA (CoCoA) **(d)** Webspam (CoCoA)

(e) CIFAR-10 (lSGD) **(f)** Fashion-MNIST (lSGD)

Figure 5.12: *Convergence of the duality-gap (lower is better) and test accuracy (higher is better)* ***over epochs*** *in elastic scenarios. This plot shows the impact of data parallelism on the convergence behavior of micro-tasks compared to uni-tasks with the given node availability profiles. For micro-tasks, the number of partitions is given in parentheses. For uni-tasks, the number of partitions varies between 2 – 16.*

In the CoCoA convergence plots (Figure 5.12), scale-in events are visible by the sudden drops of the duality-gap in the coresponding plot line. Scale-out events are only visible in Figure 5.12c as drops of the duality-gap. As scale-in/out events happen every 20s and the plots show the number of epochs on the x-axis, these events happen at different points on the x-axis. Moreover, the distance between scale-out (in) events in the plots changes, as nodes are added (removed), which reduces (prolongs) the runtime of an epoch. For lSGD, the effects are less pronounced, scale-in/out events are not visible in the plots.

The expectation for this experiment is that uni-tasks requires at most as many epochs to converge as micro-tasks (16) but fewer than in any other micro-tasks scenario. This expectation is based on the observation (see Figure 5.1) that the number of partitions/-batch size is the main determinator for the convergence rate.

CoCoA. As Figure 5.12 shows, the convergence rate for CoCoA, when scaling in and out, is much higher than in any micro-tasks scenario. Here, uni-tasks can achieve a reduction in the number of epochs to converge of between 1.24× – 1.64× for scale-in and 7.12× – 21.8× for scale-out, compared to the best micro-tasks (16) case. The reason for this is threefold:

(1) In the scale-in scenario, each local SCD solver instance gains access to additional training data, which allows it to identify more correlations locally. This is reflected as small *drops* of the duality-gap in the scale-in plot, which appear after each scale-in step. This effect is also exploited in by the auto scale-in policy (Section 5.9.1).

(2) In the scale-out scenario, only few tasks exist in the beginning, each with access to large amounts of training data. As mentioned earlier, fewer tasks generally allow for faster convergence per epoch, which is what causes this initial convergence head start. When the number of tasks increases, the duality-gap has already dropped significantly.

(3) In both cases, data chunks are recombined differently after a scale-in/out step, allowing the local SCD solver to identify additional correlations locally.

Micro-tasks, on the other hand, cannot benefit from scale-in nor out in terms of convergence, as the data per task remains the same, no matter where a task is executed.

lSGD. Similar to CoCoA, the lSGD results show that the number of epochs needed to converge increases with a larger number of micro-tasks while uni-tasks consistently requires the least number of epochs to converge. In the scale-in case, uni-tasks is 1.08× – 1.15× faster than the best micro-tasks case, and 1.51×– 1.68× faster in the scale-out case.

lSGD also exhibits a behavior where **scale-out** converges faster than scale-in when using uni-tasks. The reason for this is different than for CoCoA, though. By scaling out towards the end of the training, the batch size increases and the relative learning rate

decreases implicitly, as it is scaled with $\sqrt{n}$ for a change of $\times n$ of the batch size. Both methods, batch size increase and learning rate decrease, have been suggested in related work to improve convergence [81, 124, 86]. Both adjustments are typically done once the convergence rate slows down.

In the **scale-in** scenario, the test accuracy for CIFAR-10 on uni-tasks degrades noticeably after reaching its peak. The reason for this behavior is that scale-in happens once the convergence has already slowed down significantly and the opposite of what is suggested in the aforementioned related work happens: The batch size is decreased relative to the learning rate. The same mitigation strategies can be used, though, to counter this effect.

For CIFAR-10, the average maximal test accuracy (Table 5.15) is slightly below that of micro-tasks (16) in the scale-in case (≈0.45% lower). In the scale-out case as well as in both cases for Fashion-MNIST, uni-tasks achieves the highest average maximal test accuracy across all runs.

	uni-tasks	**micro-tasks (absolute)**				**micro-tasks (relative)**			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
Higgs (1e-8)	1103	1474	1724	1864	2007	1.34×	1.56×	1.69×	1.82×
Criteo (1e-8)	42	57	94	100	120	1.36×	2.24×	2.38×	2.86×
KDDA (1e-5)	42	68	125	–	–	1.62×	2.98×	–	–
Webspam (1e-6)	28	46	71	86	–	1.64×	2.54×	3.07×	–

(a) CoCoA

	uni-tasks	**micro-tasks (absolute)**				**micro-tasks (relative)**			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
CIFAR-10 (61%)	23.5	27.1	41.7	56.5	67.9	1.15×	1.77×	2.40×	2.89×
F.-MNIST (89%)	15.8	17.1	25.8	32.0	38.8	1.08×	1.63×	2.03×	2.46×

(b) lSGD

Table 5.13: *Absolute and relative (compared to uni-tasks) number of epochs to converge to the target duality-gap and test accuracy respectively during **scale-in**. "–" indicates that the target was not reached within the test time limit.*

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
Higgs (1e-8)	141	1474	1724	1864	2007	10.5×	12.2×	13.2×	14.2×
Criteo (1e-8)	8	57	94	100	120	7.12×	11.8×	12.5×	15.0×
KDDA (1e-5)	4	68	125	–	–	17.0×	31.2×	–	–
Webspam (1e-6)	6	46	71	86	–	7.67×	11.8×	14.3×	–

(a) CoCoA

	uni-tasks	micro-tasks (absolute)				micro-tasks (relative)			
Dataset (target)	**16**	**16**	**32**	**48**	**64**	**16**	**32**	**48**	**64**
CIFAR-10 (61%)	16.1	27.1	41.7	56.5	67.9	1.68×	2.59×	3.51×	4.22×
F.-MNIST (89%)	11.3	17.1	25.8	32.0	38.8	1.51×	2.28×	2.83×	3.43×

(b) lSGD

Table 5.14: *Absolute and relative (compared to uni-tasks) number of epochs to converge to the target duality-gap and test accuracy respectively during **scale-out**. "–" indicates that the target was not reached within the test time limit.*

	uni-tasks		micro-tasks			
Dataset	**scale-out**	**scale-in**	**16**	**32**	**48**	**64**
CIFAR-10	64.29%	63.61%	63.90%	63.34%	62.34%	61.48%
Fashion-MNIST	91.31%	91.48%	91.28%	90.66%	90.20%	89.71%

Table 5.15: *Average maximal test accuracy (higher is better) for lSGD on uni-tasks and micro-tasks. The highest test accuracy for each dataset is underlined (scale-out) and double underlined (scale-in).*

5.6.2.2 Iteration schedule projection

In this section, iteration schedules are projected and their duration determined to estimate the training time and resource utilization efficiency. In contrast to the load balancing case, no entire schedules can be projected, as the number of tasks that run at the same time is not constant. This number, however, can have a significant impact on the actual iteration time due to data transfer overheads, especially for datasets with large model update vectors, such KDDA and Webspam. As the projection of the entire schedule does not consider these overheads, no realistic runtimes can be computed. Instead,

only iteration schedules are projected which take the training time, but explicitly not the data transfer times into account.

As no publicly available, competitive elastic ML training framework is available, no actual measurements could be performed.

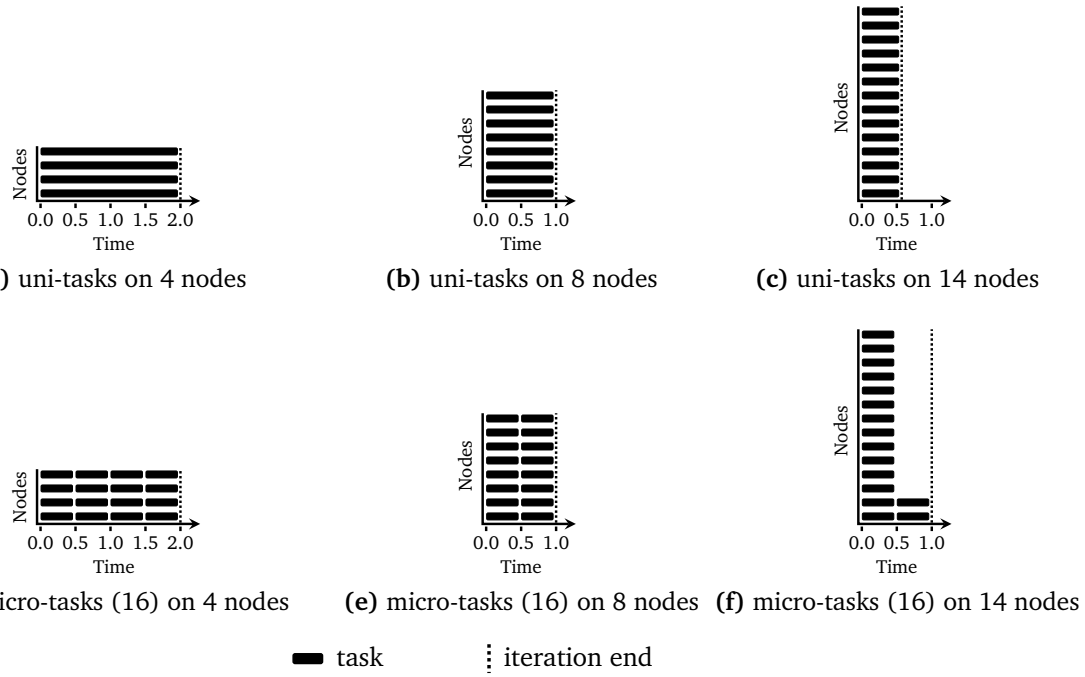


Figure 5.13: *Minimal iteration schedules for CoCoA. In uni-tasks, the workload is redistributed evenly across all tasks.*

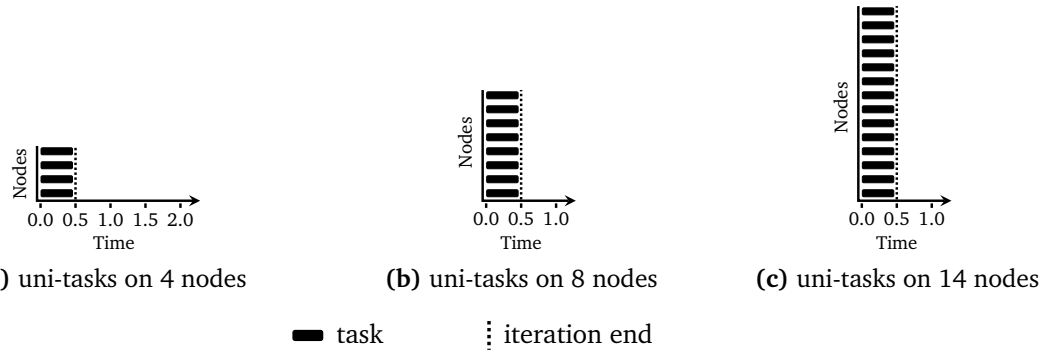


Figure 5.14: *Minimal iteration schedules for lSGD. In uni-tasks, the batch size is scaled with the number of nodes whereas it remains constant for micro-tasks. The number of iterations per epoch increases accordingly for uni-tasks. Micro-tasks schedules for lSGD are the same as in Figure 5.13*

Iteration duration. The duration of each iteration depends on

- Task runtime, which depends on the number of training samples processed by a task and whether it is being executed on a fast or a slow node.

- Maximal number of back-to-back tasks (task waves).

Iteration durations for uni-tasks and micro-tasks can be computed according to the depictions in Figures 5.13 and 5.14. All task runtimes are normalized, such that when using 16 micro-tasks, each processing 1/16th of the workload (training samples), one task runs for one time unit. For instance, consider the case with 14 nodes:

- For CoCoA on uni-tasks, the number of tasks is reduced, load is redistributed and each task runs $16/14 = 1.14\times$ longer on 14 nodes than on 16 nodes, hence the iteration duration is 1.14 as long as on 16 nodes. Micro-tasks (16) still executes 16 tasks. However, as only 14 can be executed at the same time, two task waves are necessary and the iteration duration doubles compared to 16 nodes. This makes iterations for micro-tasks (16) $2/1.14 = 1.75\times$ as long as for uni-tasks when using 14 of nodes.

- For lSGD, uni-tasks reduces the number of tasks *and* the batch size by $16/14 = 1.14\times$ when scaling in from 16 to 14 nodes, whereas micro-tasks (16) continues to process the same number of samples per iteration as on 16 nodes. To compensate for the fewer training samples per iteration, $1.14\times$ more iterations per epoch are needed. Micro-tasks (16) behaves identically to CoCoA, hence the overall iteration duration remains $2/1.14 = 1.75\times$ as long on micro-tasks (16) than on uni-tasks.

Table 5.16 lists iteration durations of all micro-task scenarios and node counts relative to uni-tasks. The results of these calculations show that iteration durations for micro-tasks (16), which requires the fewest epochs to converge among all micro-tasks configurations, are between $1.12\times$ and $1.75\times$ as long as for uni-tasks in four out of eight node configurations. Micro-tasks (64) can reduce this to at most $1.12\times$ but requires between $1.82\times$ and $2.89\times$ (Table 5.13) as many epochs to converge as uni-tasks when scaling in. In two out of four cases, the target duality-gap is not reached during the test time limit. This shows the conflict between training efficiency and scheduling efficiency for micro-tasks, which is not present in uni-tasks.

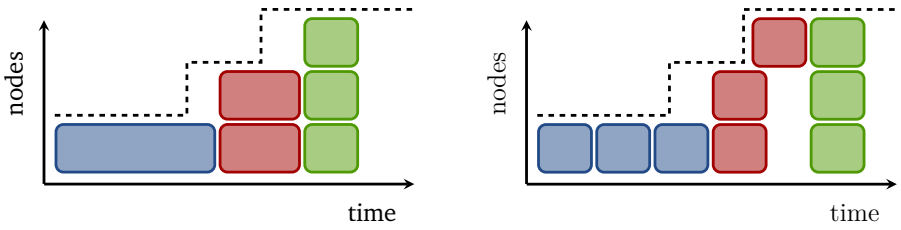
	micro-tasks			
Number of Nodes	**16**	**32**	**48**	**64**
2	1.00×	1.00×	1.00×	1.00×
4	1.00×	1.00×	1.00×	1.00×
6	1.12×	1.12×	1.00×	1.03×
8	1.00×	1.00×	1.00×	1.00×
10	1.25×	1.25×	1.04×	1.09×
12	1.50×	1.12×	1.00×	1.12×
14	1.75×	1.31×	1.17×	1.09×
16	1.00×	1.00×	1.00×	1.00×

Table 5.16: *Micro-tasks iteration schedule length for CoCoA and lSGDg relative to uni-tasks.*

The relative resource utilization efficiency is the inverse of the relative iteration duration between uni-tasks and micro-tasks.

5.6.2.3 Node acquisition and release latency

In micro-tasks and uni-tasks alike, small delays can occur after scaling-out, before the newly added nodes are used for training, as depicted in Figure 5.15.



(a) uni-tasks **(b)** micro-tasks

Figure 5.15: *Elastic (scale-out) schedule examples with 1 – 3 nodes for uni-tasks (a) and micro-tasks (b). Iterations are marked with different colors. The node availability profile is depicted by the dashed line (same in both plots).*

Here, micro-tasks have a potential advantage, as the scheduler can react to the change in node availability and move pending tasks to them during a running iteration. In uni-tasks, on the other hand, the scheduler has to wait for the current iteration to finish, as the task contract does not allow the movement of data chunks during an iteration. A mitigation strategy exists with task preemption, that is also used for straggler mitigation. As soon as a scale-in/out event occurs, running tasks may be preempted and the current iteration cut short. This would allow the scheduler to add or remove nodes sooner. The

latter has not been implemented in the elasticity policy yet, as scale-in/out events are assumed to be rare and to occur with tens of iterations in-between. In this evaluation, this is the case for all but the first few epochs of the CoCoA scale-out tests.

Similarly, during scale-in, nodes in uni-tasks cannot be released immediately, whereas in micro-tasks, tasks are generally shorter and can be rescheduled on other nodes. The same mitigation strategy, task preemption, can be used for uni-tasks.

5.7 Straggler mitigation

Similar to the stale synchronous parallel (SSP) approach described in Section 2.2.4.1, uni-tasks exploits the auto-correcting property of ML training algorithms to mitigate the impact of stragglers. Furthermore, it exploits the ability of tasks to produce valid updates (f_Δ) after processing only a subset of the intended training samples during any given iteration, by **preempting straggling tasks**.[9]

Straggler mitigation via task preemption assumes that training on tasks can be preempted frequently (the more often the better) and that a valid model update, based on the training samples that have been processed before the preemption, can be produced.

Similar to the load balancing case, tasks may process a different amount of training samples in each iteration, with preempted tasks processing fewer samples than non-preempted tasks (assuming the same planned number of samples for each task). The weight of model updates is reduced accordingly (see Equation 5.2 for details).

5.7.1 Straggler mitigation in Chicle: Preemption policy

The preemption policy implements the straggler mitigation technique described above. It complements the rebalance policy with the ability to address intermittently and unpredictably fluctuating task runtimes.

This policy subscribes to *task finished* event and emits the *finish iteration* event to individual tasks after a configurable number of tasks, given via a threshold parameter, have finished the current iteration. Upon receiving the *finish iteration* event, tasks return a model update along with the number of samples that have been processed in the current iteration. Listings 5.4 and 5.5 shows the corresponding source code.

At the time of preemption, the model update is based on all samples that have been processed before the *finish iteration* event was received. Remaining samples, that could not be processed in the current iteration, are processed in subsequent iterations. The effectiveness of this policy in mitigating stragglers depends on the number of training sample sets (H), processed by each task during each iteration. A task is expected to be preemptable in-between the processing of any two sample sets. This is the case for CoCoA, where thousands of training sample sets are typically processed in each iteration.

[9] At the time of writing, a US patent application is in preparation for this straggler mitigation technique.

The current implementation of mSGD and lSGD is based on libtorch. During training, each task processes H training sample sets of with L training samples. For each H, a set of L training samples is handed over to libtorch, which performs the actual training (forward/backward propagation and gradient update) on the entire set. Once the entire set has been processed, the result is returned to the Chicle task. No intermediate results exist.

- In lSGD, $H > 1$ rounds of local training are performed. At the end of each round, libtorch returns a result to Chicle. This result constitutes a valid model update and can be returned to the trainer in case of preemption. Otherwise, the next round of local training is started.
- In mSGD, $H = 1$, only one round of local training is performed and no preemption is possible. Using a different implementation of the local training algorithm may make mSGD preemptable as well but has not been explored as part of this work.

Table 5.17 lists all parameters of the preemption policy.

Parameter	Description
threshold	Fraction of tasks that need to finish before the remaining tasks are preempted.

Table 5.17: *Parameters of the preemption policy parameters used in this evaluation.*

```
1  void workerFinishedEventHandler()
2  {
3    // 'numWorkersFinished' holds the number of finished workers in this iteration.  It
4    // is reset to 0, before a new iteration starts.
5    numWorkersFinished++;
6
7    // 'workers is a list of all currently active workers.
8    if (numWorkersFinished >= workers.size * threshold) {
9      // Iterate over all workers and preempt unfinished ones. The call to 'preempt()' is
10     // asynchronous and non-blocking.
11     for (worker : workers) {
12       if (!worker->finished)
13         worker->preempt(); // emit 'finish iteration' event
14     }
15   }
16 }
```

Listing 5.4: *Simplified C++ code of the preemption policy's decision algorithm in the* worker finished *event handler*

```
void Solver::run(Dataset *dataset) {
  // 'done' is set when the training is finished.
  while (!done) {
    wait(event::iteration_started);

    // 'to_process' is the target number of training samples to process in the current
    // iteration.
    int to_process = H * L;

    // 'processed' is the number of training samples processed in the current
    // iteration.
    int processed = 0;

    // 'preempted' is set when the 'finish iteration' event is received.
    for (processed = 0; processed < to_process && !preempted; processed += L) {
      // train on L samples.
    }

    // Push model updates and signal that the iteration is finished. Tell trainer
    // how many samples have been processed and how many should have been processed.
    send(model_update, processed, to_process);
    signal(event::iteration_finished);
  }
}
```

Listing 5.5: *Simplified C++ code of a solver, which is executed in the task on each worker node.*

5.7.2 Evaluation

This section presents the results of the straggler mitigation experiments performed on the test cluster. Stragglers were introduced artificially by slowing down one randomly selected task in each iteration by ≈50%. The slowdown was realized by inserting small pauses in-between the processing of training sample sets. All experiments use the test setup and datasets described in Section A.3.3 as well as the CoCoA and lSGD training algorithms (see Section A.3.1 for details). Table 5.18 lists parameter values used for the preemption policy. Parameters for CoCoA and lSGD are the same as in Section 5.5.2.2.

Parameter	Value
threshold	0.5

Table 5.18: *Chicle's preemption policy parameters used here.*

Experiments. Chicle with task preemption is compared to Chicle without task preemption in the above described scenario. No direct comparison to another framework is presented here. No other known CoCoA implementation supports straggler mitigation. For lSGD, the SSP concept can be implemented but is not readily available as part of PyTorch, which is used as reference framework later on. Instead, Chicle with task preemption and stragglers is compared to a baseline scenario without stragglers. If the former achieves the same performance as the latter, in terms of iteration runtime and convergence behavior, it is considered effective.

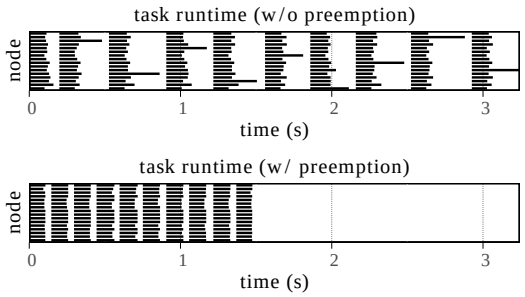
In contrast to the previous evaluations, the wall time can be measured, as no comparisons to micro-tasks are performed.

5.7.2.1 Task and iteration runtimes

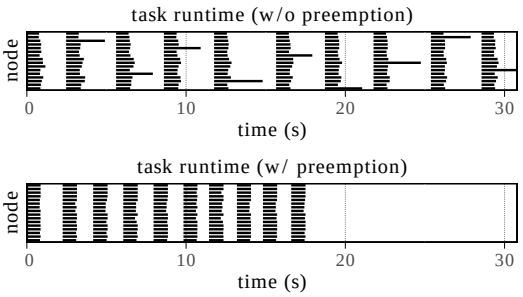
Figures 5.16a through 5.16f show swimlane diagrams of the first 10 iterations with stragglers. The first row of each figure shows the task execution on each node without task preemption and the second row shows it with task preemption. In the latter case, stragglers are mitigated effectively for all datasets. As with the load balancing experiments, the benefit of mitigating stragglers is smaller for KDDA and Webspam, as a large fraction of time is spend with model update vector transfer, which is unaffected by the introduced stragglers.

This is also reflected in the mean iteration time as well as the standard deviation of task runtimes listed in Table 5.19. On average, iteration length is reduced by 1.64× with task preemption compared to without. Its effectiveness is also shown when compared to the baseline, that does not have any *artificially introduced* stragglers. Here, task preemption reduces iteration runtimes by 1.17×. The speedup compared to the baseline is due to natural, unmitigated stragglers in the baseline as well as due to the slight heterogeneity of the test cluster which is partially mitigated by task preemption.

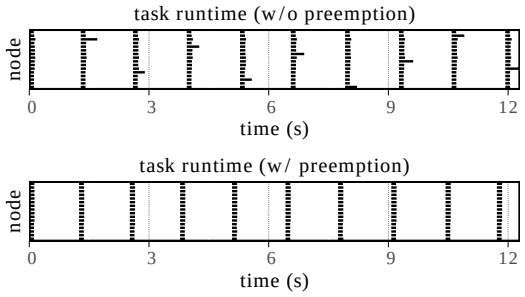
The latter is also visible in Figures 5.16a through 5.16f where tasks on some nodes run consistently longer than on others. The task preemption policy could be combined with the rebalance policy to mitigate the impact of the slight heterogeneity of the test cluster.



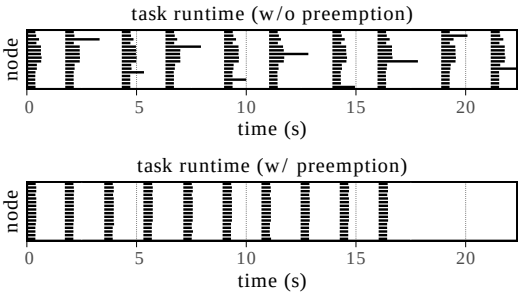
(a) Higgs (CoCoA)



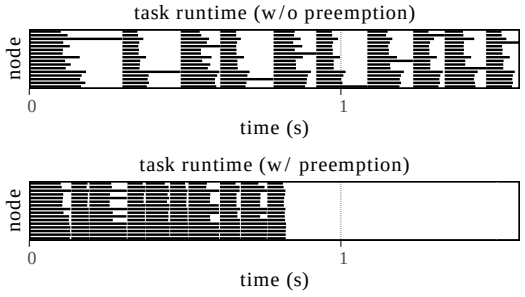
(b) Criteo (CoCoA)



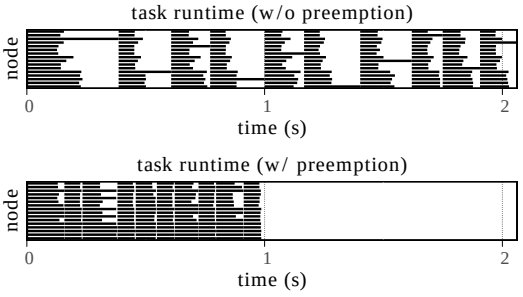
(c) KDDA (CoCoA)



(d) Webspam (CoCoA)



(e) CIFAR-10 (lSGD)



(f) Fashion-MNIST (lSGD)

Figure 5.16: *Visualization of the task preemption straggler mitigation method during ten training iterations. Plots in the top (bottom) row show the task runtimes on each node without (with) preemption. This type of diagram is described in Section 2.5.2.1.*

	Higgs			Criteo		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	186ms	314ms	152ms	1834ms	3177ms	1651ms
mean task time	113ms	123ms	108ms	863ms	968ms	840ms
task time std. dev.	15ms	42ms	5ms	92ms	387ms	35ms

	KDDA			Webspam		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	1336ms	1358ms	1346ms	1804ms	2298ms	1819ms
mean task time	131ms	144ms	126ms	481ms	527ms	411ms
task time std. dev.	14ms	51ms	6ms	110ms	221ms	20ms

(a) CoCoA

	CIFAR-10			Fashion-MNIST		
	baseline	**w/o**	**w/**	**baseline**	**w/o**	**w/**
mean iter. time	136ms	185ms	105ms	166ms	213ms	118ms
mean task time	73ms	78ms	65ms	88ms	92ms	75ms
task time std. dev.	17ms	26ms	6ms	24ms	32ms	7ms

(b) lSGD

Table 5.19: *Summary of results (mean iteration and task runtime, as well as standard deviation of the latter) for the straggler experiments. Each dataset was trained with (w/) and without (w/o) task preemption and compared to the baseline, without stragglers.*

5.7.2.2 Epochs to converge

Figure 5.17 shows the impact of task preemption on the convergence behavior of CoCoA and lSGD. Table 5.20 lists the number of epochs needed to converge to the training target. With task preemption, the training target is reached on average with ≈2.7% fewer epochs than without. While these differences are small enough to be the result of the stochastic nature of the algorithms, where small variations can occur, another explanation is also possible: With task preemption, the number of training samples processed during each iteration is reduced and the frequency of global communication therefore increased. This has a similar effect as reducing the number of partitions for CoCoA and the global batch size in lSGD. Both can reduce the number of epochs needed to converge.

	baseline		without preemption		with preemption	
Dataset (target)	**epochs**	**runtime**	**epochs**	**runtime**	**epochs**	**runtime**
Higgs (1e-8)	1471	276.2s	–	–	1438	221.1s
Criteo (1e-8)	54	102.3s	56	178.5s	54	90.8s
KDDA (1e-5)	68	97.8s	67	98.1s	68	98.2s
Webspam (1e-6)	46	86.8s	48	115.1s	44	84.0s

(a) CoCoA

	baseline		without preemption		with preemption	
Dataset (target)	**epochs**	**runtime**	**epochs**	**runtime**	**epochs**	**runtime**
CIFAR-10 (61%)	27.13	68.9s	27.12	100.6s	27.72	57.7s
F.-MNIST (89%)	17.11	68.9s	17.11	94.4s	16.31	50.6s

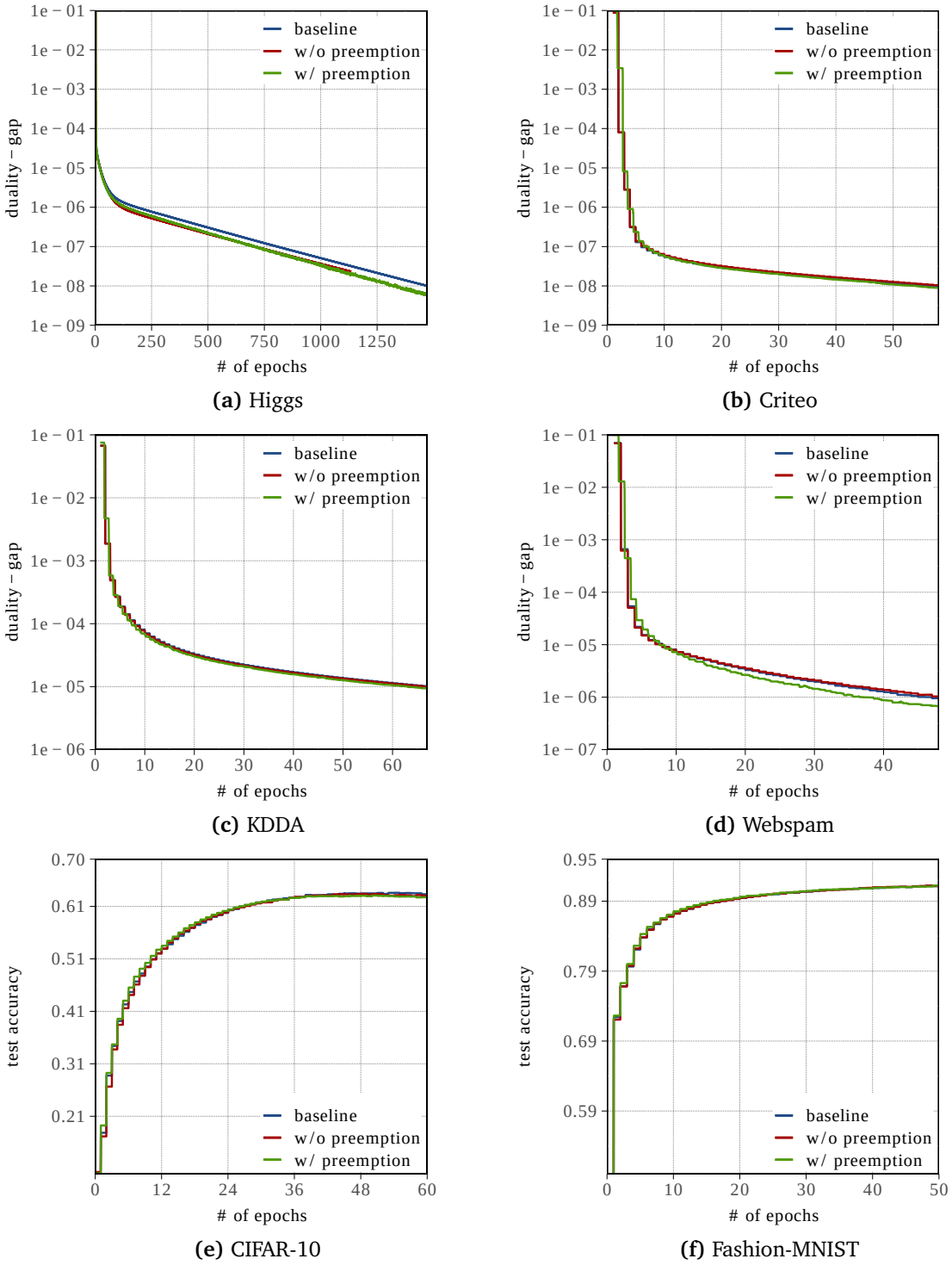
(b) lSGD

Table 5.20: *Number of epochs and time to converge to the target duality-gap and test accuracy respectively. "–" indicates that the target has not been reached within the test time limit.*

However, in SSP, a commonly used method to mitigate stragglers, the number of epochs needed to converge generally increases in face of stragglers [34].[10] The reason for this difference lies in the methods themselves:

- With task preemption, the effective batch size is reduced, as preempted tasks process fewer than planned training samples per iteration, after which all tasks synchronize. At the same time, the task preemption method guarantees that all tasks receive all model updates before starting the next iteration.
- In SSP, the effective batch size may be reduced as well (as updates from straggling tasks are not considered immediately) - which generally supports faster convergence per epoch. In contrast to task preemption, however, tasks are not guaranteed to receive all model updates before starting the next iteration. Hence, some tasks work based on an outdated (stale) model and thus making training less effective per processed training sample.

[10] Ho et al. [34] report an increase in the number of iterations needed to converge, not epochs. However, in SSP, the number of training samples processed in each iteration remains constant, hence the number of epochs increases with the number of iterations.



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

(e) CIFAR-10 **(f)** Fashion-MNIST

Figure 5.17: *Convergence of the duality-gap (lower is better) and test accuracy (higher is better)* ***over epochs*** *in a straggler scenario.* Baseline *is without stragglers,* w/o preemption *with stragglers but without mitigation and* w/ preemption *with stragglers and mitigation.*

5.7.2.3 Time to converge

Figure 5.18 shows the convergence over time for each dataset. Here, a noticeable benefit of task preemption can be observed for all datasets, except for KDDA. The iteration runtime for KDDA is dominated by communication, hence mitigating stragglers has little effect and the overall training time remains virtually constant, as the data in Table 5.19b shows. On average, a training time reduction of 1.62×, compared to without task preemption, is achieved. Compared to the baseline, training times reduce by 1.17×.

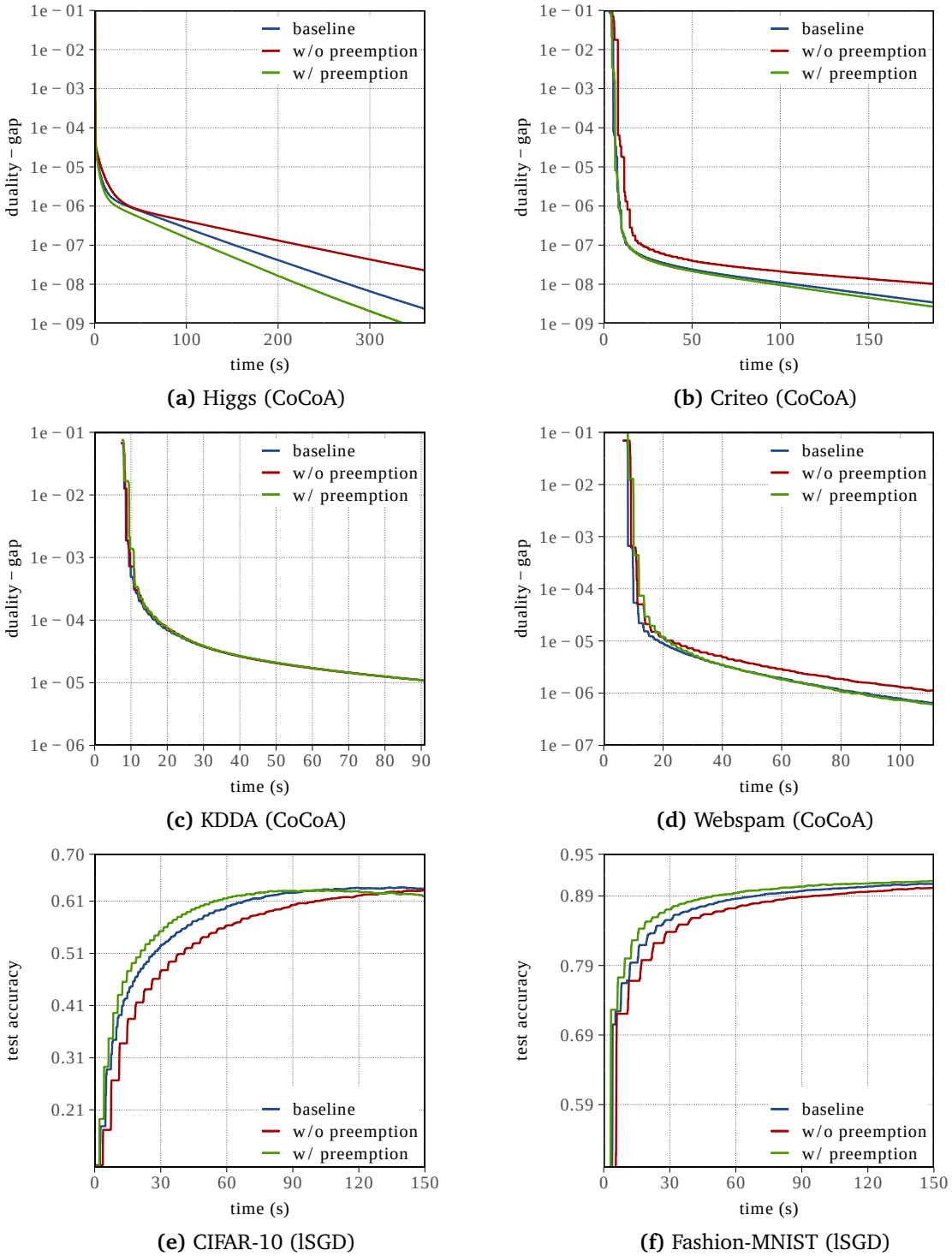
5.7.2.4 Evaluation summary

Table 5.21 lists the average maximal test accuracy for lSGD. With task preemption, it is reduced by 0.37% on average, compared to without task preemption.

Datasets	w/ preemption	w/o preemption
CIFAR-10	63.17%	<u>63.60%</u>
Fashion-MNIST	91.27%	<u>91.33%</u>

Table 5.21: *Average maximal test accuracy (higher is better) for the straggler experiment. The highest test accuracy for each dataset is underlined.*

Overall, this evaluation has shown that task preemption is an effective method to mitigate the impact of stragglers with uni-tasks. On average, the number of epochs needed to converge remains virtually unchanged but training time is cut short by 1.62×.



(a) Higgs (CoCoA) **(b)** Criteo (CoCoA)

(c) KDDA (CoCoA) **(d)** Webspam (CoCoA)

(e) CIFAR-10 (lSGD) **(f)** Fashion-MNIST (lSGD)

Figure 5.18: *Convergence of the duality-gap (lower is better) and test accuracy (higher is better)* ***over time*** *in a straggler scenario.* Baseline *is without stragglers,* w/o preemption *with stragglers but without mitigation and* w/ preemption *with stragglers and mitigation.*

5.8 Comparison with state-of-the-art frameworks

In this section, Chicle is compared with two state-of-the-art distributed ML frameworks:

(1) Snap ML [103], a rigid, state-of-the-art, distributed ML training framework, using the CoCoA training algorithm.

(2) PyTorch [85], a rigid, state-of-the-art, distributed ML training framework, using the Mini-batch SGD (mSGD) training algorithm.

Like Chicle, both frameworks use RDMA for communication. The purpose of this experiment is to show that Chicle, as the first implementation of uni-tasks, performs competitively to state-of-the-art frameworks. As neither of them is elastic, supports load balancing nor natively implements straggler mitigation techniques[11], only baseline tests are presented here. Litz [114], the work that is most closely related to Chicle was not publicly available at the time of writing and could not be compared against.

All experiments use the test setup and datasets described in Section A.3.3. The same training algorithm parameters were used in Chicle and the compared-to framework.

5.8.1 Comparison with Snap ML

This section presents a comparison of Chicle with Snap ML for the CoCoA training algorithm. For CoCoA, data partitioning can significantly impact convergence. Chicle and Snap ML use different data partitioning schemes:

- Chicle splits up the training data into small (here: 2MiB) chunks and assigns them to nodes in a random order, such that each node gets the same number ±1 of chunks. This results in C chunks.
- Snap ML splits the training data into K equally large partitions for K nodes and assigns one partition per node.

As both implement the same training algorithm and use the same parameters for training, convergence per epoch should be identical in both cases, provided an identical data partitioning and an identical training sample processing order on each local solver. Neither is the case here, hence differences are to be expected. In general, it is not possible to predict how data partitioning (given the same number of partitions) and training sample processing order impact convergence.

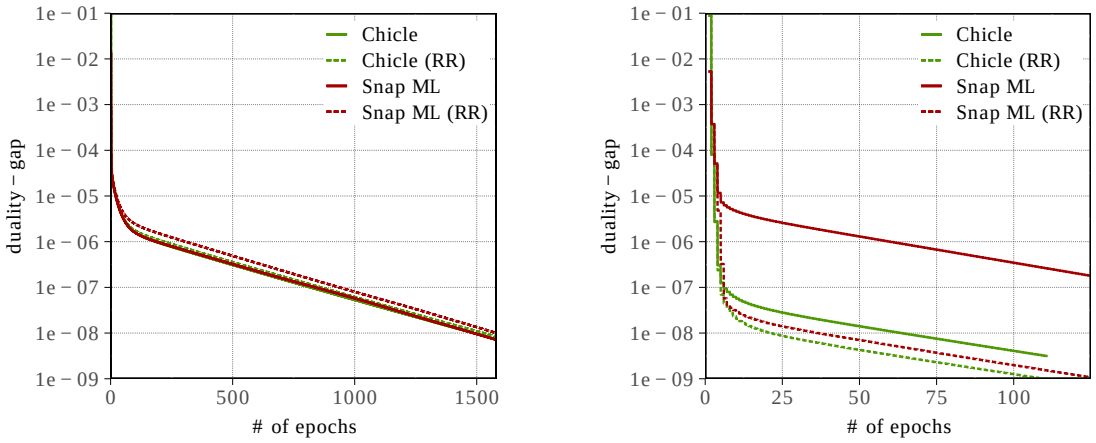
An attempt has been made to reduce these differences, to show that uni-tasks does not inherently change the convergence per epoch from that of Snap ML. Due to the different implementations of both frameworks, it is not possible to completely align both aspects. However, in order to reduce differences, Snap ML has been modified to partition training

[11] PyTorch provides functionality to implement SSP. As part of this work, SSP has not been implemented on top of PyTorch.

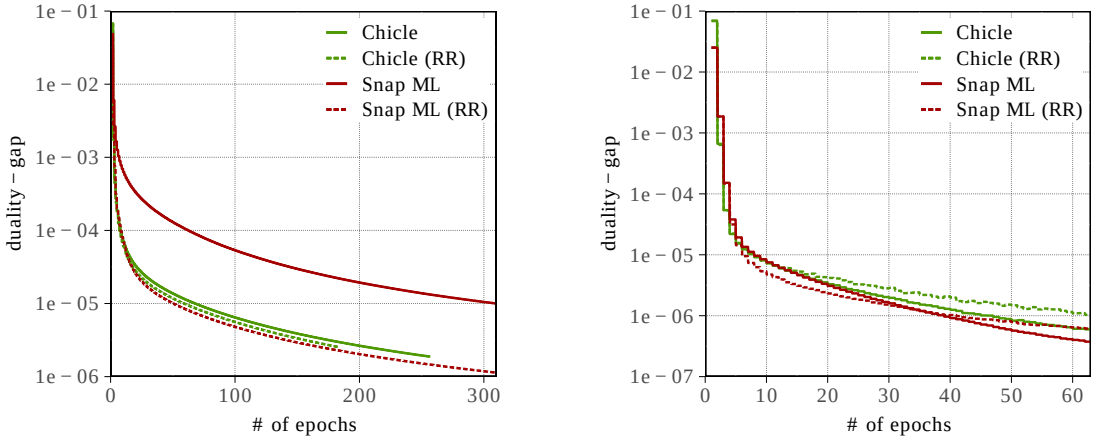
data, similar to Chicle, into C chunks and assign them to nodes in a round-robin fashion (Snap ML (RR)). Chicle has been modified to assign chunks not randomly but also in a round-robin fashion (Chicle (RR)). This approximates both data partitioning schemes, but it doesn't make them identical, as Chicle's data loader is chunk size based, whereas the modified Snap ML data loader is number of partitions based. For each dataset, C was set to match the number of chunks used in Chicle.

Furthermore, Snap ML's local SCD solver can use multiple threads to speed up training. For Chicle, a multi-threaded local solver has not been implemented yet, as it is not considered necessary to demonstrate the viability of uni-tasks. For this reason, only a single thread for each solver instance was used for Snap ML in this evaluation.

Figure 5.20 shows the results of these experiments which are summarized Table 5.22. Runtime includes time to compute the duality-gap in both cases.

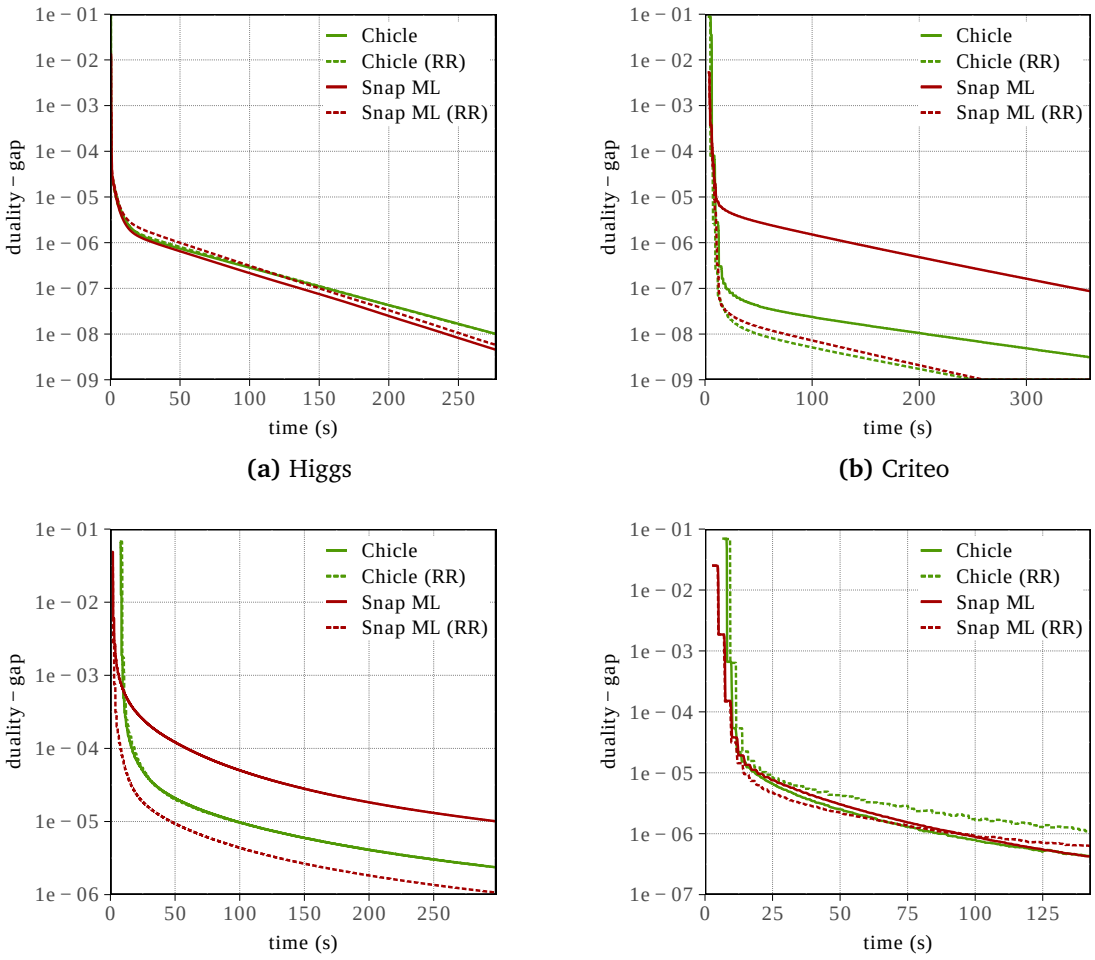


(a) Higgs **(b)** Criteo



(c) KDDA **(d)** Webspam

Figure 5.19: *Comparison of CoCoA on Chicle with Snap ML w.r.t.* ***epochs*** *to convergence.*



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

Figure 5.20: *Comparison of CoCoA on Chicle with Snap ML w.r.t.* ***time*** *to convergence.*

	Chicle		Chicle (RR)		Snap ML		Snap ML (RR)	
Dataset (target)	**epochs**	**time**	**epochs**	**time**	**epochs**	**time**	**epochs**	**time**
Higgs (1e-8)	1474	275s	1528	277s	1486	241s	1582	256s
Criteo (1e-8)	57	185s	21	49s	236	539s	35	72s
KDDA (1e-5)	68	98s	59	95s	310	299s	50	47s
Webspam (1e-6)	46	87s	64	150s	39	94s	42	98s

(a) Absolute results

	Snap ML		Snap ML (RR)	
Dataset (target)	**epochs**	**time**	**epochs**	**time**
Higgs (1e-8)	1.01×	0.88×	1.04×	0.92×
Criteo (1e-8)	4.14×	2.60×	1.67×	1.58×
KDDA (1e-5)	4.54×	2.89×	0.85×	0.52×
Webspam (1e-6)	0.85×	1.06×	0.66×	0.65×

(b) Relative to Chicle

Table 5.22: *Absolute (a) and relative (b) number of epochs and time (r/t) to converge (lower is better) for Chicle and Snap ML. A relative result* > 1 *means that Chicle is faster than Snap ML.*

Results show that convergence per epoch is similar in all four cases for the Higgs and Webspam datasets, indicating that for those datasets, data partitioning is less important. For Criteo and KDDA, however, significant differences in the convergence per epoch can be observed. These differences reduce when comparing Snap ML (RR) to Chicle and Chicle (RR), indicating the importance of data partitioning for both datasets.

In general, results for Snap ML (RR) are similar to those of Chicle and Chicle (RR), hence no clear advantage of disadvantage of either framework can be identified, which confirms the initial expectation. Remaining differences are due slight differences in data partitioning as well as the different training sample processing order.

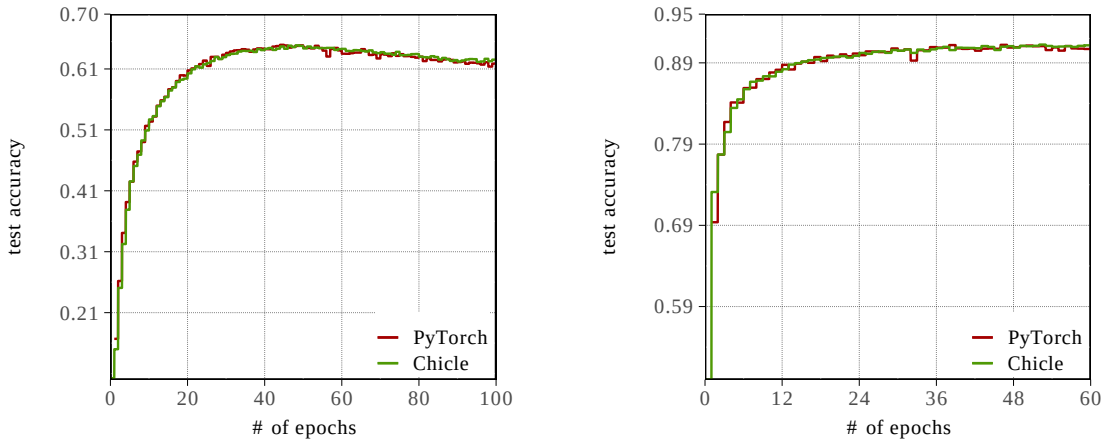
Table 5.21b shows that the speedup of Chicle over Snap ML is larger per epoch than over time, thus an epoch takes more time in Chicle than on Snap ML. As in these experiments, no data chunks are moved during training, differences in time per epoch are rooted in the communication subsystem, i.e., the lack of efficient reduce and broadcast operations, as well as in the CoCoA implementation, more specifically the local SCD solver, on Chicle.

Overall, Chicle performs better than Snap ML and similar to Snap ML (RR) in a baseline scenario.

5.8.2 Comparison with PyTorch

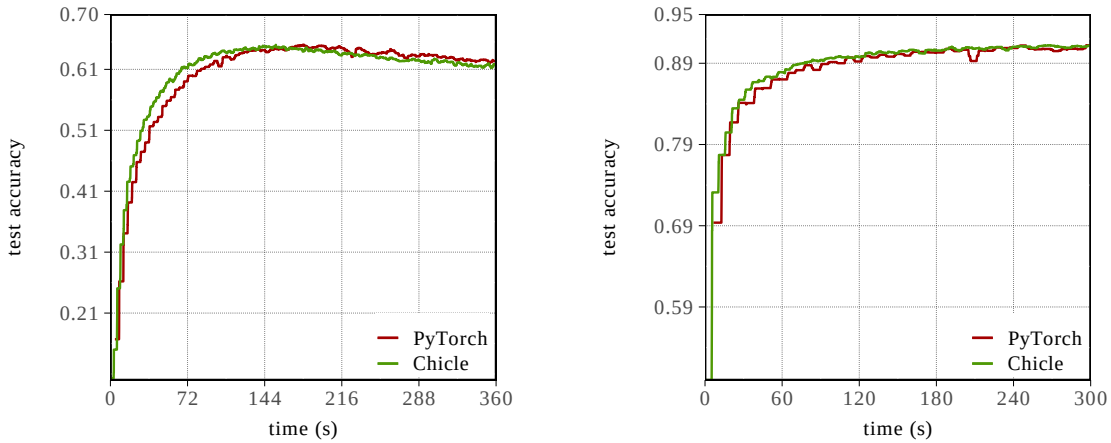
For the comparison with PyTorch, the same training algorithm parameters as for previous tests are used (Table 5.6), except that $H = 1$, i.e., only a single local round of training is performed per iteration. The built-in PyTorch distributed training functions[12] do not allow the training with $H > 1$, as tasks synchronize implicitly after training on L samples. For that reason, the mSGD algorithm has been evaluated with $H = 1$ on Chicle and PyTorch.

Figures 5.21 and 5.22 show the convergence plots which are summarized in Tables 5.23 and 5.24. Runtime does not include the computation of the test accuracy in either case.



(a) CIFAR-10 **(b)** Fashion-MNIST

Figure 5.21: *Convergence over **epochs** (higher is better) of mSGD on Chicle and PyTorch.*



(a) CIFAR-10 **(b)** Fashion-MNIST

Figure 5.22: *Convergence over **time** (higher is better) of mSGD on Chicle and PyTorch.*

[12] `torch.nn.parallel.distributed_cpu.DistributedDataParallelCPU module`

The convergence over epoch plots show a very similar convergence behavior of Chicle and PyTorch. The test accuracy targets are also reached in virtually the same number of epochs. This is not surprising, as both use libtorch, which performs the actual training on each task.

Convergence over time, is up to 1.30× faster on Chicle than on PyTorch. As both use libtorch for training and RDMA for communication, the differences are likely due to overheads incurred by Python: Chicle is written in C++ and PyTorch in Python. While PyTorch also uses the libtorch C++ code for the actual training, data exchange between Python and C++ adds some overhead.

Chicle as well as PyTorch exhibit the same behavior for the CIFAR-10 dataset, where test accuracy decreases after reaching a peak. This behavior has been observed for CIFAR-10 throughout the evaluation of Chicle and uni-tasks. Mitigation strategies include learning rate and batch size adjustment during the training.

	Chicle		**PyTorch**		**Speedup**	
Dataset (target)	**epochs**	**runtime**	**epochs**	**runtime**	**epochs**	**runtime**
CIFAR-10 (61%)	21.6	66.2s	21.4	86.0s	0.99×	1.30×
Fashion-MNIST (89%)	14.0	72.5s	14.6	93.8s	1.04×	1.29×

Table 5.23: *Number of epochs and time to converge (lower is better) as well as relative speedup of Chicle over PyTorch.*

Maximal test accuracy is also very close in both cases (Table 5.24).

Dataset	**Chicle**	**PyTorch**
CIFAR-10	65.16%	<u>65.19%</u>
Fashion-MNIST	<u>91.42%</u>	91.28%

Table 5.24: *Average maximal test accuracy (higher is better). The highest value is <u>underlined</u>.*

Overall, Chicle compares competitively against PyTorch, demonstrating that uni-tasks does not slow down mSGD training in a baseline scenario. The latter is important, as the benefit on heterogeneous, shared and straggler-afflicted scenarios should not come at a cost of the *normal* (baseline) case.

Observed differences in the number of epochs to converge and maximal test accuracy are not the result of targeted improvements in Chicle but of different data partitioning schemes as well as a different training sample processing order.

5.9 CoCoA-specific optimizations enabled by uni-tasks

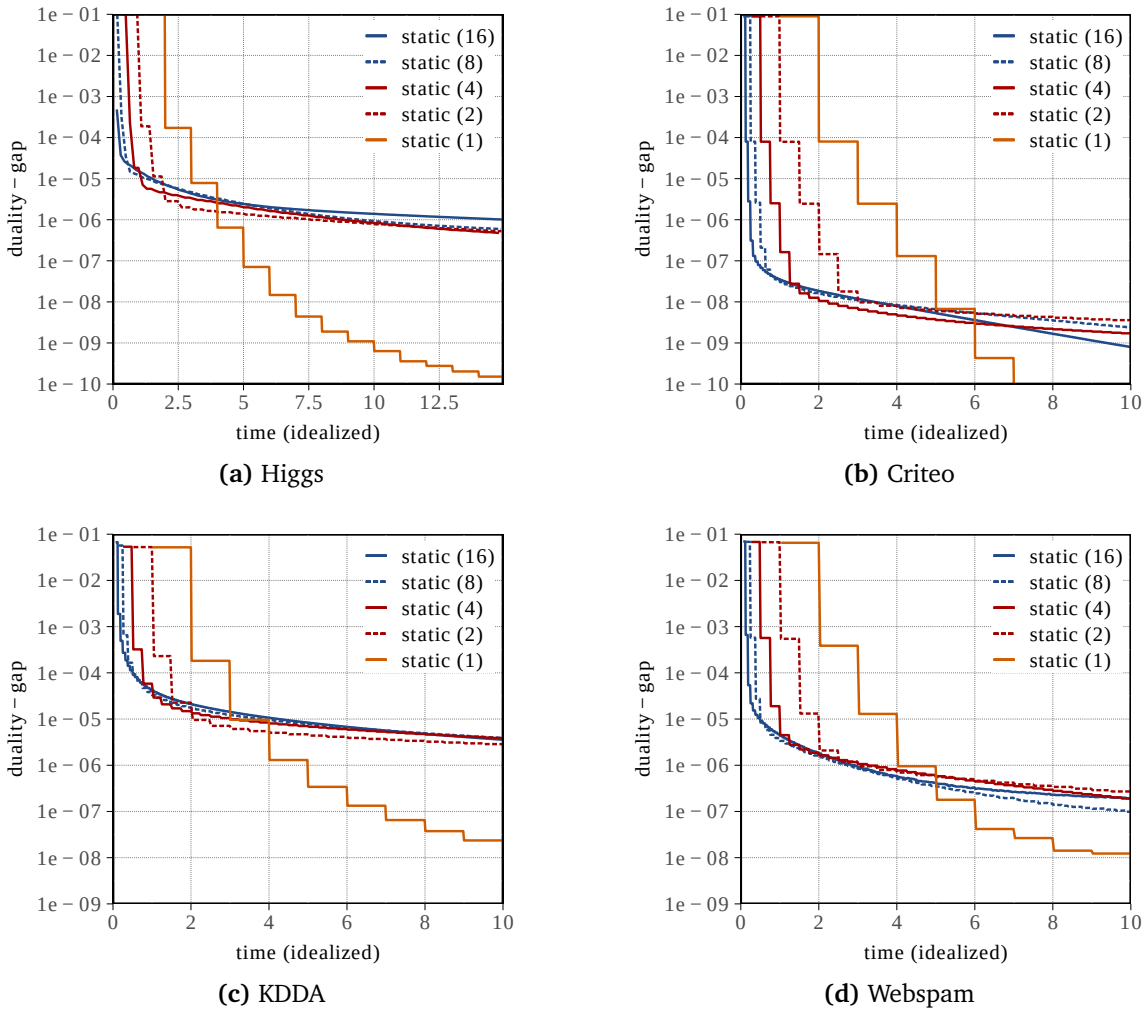
This section presents two approaches of how to exploit CoCoA-specific properties with uni-tasks:

- *Resource requirements of CoCoA are heterogeneous*: The number of nodes that result in the highest convergence rate changes over the course of the training. With uni-tasks, this heterogeneity can be exploited to reduce time to convergence and overall resource utilization.
- After detecting all correlations across all local chunks, convergence rate drops significantly. This can be prevented by exchanging a subset of chunks in each epoch and is enabled by uni-tasks to access all local data chunks but also to move them across tasks in-between iterations. This shows that uni-tasks can have benefits beyond scheduling.

5.9.1 Auto scale-in policy for an increased convergence rate

The convergence rate per epoch is inversely correlated to the number of nodes (and therefore data partitions), i.e., the more nodes, the slower the convergence per epoch, as already shown in Figure 5.1. Up until a dataset-specific point, the number of epochs increases slower than the number of nodes and a net-benefit can be achieved by using more nodes. Hence, a number of nodes should exist, where the increase in parallelism due to an increase in nodes just outweighs the increase in the number of epochs required to converge. A peculiarity of CoCoA, which is exploited by this policy, is that this number is not constant but shifts towards fewer nodes during the training.

CoCoA's stateful local solvers (e.g. SCD), update their local model after each training sample. This allows them to efficiently discover correlations between training samples locally. At some point in time, however, all local correlations will have been discovered. From this point on, further discoveries can only be made by propagating correlation information via the global model. As this requires global communication, which occurs orders of magnitude less often than local updates, it is less efficient than local discoveries. As result, convergence rate per epoch, as well as per time, drops noticeably. As with more local data, more correlations can be discovered locally and this drop occurs at later epochs, where more correlations have already been discovered and the duality-gap is lower.



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

Figure 5.23: *Example of the convergence of the duality-gap (lower is better) using 1 – 16 nodes (node count in parentheses) and the same number of partitions over (hardware-independent) idealized time: 1 partition requires 1 time unit per iteration, K partitions require* $1/K$ *time units per iteration, assuming parallel execution. This plots therefore shows the non-linear correlation between the number of partitions and the duality-gap over different phases of the training process.*

This effect can be observed, to varying degrees, in all datasets shown in Figure 5.23. Here, the duality-gap is plotted against an idealized time (assuming zero scheduling overheads and perfect scaling) for one to 16 nodes. In the beginning, convergence is generally faster with more nodes than with fewer nodes (due to the higher level of parallelism). However, the convergence curve also flattens out sooner, i.e., at larger duality-gaps, for more nodes than for fewer nodes, despite the advantage of a higher level of parallelism. The region that separates high convergence rates from low convergence rates is referred to as the *knee*.

For instance, in Figure 5.23a the highest convergence rate in the beginning can be achieved with 16 nodes but it also starts to flatten earliest, between a duality-gap of 1e-4 and 1e-5. At this point, the convergence rate curve for four nodes is still steeper, and only flattens just below 1e-5. If the training algorithm switched from 16 to four nodes once the curve flattens, the convergence rate could remain high, until the slow-down occurs here as well, at which point training could be scaled-in further, e.g., to two or one node(s). For all other datasets, a similar convergence behavior can be observed, except the knee occurs at different duality-gap values.

For all datasets, one node is a special case. Here, all correlations can be discovered locally, throughout the entire training process and no knee will ever occur.

5.9.1.1 Auto-scale-in approach

The idea of this policy is to exploit the fact that the knee occurs at lower duality-gaps for a smaller number of nodes [106]. Hence, if a knee occurs for some number of nodes a, it will not yet have occurred for a smaller number of nodes $b < a$. By scaling in, and therefore giving each local solver additional training data, it can again detect correlations locally, thus staying in-front of the knee[13].

For this to work, tasks need to be able to access all node-local training data, hence this policy could not be realized with micro-tasks, as here, even if nodes are removed, the same number of tasks is merely executed on fewer nodes, but tasks do not gain access to more training samples.

Knee detection. In order to detect the knee, the convergence rate of the duality-gap over time is observed. Once the convergence rate drops, the policy assumes that a knee has occurred and training is scaled in to fewer nodes.

Figure 5.24 visualizes the method used to identify the knee. Using the difference between two slopes, a long-term slope S_l and a short-term slope S_s. S_l represents to convergence since time t_0, which is either the beginning of the training or time of the last scale-in action (whichever occurred later), and now (t_n), and is computed as in Equation 5.7. Here, v_0 represent the duality-gap at time t_0 and v_n represent the duality-gap at time t_n.

$$S_l = \frac{\log_{10}(v_n) - \log_{10}(v_o)}{t_n - t_0} \tag{5.7}$$

The short-term slope S_s represents the convergence over the last N iterations, i.e., since time t_{n-N}, and now and is computed as in Equation 5.8. Here, v_{n-N} represent the duality-gap at time t_{n-N}.

$$S_s = \frac{\log_{10}(v_n) - \log_{10}(v_{n-N})}{t_n - t_{n-N}} \tag{5.8}$$

[13] A US patent application, that describes this idea, has been filed.

Once the short term slope S_s diverges significantly from the long term slope S_l, i.e., once $S_s \times d < S_l$, with d being a damping factor, a knee has been detected and a scale-in action is initiated. Upon scale-in, the number of workers K is reduced by a factor m. A description of all parameters is in Table 5.25.

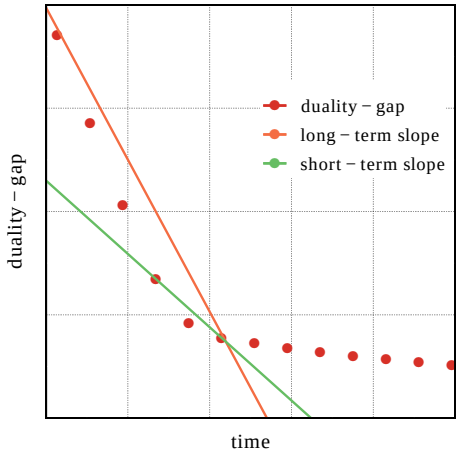


Figure 5.24: *Schematic view of the long-/short-term slope of the duality-gap that is used to identify the* knee. *The y-axis is* $\log_{10}$ *scaled.*

Parameter	Description
N	Number of iterations to compute the short-term slope S_s over. A smaller value leads to a faster, but less reliable *knee* identification as the impact of potential occasional stragglers is relatively higher for smaller values of N. Larger values of N lead to a slower but more robust *knee* detection.
d	A damping factor that determines the magnitude by which the short term and long term slops need to diverge before a scale-in action is triggered. A small d (e.g., $d = 1.0$) can falsely trigger the scale-in action because of jitter in the duality-gap, whereas a large d (e.g., $d = 5.0$) can delay or event prevent the triggering a scale-in action.
m	The scale-in factor. During scale-in all but $1/m$-th of nodes are removed and $(m-1)/m$ of all data chunks have to be moved to the remaining nodes, which represents a significant overhead. Scaling in in many small steps may therefore be overall less beneficial than scaling in using fewer large steps.
K_{min}	The minimal number of nodes to scale-in to.

Table 5.25: *Overview of parameters of the scale-in policy. Concrete values for these parameters are given in the corresponding evaluation section.*

Using the auto scale-in policy, training can be accelerated while, at the same time, the number of allocated resources is reduced, i.e., more *effective work* can be accomplished using fewer resources, thus increasing resource utilization efficiency. A limitation of this approach is that the memory capacity of the remaining nodes has to be sufficiently large to hold all data chunks.

5.9.1.2 Auto scale-in in Chicle

The auto scale-in policy implements the method described above. As an additional hardware-heterogeneity-aware improvement, it removes nodes upon scale-in based on their performance: Slow nodes are removed first. Node performance is determined as in the rebalance policy (Section 5.5.1). While the latter is not necessary to exploit the algorithmic benefits of scaling in, it exploits available knowledge about the system to speed up training further on heterogeneous clusters. The source code of this policy is shown in Listing 5.6. The `decide` function is called upon the reception of an *iteration finished* event.

```
long itSinceSI = 0;    // iterations since last scale-in
list<Worker*> workers; // 'workers' is a list with all currently active workers.

// 'dg' is the log10-scaled duality gap value after iteration 'it' at time 'ts'
void decide(double dg, long ts, long it)
{
  if (itSinceSI == 0)
    lastSI = pair(ts, dg); // (t_0, v_0)
  itSinceSI++;

  // 'history' is a ring buffer with maximal size N that contains (timestap, duality gap)
  // pairs for the last N iterations.
  // - 'history.first' always points to the oldest value.
  // - 'history.size' contains the current number of elements in the buffer 0...N
  history.push_back(pair(ts, dg))

  // Compute slopes of long- and short-term gradients
  double S_l = (dg - lastSI.dg) / (ts - lastSI.ts);
  double S_s = (dg - history.first.dg) / (ts - history.first.ts);

  // Detect knee and scale-in by a factor of m, if it is detected.
  if (S_s * d > S_l && itSinceSI >= N && workers.size > K_min) {
    // Sort the 'workers' list in descending order of their median runtime τ_k as in the
    // rebalance policy (see Section 5.5.1 for details).
    sort(workers);

    // Identify (number of) workers that need to be removed.
    int remove = workers.size - max(K_min, (workers.size / m));
    list<Worker*> removeWorkers;

    while (remove > 0) {
      removeWorkers.push_back(workers.first);
      workers.pop_first;
      remove--;
    }

    scalein(workers, removeWorkers); // call 'scalein' function of the 'elasticity' policy
                                     // (see Section 5.6.1 for details).
    itSinceSI = 0;
  }
}
```

Listing 5.6: *Simplified C++ code of the scale-in policy's decision algorithm.*

5.9.1.3 Evaluation

This section presents evaluation results for the auto-scale-in policy. The purpose of these experiments is to show the reduction in the number of epochs and training time compared to a static setting. Two scenarios have been tested:

(1) Assume that all training data fits inside the memory of a single node ($K_{min} = 1$).

(2) Assume that all training data fits inside the memory of a two nodes ($K_{min} = 2$). The purpose of this scenario is to show the benefits of scale-in to multiple nodes, as the convergence behavior on a single node differs significantly from that of two or more nodes.

All experiments use the test setup and datasets described in Section A.3.3. Parameter values for the auto-scale-in policy are listed in Table 5.26. Parameters for CoCoA are the same as in Section 5.5.2.2. Table 5.27 shows the data volume that needs to be transferred during scale-in.

Parameter	**Value**
N	2
d	1.25
m	4
K_{min}	1, 2 (depending on scenario)

Table 5.26: *Chicle's auto-scale-in policy parameters used in this evaluation.*

Convergence plots per epoch and over time are shown in Figures 5.25 and 5.26 and summarized in Table 5.28. The convergence plots show that the scale-in policy, by reducing the number of nodes (and therefore data partitions) based on feedback from the training algorithm, improves convergence rate over time in most cases. The specific improvement depends on the target duality-gap. Aside from improving the convergence rate, the smallest, within the test time limit achievable duality-gap, also decreases significantly.

In 13 out of 16 cases, auto scale-in to two nodes improves time to convergence by up to 7.22×. In the other three cases, the best static node count takes at least 0.61× as much time as auto scale-in. When scaling in to a single node, auto scale-in improves time to convergence in 14 out of 20 cases by up to 2.94×. In the other cases, the best static node count takes at least 0.51× as much time as auto scale-in. However, the best static node count varies and is generally not known in advance. Multiple trial runs would be necessary to identify the best static node count, which also varies by target duality-gap and dataset. Moreover, in cases where auto scale-in is not the fastest, it is the second

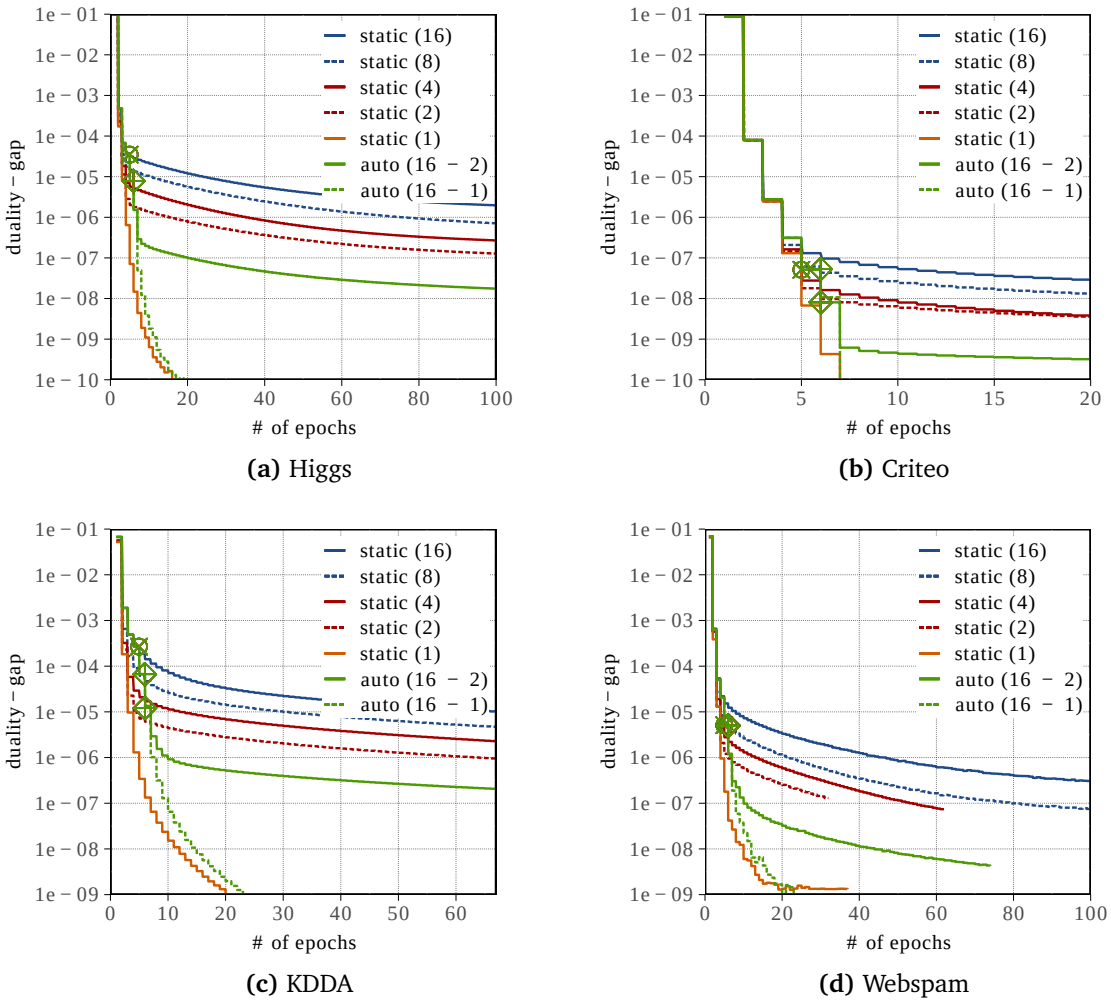
fastest in all but one case (KDDA, 1e-5). KDDA is a special case, as even in the beginning, 16 nodes converge slower over time than any other node count. This is caused by the large model update vector in KDDA, which incurs high relative communication overheads. As result, the initial convergence rate over time for auto scale-in is lower than any static node count with less than 16 nodes. However, upon scale-in, the auto scale-in policy catches up. This is due to the hardware-heterogeneity-awareness that removes the slowest nodes first during scale-in.

For each dataset, there is a brief period where scaling in to two nodes results in faster convergence than when scaling in to one node. This is caused by the longer epoch duration on one node compared to two nodes. However, as the convergence rate per epoch is higher on one node than on two, the former catches up and overtakes the latter after said brief period. This suggests that adding another step between four and one nodes accelerate training further. However, for the auto scale-in policy to detect a knee, multiple iterations need to pass, hence the delay to scale-in to one node may nullify the benefits of shorter epoch duration. Due to the expected small benefits, this has not been implemented.

5.9.1.3.1 Overheads. Scaling in incurs data transfer overheads. Table 5.27 lists the amount of data that needs to be transferred when scaling in. The time given in Table 5.28 includes the data transfer time on the 56Gbps Infiniband network of the test cluster, hence on this network, benefits outweigh the costs. On slower networks, this might not be the case. However, slower networks would also benefit from the reduced reduce and broadcast time after scaling in, especially for the KDDA and Webspam datasets.

Datasets	**16 → 4**	**4 → 2**	**4 → 1**
Higgs	0.2 GiB / 0.6 GiB	0.6 GiB / 1.2 GiB	0.6 GiB / 2.4 GiB
Criteo	1.2 GiB / 3.6 GiB	3.6 GiB / 7.2 GiB	3.6 GiB / 14.3 GiB
KDDA	0.2 GiB / 0.6 GiB	0.6 GiB / 1.2 GiB	0.6 GiB / 2.5 GiB
Webspam	0.8 GiB / 2.5 GiB	2.5 GiB / 4.9 GiB	2.5 GiB / 9.8 GiB

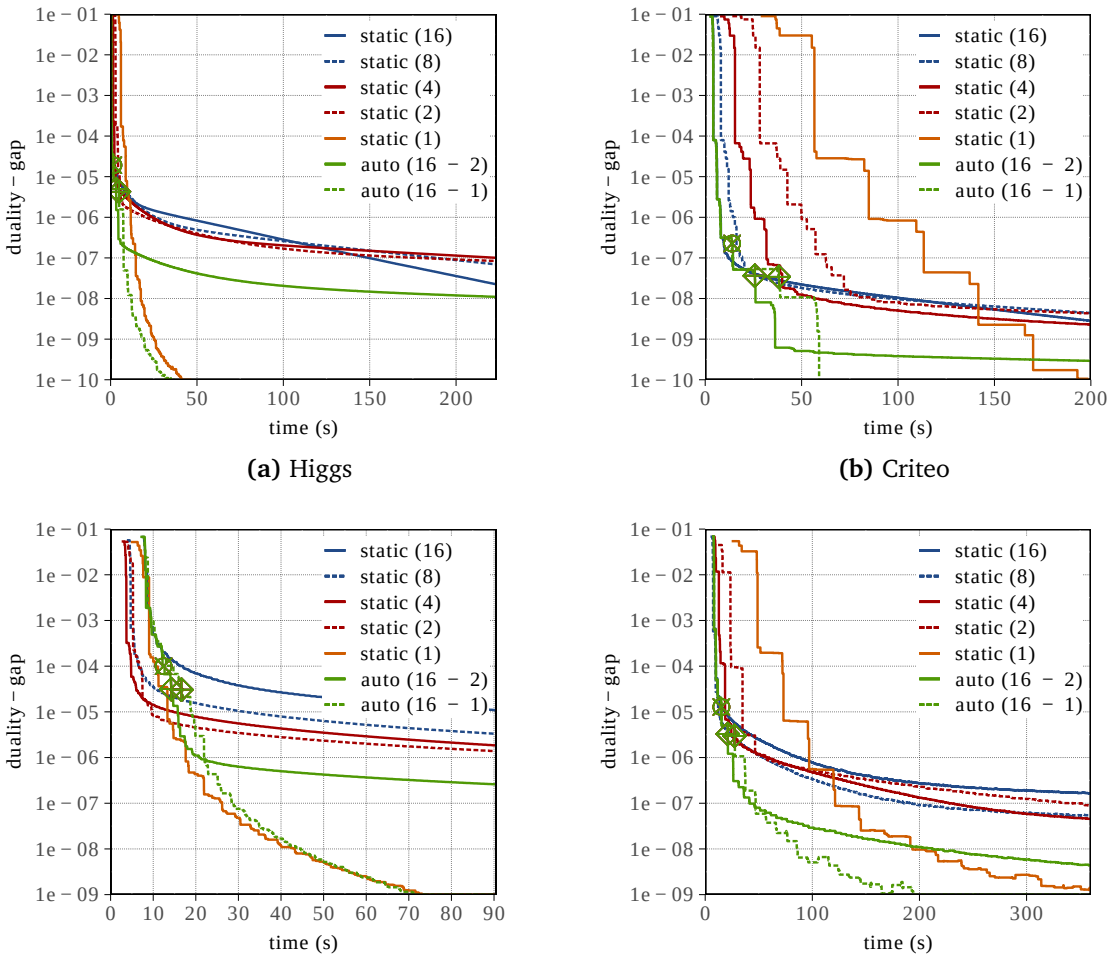
Table 5.27: *Data transfer volumes during scale-in for each sending/receiving node.*



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

Figure 5.25: *Duality-gap vs.* ***epochs*** *plots for the evaluated datasets and settings. Circles depict a scale-in from 16 to 4 workers, diamonds a scale-in from 4 to 2 and 1 worker(s) respectively.*



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

Figure 5.26: *Duality-gap vs.* ***time*** *plots for the evaluated datasets and settings. Circles depict a scale-in from 16 to 4 workers, diamonds a scale-in from 4 to 2 and 1 worker(s) respectively.*

run	1e-05	1e-06	1e-07	1e-08	1e-09
static (16)	4.1s	41.1s	149.3s	264.3s	–
static (8)	3.6s	26.1s	188.5s	–	–
static (4)	2.9s	24.1s	224.0s	–	–
static (2)	5.2s	20.7s	177.3s	–	–
static (1)	8.7s	11.7s	14.6s	17.5s	26.3s
auto (16 – 2)	**1.6s**	**4.4s**	**20.7s**	**253.1s**	–
auto (16 – 1)	1.6s	7.3s	7.5s	12.1s	19.9s
speedup (16 – 2)	1.79× (4)	4.67× (2)	7.22× (16)	1.04× (16)	–
speedup (16 – 1)	1.79× (4)	2.63× (1)	1.93× (1)	1.45× (1)	1.32× (1)

(a) Higgs

run	1e-05	1e-06	1e-07	1e-08	1e-09
static (16)	6.2s	8.1s	**12.1s**	103.0s	299.5s
static (8)	12.2s	15.4s	19.2s	96.6s	–
static (4)	23.6s	25.7s	31.8s	57.1s	–
static (2)	42.6s	50.0s	62.6s	86.5s	–
static (1)	84.8s	84.8s	113.3s	141.7s	170.1s
auto (16 – 2)	**6.1s**	**7.9s**	14.4s	**26.1s**	**66.8s**
auto (16 – 1)	6.1s	7.8s	14.0s	56.5s	59.1s
speedup (16 – 2)	1.01× (16)	1.03× (16)	0.84× (16)	2.19× (4)	4.48× (16)
speedup (16 – 1)	1.02× (16)	1.03× (16)	0.86× (16)	1.01× (4)	2.88× (1)

(b) Criteo

Table 5.28: *Time to converge to duality-gap 1e-5 – 1e-9 for each dataset and speedup of auto scale-in compared to the best static node count larger than the minimum (1 or 2). The best static node count is given in parentheses. The fastest run for 1 or more nodes is underlined and* ***bold*** *for 2 or more nodes. "–" indicates that the respective duality-gap was not reached within the test time limit.*

run	1e-05	1e-06	1e-07	1e-08	1e-09
static (16)	97.7s	–	–	–	–
static (8)	31.6s	238.3s	–	–	–
static (4)	14.8s	148.2s	–	–	–
static (2)	**9.7s**	124.9s	–	–	–
static (1)	13.3s	17.6s	26.2s	43.3s	90.1s
auto (16 – 2)	15.8s	**20.9s**	**241.6s**	–	–
auto (16 – 1)	19.0s	21.9s	28.9s	44.7s	77.6s
speedup (16 – 2)	0.61× (2)	5.98× (2)	>1.49× (-)	–	–
speedup (16 – 1)	0.51× (2)	0.80× (1)	0.91× (1)	0.97× (1)	1.16× (1)

(c) KDDA

run	1e-05	1e-06	1e-07	1e-08	1e-09
static (16)	18.7s	86.4s	–	–	–
static (8)	**12.2s**	53.7s	185.9s	–	–
static (4)	18.5s	59.4s	227.8s	–	–
static (2)	34.9s	57.9s	343.4s	–	–
static (1)	73.1s	97.0s	121.2s	193.5s	–
auto (16 – 2)	15.1s	**26.0s**	**41.4s**	**212.1s**	–
auto (16 – 1)	15.2s	36.8s	47.1s	85.9s	204.0s
speedup (16 – 2)	0.81× (8)	2.07× (8)	4.49× (8)	>1.70× (-)	–
speedup (16 – 1)	0.80× (8)	1.46× (8)	2.57× (1)	2.25× (1)	>1.76× (-)

(d) Webspam

Table 5.28: *Time to converge (continued) to duality-gap 1e-5 – 1e-9 for each dataset and speedup of auto scale-in compared to the best static node count larger than the minimum (1 or 2). The best static node count is given in parentheses. The fastest run for 1 or more nodes is underlined and **bold** for 2 or more nodes. "–" indicates that the respective duality-gap was not reached within the test time limit.*

5.9.2 Chunk shuffle policy

The chunk shuffle policy is based the same insight as the auto scale-in policy, namely that, at some point in time, all local correlations have been identified and convergence slows down. However, in contrast to the auto scale-in policy, the number of nodes is not reduced. Instead, a randomly selected subset of data chunks (the size of which corresponds to the fraction f of a node's number of local data chunks) is periodically shuffled across workers, such that each worker is able to see a different combination of chunks in each epoch. This has a similar effect as scale-in, as local solvers can identify additional correlations locally, but does not reduce the number of compute resources. A similar approach was described by Ioannou et al. [105] to optimize local, multi-threaded training. A description of all parameters is in Table 5.29.

Parameter	Description
f	Fraction of data chunks (range: 0.0 – 1.0) to be transferred between epochs. $f = 1.0$ results in a full shuffle.

Table 5.29: *Overview of parameters of the shuffle policy. Concrete values are given in the corresponding evaluation section.*

While this policy does not reduce the number of allocated resources, it can accelerate training even further than the auto scale-in policy, given a sufficiently fast network, as the level of parallel execution does not decrease. Moreover, it can also be used in cases where fewer workers do not have sufficient memory capacity to hold all data chunks.

5.9.2.1 Chunk shuffling in Chicle

The shuffle policy implements the method described above. The source code of this policy is shown in Listing 5.7. The `shuffle` function is called upon the reception of an *epoch finished* event.

```
map<Worker*, list<Chunk*>> mapping; // lists of chunks for each worker

void shuffleChunks() {
  // Number of shuffling rounds depends on the f configuration parameter.
  // See Table 5.29 for details.
  int rounds = max(1, numChunks / workers.size * f);

  for (int round = 0; round < rounds; round++) {
    // randomly shuffle workers order and chunk list of each worker
    shuffle(workers);

    // split workers list in two parts across which chunks will be swapped.
    vector<Worker*> part1; // workers '0' to '(workers.size + 1) / 2'
    vector<Worker*> part2; // workers '(workers.size + 1) / 2 + 1' to 'workers.size'

    int idx1 = 0; int idx2 = 0;
    while (idx1 < part1.size || idx2 < part2.size) {
      // Select two workers, one from each part
      Worker* w0 = part1[idx1];
      Worker* w1 = part2[idx2];

      // pick a chunk from each worker - due to the initial shuffling, chunks are picked
      // at random.
      Chunk* c0 = mapping[w0].front();
      mapping[w0].pop_front();
      Chunk* c1 = mapping[w1].front();
      mapping[w1].pop_front();

      // swap chunks.  This is an asynchronous, non-blocking call.
      moveChunk(c0, w0, w1);
      moveChunk(c1, w1, w0);

      // if part1 and part2 have different sizes, the last worker of the smaller part has
      // two chunks exchanged.
      if (idx1 < part1.size)
        idx1++;
      if (idx2 < part2.size)
        idx2++;
    }
  }

  // wait for all chunk data movements to complete.
}
```

Listing 5.7: *Simplified C++ code of the shuffle policy's algorithm.*

5.9.2.2 Evaluation

This experiment shows the benefit of recombining data chunks by shuffling them across all tasks in-between epochs. All experiments use the test setup and datasets described in Section A.3.3. The chunk shuffle policy (§5.9.2), which is configured to randomly swap $f = 5\%$, 10% and 15% of chunks of each task with that of another, randomly selected task.

	1e-05	**1e-06**	**1e-07**	**1e-08**	**1e-09**
baseline	4.4s	35.4s	152.5s	277.3s	–
shuffle (5%)	1.9s	4.9s	8.3s	12.3s	19.8s
shuffle (10%)	1.3s	2.8s	4.5s	6.8s	12.4s
shuffle (15%)	1.0s	2.1s	3.3s	5.1s	10.8s
speedup (5%)	2.36×	7.27×	18.43×	22.56×	>18.21×
speedup (10%)	3.38×	12.60×	33.66×	40.82×	>29.03×
speedup (15%)	4.25×	16.95×	46.83×	53.95×	>33.41×

(a) Higgs

	1e-05	**1e-06**	**1e-07**	**1e-08**	**1e-09**
baseline	5.4s	9.1s	13.2s	107.5s	330.9s
shuffle (5%)	6.0s	7.7s	9.8s	29.1s	63.4s
shuffle (10%)	6.2s	8.2s	10.2s	20.4s	38.5s
shuffle (15%)	6.6s	8.2s	10.1s	16.2s	29.3s
speedup (5%)	0.91×	1.18×	1.34×	3.69×	5.22×
speedup (10%)	0.87×	1.11×	1.29×	5.27×	8.59×
speedup (15%)	0.82×	1.11×	1.30×	6.64×	11.31×

(b) Criteo

Table 5.30: *Time to converge to duality-gap 1e-5 – 1e-9 for each dataset and speedup of shuffling compared to the baseline. "–" indicates that the respective duality-gap was not reached within the test time limit.*

Figures 5.27 and 5.28 show the convergence per epoch and over time of this experiment and Table 5.30 summarizes the results. Results indicate that chunk shuffling is beneficial and reduces the training time significantly, depending on the target duality-gap. Even when shuffling only 5% of all data chunks per epoch, a speedup of up to 30.4× can be achieved. With 15%, a speedup of up to 55.7× is possible. Only in a single case (Criteo, 1e-5) is shuffling not beneficial. Table 5.31 lists the data volumes that are

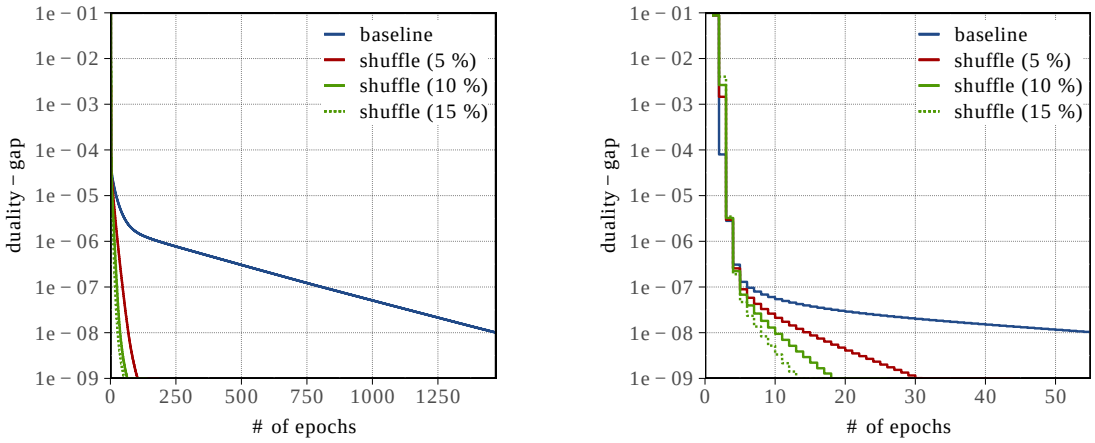
	1e-05	**1e-06**	**1e-07**	**1e-08**	**1e-09**
baseline	97.8s	–	–	–	–
shuffle (5%)	34.5s	63.2s	107.6s	203.1s	–
shuffle (10%)	27.3s	46.4s	85.7s	176.3s	–
shuffle (15%)	26.2s	45.4s	84.2s	175.3s	359.0s
speedup (5%)	2.83×	>5.70×	>3.35×	>1.77×	–
speedup (10%)	3.58×	>7.76×	>4.20×	>2.04×	–
speedup (15%)	3.73×	>7.94×	>4.27×	>2.05×	>1.00×

(c) KDDA

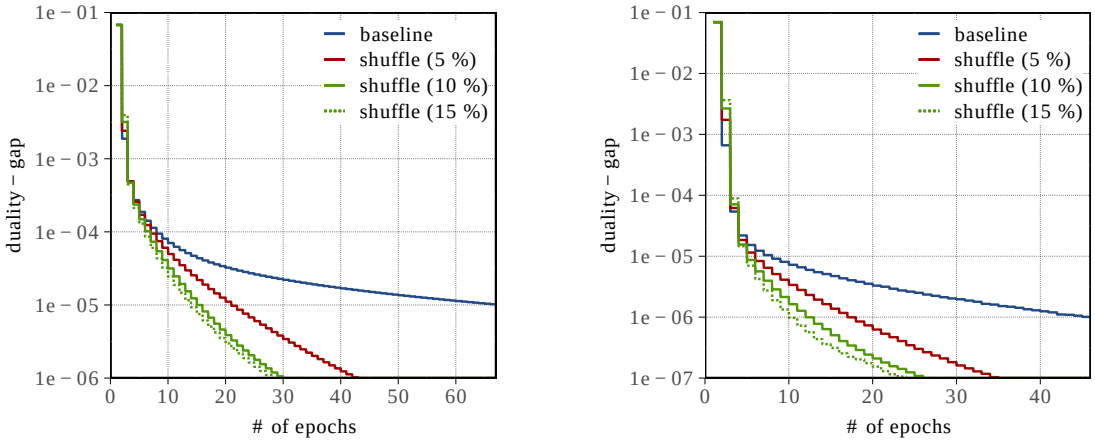
	1e-05	**1e-06**	**1e-07**	**1e-08**	**1e-09**
baseline	18.7s	86.4s	–	–	–
shuffle (5%)	15.3s	36.7s	68.2s	138.8s	–
shuffle (10%)	13.8s	27.7s	53.5s	123.8s	–
shuffle (15%)	13.7s	23.0s	49.7s	120.9s	–
speedup (5%)	1.22×	2.36×	>5.28×	>2.59×	–
speedup (10%)	1.36×	3.11×	>6.73×	>2.91×	–
speedup (15%)	1.36×	3.75×	>7.25×	>2.98×	–

(d) Webspam

Table 5.30: *Time to converge (continued) to duality-gap 1e-5 – 1e-9 for each dataset and speedup of shuffling compared to the baseline. "–" indicates that the respective duality-gap was not reached within the test time limit.*

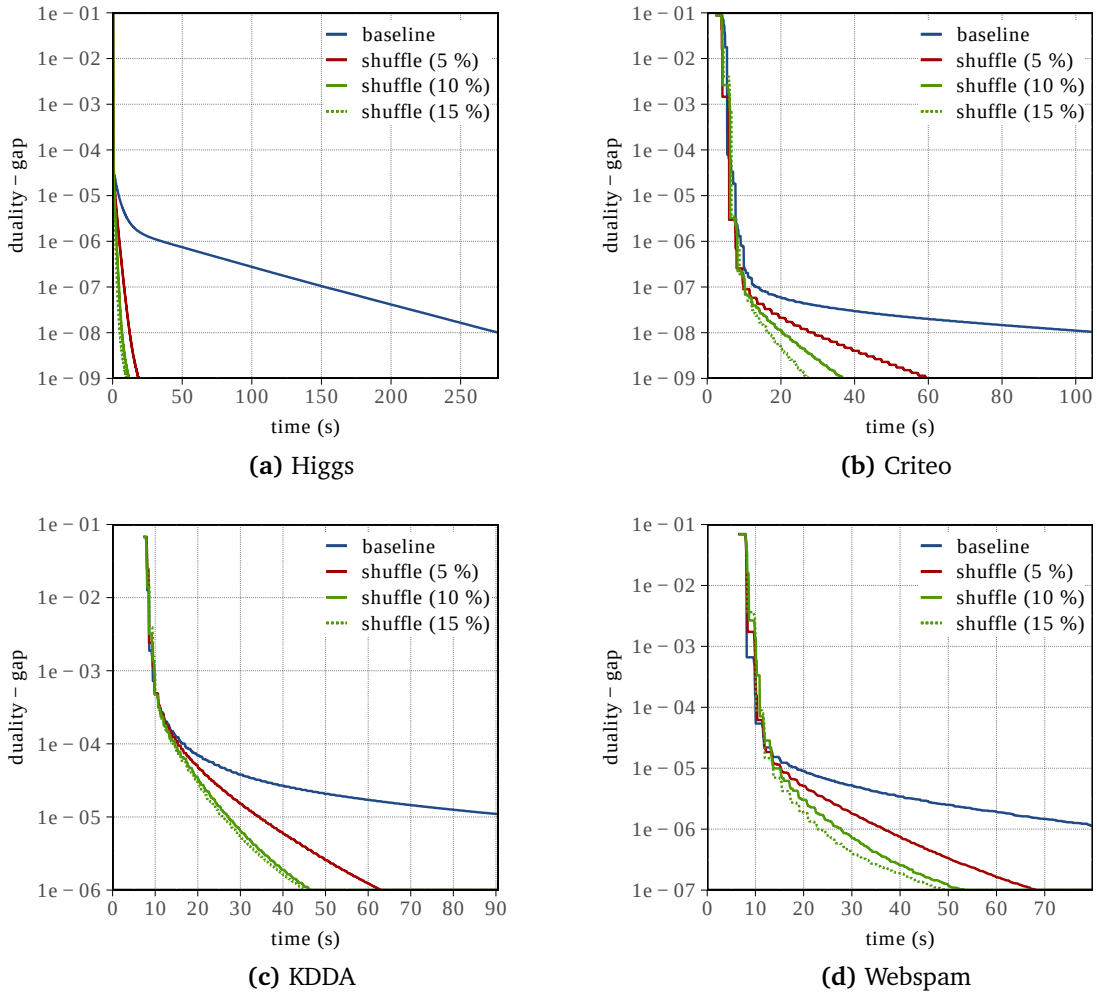


(a) Higgs **(b)** Criteo



(c) KDDA **(d)** Webspam

Figure 5.27: *Convergence of the duality-gap (lower is better) over epochs using the chunk shuffle policy with 5%, 10% and 15% of chunks shuffled during each epoch and the baseline, without shuffling, as reference.*



(a) Higgs **(b)** Criteo

(c) KDDA **(d)** Webspam

Figure 5.28: *Convergence of the duality-gap (lower is better) over time using the chunk shuffle policy with 5%, 10% and 15% of chunks shuffled during each epoch and the baseline, without shuffling, as reference.*

transferred during each epoch. These volumes do not need to be transferred from a single node but are evenly spread across all 16 nodes. Nevertheless, data transfer volumes pose the biggest issue with this policy. While the auto scale-in policy also transfers data chunks, it does so only during scale-in events, which are rare, compared to the number of epochs.

As the plots in Figure 5.28 show, shuffling is only beneficial beyond the knee. A knee detection has not been implemented for this policy yet, but could be adapted from the auto scale-in policy.

While global training sample randomization is common in many (rigid) frameworks, e.g. in PyTorch, this is typically done by loading the entire dataset on all nodes and not by directly exchanging loaded training data as Chicle does. This has two disadvantages:

- Training data loading can be time-consuming. For instance, it takes ≈11 minutes to load the entire 25GiB Criteo dataset by Chicle from the widely used SVM data format [108]. If the training data file is stored on a shared network file system, it also needs to be transferred to each node, potentially at the same time. Chicle, instead, loads each training sample only once, reducing data loading overheads, and further keeps only a single copy of each training sample in memory across all tasks to reduce memory consumption. However, Chicle's approach limits the granularity with which training samples can be shuffled to sets stored in the same data chunk. Loading all training samples on all nodes allows to shuffle on a per sample basis.

- Per-sample state, as it exists for CoCoA, needs to be transferred if the training sample is processed by a different task. In uni-tasks, the data chunk can store this state and automatcially transfers it along with the training sample, whereas if training data is loaded from a shared file system, this state needs to be transferred separately.

Overheads. Continuous data shuffling incurs data transfer overheads. Table 5.31 lists the amount of data that needs to be transferred during each epoch. The time given in Table 5.30 includes the data transfer time on the 56Gbps Infiniband network of the test cluster. In contrast to the auto scale-in policy, where data transfer overheads only incur twice, data is transferred during each epoch.

Datasets	5%	10%	15%
Higgs	15.1MiB	30.2MiB	45.3MiB
Criteo	91.8MiB	183.6MiB	275.5MiB
KDDA	15.8MiB	31.6MiB	47.4MiB
Webspam	62.9MiB	125.8MiB	188.7MiB

Table 5.31: *Per-epoch and node data transfer volumes during this experiment. Numbers are accumulated for send and receive.*

5.10 Related Work

System	year	straggler mitigation	h/w-hetero.-aware	elastic
Cipar et al. [31]	2013	+	-	-
Ho et al. [34]	2013	+	-	-
Li et al. [47]	2014	+	-	+
MXNet [53]	2015	+	-	-
TensorFlow [51, 62]	2015	+	-	-
FlexRR [68]	2016	+	-	-
Proteus [82]	2017	+	-	+
PyTorch [85]	2017	+	-	-
Snap ML [103]	2018	-	-	-
Litz [114]	2018	+	-	+
Optimus [112]	2018	+	-	+
Chicle [106]	2018	+	+	+

Table 5.32: *Summary of distributed ML frameworks w.r.t. their ability to address scheduling challenges.*

There is a large body of research that deals with efficient distributed ML training. Table 5.32 compares various works w.r.t. which scheduling challenges they address. In the following, the related work for each scheduling challenge is discussed in more detail.

Hardware-heterogeneity-aware load balancing Hardware-heterogeneity-aware load balancing is not explicitly addressed in related work on ML *training*. Heterogeneity in

related work often refers to the ability to use multiple types of hardware for computation, such as GPUs, TPUs and other accelerators or to scale from mobile devices to large compute clusters [53, 51, 62]. The ability to balance load across hardware resources of different speeds, that are used at the same time, is not addressed in related work.

General-purpose distributed application frameworks, such as Spark, are able to execute distributed ML training applications and balance load on heterogeneous clusters. However, they often suffer from lower performance compared to special-purpose ML frameworks [77, 62, 103], as they are optimized for deterministic, synchronous execution where tasks cannot explicitly hold state. For these reasons, they fail to exploit special properties of distributed ML algorithms, such as the ability to tolerate bounded errors during model updates [114]. A comparison of CoCoA on Spark and on Chicle has shown that Spark is 13.9× slower than Chicle, in a baseline scenario.

Furthermore, as performance differences between nodes are long-lasting or static, methods, such as SSP are not suitable to address this issue, as they can only deal with sporadic, short-term performance fluctuations.

Hardware-heterogeneity-aware load balancing is a primary concern for Chicle and addressed by the rebalance policy (Section 5.5.1).

Elasticity While being a desirable property in modern compute clouds and shared cluster environments, elasticity has gotten little attention by related work. Notable exceptions are Litz [114], which enables elastic distributed ML training on shared clusters using a micro-task approach and Proteus [82], which enables distributed ML training on transient, revocable resources by making workers stateless. Stateless workers are advantageous from a scheduler point of view, as no state needs to be transferred when adding or removing resources, but it also incurs overheads. For instance, in CoCoA, there is a per-sample state on each task which can amount to ≈368MiB (Criteo).

Optimus [112] is a cluster resource manager for elastic, distributed ML training based on MXNet [53]. The problem Optimus is addressing is orthogonal to the problems addressed with uni-tasks and Chicle. Instead of improving the convergence rate and efficiency of an individual training application, Optimus attempts to minimize average job completion time across all ML training applications sharing a cluster at the same time, by assigning more or fewer resources to applications. Furthermore, its MXNet based implementation requires to checkpoint application state to a shared file system (HDFS) and to restart an application to adjust the resources it is using. Chicle, on the other hand, can adjust the number of workers (tasks) without the need to checkpoint and restart, which is beneficial when large amounts of training data have to be loaded or nodes are added or removed frequently.

While related work extensively studies the impact of the batch size (for mSGD) on the number of epochs needed to converge [124], it has not been done in the context of elastic scheduling. Peng et al. [112] report that the batch size for mSGD remains constant during elastic scaling in their approach. The advantage is that convergence per epoch re-

mains independent of the elastic execution. However, it also either limits scalability (if it is chosen too small) and resource utilization efficiency (if only few samples are processed on each worker/task per iteration) or impairs convergence speed per epoch (if it is chosen too large). Chicle, dynamically adjusts the batch size (mSGD/lSGD) and the number of partitions (for CoCoA) which allows to set both values to maximize convergence per epoch with any number of nodes. A drawback of Chicle's approach is, however, that the number of nodes using during training can influence convergence.

Straggler mitigation Efficient distributed and scalable ML training requires effective straggler mitigation and is therefore a primary concern for many works. The general approach chosen by most works is relaxed consistency [23, 19, 25, 34, 43, 47, 82, 114, 104] where workers can run partially asynchronous and therefore don't have to wait for a straggler. SSP [31] is one of the most widely used approaches. A drawback of SSP is a potentially negative impact on per-epoch convergence rate due to additionally introduced bounded errors. FlexRR [68] improves upon SSP by temporarily reassigning work from slower workers to faster workers for the remainder of an iteration. This, however, comes at the cost of extra memory to hold training data assigned to other workers and effectively restricts the order in which local solvers can process examples to a non-random sequence. TensorFlow [62] additionally supports synchronous execution with backup workers to mitigate stragglers, similar to backup workers in MapReduce [6], at the cost of consuming additional resources. A similar approach is proposed by Karakus et al. [122].

Compared to the straggler mitigation techniques in related work, Chicle's task preemption policy does not require additional resources. Furthermore, the evaluation has not shown a systematic impairment of the per-epoch convergence rate but on the contrary, a small reduction in the number of epochs needed to converge (Section 5.7.2.2).

Other Many distributed ML training frameworks do not address any of the above mentioned issues, such as the original CoCoA implementation [46] and Snap ML [103]. However, these typically focus on other aspects of the training process, such as optimized solvers.

5.11 Conclusion and future work

This chapter presented uni-tasks, a novel task execution model for elastic, hardware-heterogeneity-aware, straggler-mitigating distributed ML training and Chicle, a prototypical implementation thereof. Two state-of-the-art distributed ML training algorithms, CoCoA and lSGD, have been implemented and evaluated w.r.t. their performance in elastic, heterogeneous and straggler afflicted scenarios. The evaluation showed that uni-tasks, in comparison to micro-tasks, reduces the total amount of work (number of epochs) necessary for elastic and load balanced training, and therefore utilizes available resources

more efficiently. A comparison with two rigid state-of-the-art distributed ML training frameworks furthermore showed that Chicle provides competitive performance. While the average maximal test accuracy of lSGD on uni-tasks exceeded the highest of what can be achieved on micro-tasks in many cases, the evaluation also showed that the maximal test accuracy for lSGD can be impaired by uni-tasks. An investigation into the impact of uni-tasks and the maximal test accuracy is left for future work.

5.11.1 Applicability and limitations

The uni-tasks concept is expected to be applicable beyond ML training algorithms. All algorithms that exhibit the properties listed in Section 5.2.2 can generally be implemented on top of uni-tasks. If not all properties are fulfilled, uni-tasks may still have limited applicability to iterative algorithms. The following list gives an overview over these limitations.

- In order to benefit from uni-tasks's elasticity, the algorithm needs to allow work to split up into independent sub-problems of the same or arbitrary size. Due to Chicle's synchronous implementation, sub-problems are ideally large.
- In order to benefit from uni-tasks's load balancing property, algorithms need to allow work to be split up into independent sub-problems of arbitrary size.

  Moreover, the *size* of the sub-problem p needs to correlate to the time it takes to solve a sub-problem, i.e., $\text{time}(p) = \text{size}(p) \times x + y$ with x being a factor determining the time needed to process individual work items and y a constant time overhead. The closer y is to 0, the more effective load can be balanced with uni-tasks. For instance, with mSGD, at least for the tested datasets and test cluster, y is a large constant, making $\text{time}(p)$ virtually independent of $\text{size}(p)$, for the tested sub-problem sizes. This makes uni-tasks's load balancing method, by adjusting the size of p, less effective.

  This, however, is not a real disadvantage of uni-tasks over micro-tasks, as here the cost for y has to be paid once per task, of which there need to be many in order to allow load balancing, whereas in uni-tasks, there is only a single one.
- In order to benefit from Chicle's straggler mitigation method, sub-problems need to be preemptable without impairing eventual correctness of the final result. Ideally, sub-problems are preemptable at arbitrary points in time. Similar to before, in contrast to lSGD, the current mSGD implementation does not fulfill this requirement.

If some or all of the necessary properties are fulfilled, uni-tasks can be used. Furthermore, it is most beneficial for algorithms that benefit from being executed using a small number of tasks with access to a large fraction of data, such as ML training. For other problems, uni-tasks may still provide benefits, as systems based thereon, such as Chicle,

may reduce system-related overheads, due to a smaller number of tasks compared to micro-tasks-based systems.

5.11.2 Future work

Chicle serves as proof-of-concept for uni-tasks and several features that can be beneficial in real-world deployments have not been implemented yet. The following (non-exhaustive) list contains possible future work items:

- Integration and evaluation of GPUs in Chicle is one of the most important future work items, which, due to lack of GPU resources, could not be done as part of this work. While GPUs may shift bottlenecks, they are not fundamentally different from CPUs from a scheduler's perspective and should therefore work well with uni-tasks and Chicle. Moreover, the reduced number of tasks of uni-tasks compared to micro-tasks may prove even more beneficial here, as fewer GPU kernels need to be started and fewer results collected.
- Investigation and implementation into advanced optimization techniques, such as automatic learning rate, momentum and batch size adjustment for mSGD and lSGD to improve convergence speed and maximal test accuracy.
- Scaling the batch size (mSGD/lSGD) and the number of partitions (CoCoA) with the number of nodes during elastic execution influences the convergence behavior of the algorithms. While the evaluation (Section 5.6.2) has shown that convergence rate per epoch as well as (in many cases) maximal test accuracy is not impaired, this behavior may not always be acceptable. An investigation into the benefits of limiting the range in which the batch size and the number of partitions can change as well as the maximal beneficial scale-out level remains for future work.
- Furthermore, while Chicle's synchronous approach works well for the evaluated datasets on the test cluster, its efficiency degrades when tasks become very short, as runtime jitter and other system overheads can become relatively large. A future investigation into the applicability of asynchronous approaches, such as SSP, may therefore be worthwhile.
- Fault tolerance and recovery in case of node failures was not addressed as part of this thesis but can be relevant if training takes hours or days.
- Implementation and evaluation of additional distributed ML algorithms, such as multinomial logistic regression (MLR) and latent dirichlet allocation (LDA), as well as an investigation into the applicability of uni-tasks to non-ML algorithms.

Overall, uni-tasks represents a novel, effective and efficient approach to elastic scaling, load balancing and straggler mitigation for distributed ML training applications. Chicle

is one of the first distributed, elastic ML frameworks and, to the best of my knowledge, the only one that also supports load balancing on heterogeneous clusters, which further increases resource utilization efficiency without compromising application performance.

6. Conclusion

With the advent of cloud computing, the compute infrastructure became shared, virtualized, and heterogenous. As a result, the performance of cloud-provided compute services is less predictable. New cluster scheduling techniques apply elasticity and heterogeneity awareness to mitigate this problem, while at the same time aiming at improving cloud resource utilization.

Modern distributed application frameworks, such as Apache Spark, rely on micro-tasks as a scheduling technique to address heterogeneity, and to enable elastic execution in shared environments. Since executing each micro-task as a separate process would incur prohibitive overheads, long running task executor processes are used to partially amortize startup costs across many tasks. While breaking up work into many micro-tasks enables *natural* load balancing on heterogeneous clusters, each additional task also incurs some additional overhead which reduces overall resource utilization efficiency.

Unfortunately, current schedulers do not consider an important property of these executors: Over their lifetime, executors accumulate state. Just-In-Time (JIT) compilers translate interpreted code into native code to accelerate execution of subsequent tasks and data is cached such that future accesses can be served quicker. Ignoring this property leads to sub-optimal scheduling decisions.

In Chapter 3, methods to improve scheduling of individual applications on heterogeneous clusters were examined and the impact of various factors on the task runtime analyzed. During this analysis, the importance of the executor state on task runtime was recognized. A new scheduling technique, stage packing, that exploits this state to reduce task and application runtime, was devised. Stage packing was implemented in HCL-SP, a state-, directed acyclic graph (DAG)-, task- and hardware-heterogeneity-aware scheduler, and integrated into Spark. Using machine learning (ML) techniques to predict task runtimes as well as simple rules to select nodes classes on heterogeneous clusters were integrated. The evaluation has shown that application runtime on heterogeneous clusters can be reduced by $\approx$1.4$\times$ while resource utilization was tripled.

However, executors were still idle for two thirds of the time. To address this issue, executors needed to be shared efficiently across applications. This was addressed in Chapter 4, where executor sharing overheads were analyzed and quantified. Based on the findings, a new resource manager and elastic application scheduler, Mira, has been developed and integrated into Spark. Mira treats executors as resource which can be shared across applications in milliseconds instead of seconds. As executors are not shut

down anymore, their accumulated state remains intact and task execution is accelerated. The evaluation has shown that Mira almost doubles the speed of applications in a shared environment and increase resource utilization to 96.50%. This demonstrates the importance of efficient resource sharing and the ability of application frameworks to scale elastically in order to do so.

For one important class of applications, however, elastic execution remains a challenge. In recent years, ML technologies, that powers today's pervasive digital assistants, recommendation systems, social networks and more have emerged. Distributed iterative-convergent ML training algorithms allow to analyse vast amounts of data that these technologies rely on. Algorithms such as Mini-batch SGD (mSGD) for the training of neural networks (NNs), however, have peculiarities that make the micro-task approach less efficient: Splitting up the training process into arbitrarily small tasks impairs their efficiency on an algorithmic level and can increase the total amount of work that needs to be performed to achieve the same result. In consequence, only few elastic and no hardware-heterogeneity-aware distributed ML training frameworks exist.

In Chapter 5, a new execution model for iterative-convergent distributed ML training algorithms, uni-tasks, and Chicle, an implementation thereof, are presented. Instead of scheduling tasks, uni-tasks schedules small chunks of training data that can be moved across and recombined on nodes. In consequence, the number of tasks equals the number of nodes, which allows the training algorithm to work at maximal efficiency with any number of nodes. Chicle uses task runtime prediction to balance load across nodes of heterogeneous clusters and redistributes data chunks to elastically scale in and out. Furthermore, Chicle implements a novel straggler mitigation technique based on task preemption. The evaluation of two state-of-the-art algorithms, Communication-efficient distributed dual Coordinate Ascent (CoCoA) and Local SGD (lSGD) has shown that with uni-tasks, the number of epochs needed to converge is reduced by up to one order of magnitude compared to micro-tasks. Chicle also performs competitively in non-elastic scenarios on homogeneous clusters, as a comparison with two state-of-the-art ML frameworks has shown. Chicle achieves an average speedup of 1.9× and 1.3× depending on the training algorithm.

6.1 Outlook and future work

The work presented here has touched many aspects of scheduling, resource management and execution for data-analytics as well as machine learning applications. While many issues have been addressed, new ones were identified during the course of this work. The importance of the scale-out level for distributed execution and the resource utilization efficiency thereof has been recognized. The resource scale-out factor mechanism in Mira addresses this aspect but a policy to determine the optimal value has yet to be devised.

The work on uni-tasks and Chicle has touched many aspects that could not all be addressed as part of this thesis. While the evaluated algorithms cover some of the most

important and often used algorithms, several other potential algorithms, such as Multinomial Logistic Regression (MLR), Latent Dirichlet Allocation (LDA) and others exist that may benefit from uni-tasks. Furthermore, additional algorithm-specific optimizations similar to the presented auto scale-in and continuous shuffling policies may exist for other algorithms. Beyond ML, uni-tasks and Chicle may have applications for instance with graph algorithms, such as page rank and others that can benefit from few large tasks instead of many small ones and fulfill the general assumptions made by uni-tasks (Section 5.3.1).

In the near and medium term, the integration of techniques developed during this work into commercial products is also planned.

A. Appendix

A.1 HCL

A.1.1 HCL-SP Representational State Transfer (REST) interface

HCL-SP implements a programming-language agnostic REST interface which is accessible via a built-in Hypertext Transfer Protocol (HTTP) server and accepts and returns data encoded in the text-based JavaScript Object Notation (JSON) data format. REST and JSON are already used within Spark, e.g., to communicate with the YARN resource manager (RM) and a new connector for HCL-SP has been implemented inside the `HclSchedulerClient`.

URL	Event types
`/apps/{app}`	*Application-events* for the application with the id `{app}`. Application ids are unique. Table A.2 lists all events of this type.
`/apps/{app}/jobs/{job}`	*Job events* for the job with the id `{job}` of application `{app}`. Job ids are unique within each application. Table A.3 lists all events of this type.
`/apps/{app}/stages/{stage}`	*Stage events* for the stage with the id `{stage}` of application `{app}`. Stage ids are unique within each application. Table A.4 lists all events of this type.
`/apps/{app}/stages/{stage}/tasks/{task}`	*Task events* for the task with the id `{task}` of stage `{stage}` of application `{app}`. Task ids are unique within their stage. Table A.5 lists all events of this type.

Table A.1: *HCL-SP's REST interface hierarchy.*

All events include an `event` field that specifies the event and a `timestamp` field that specifies the time the event was generated in microseconds. The meaning of the former is only unique in combination with the event type. In the following, all available events for each event type are listed:

Event	Description
`start`	**Application submitted:** A new application with the given id `{app}` is submitted. The `{app}` is a unique identifier of the application instance. This event contains the following additional fields: ■ `name`: Name of the application. This field is the same for all instances of an application.
`end`	**Application finished:** An application has finished. This event contains no additional fields.

Table A.2: *List of HCL-SP's application events.*

Event	Description
`submit`	**Submit job:** A new job is added to the application. This event contains the following additional fields: ■ `stages`: List of stages in this job and their dependencies. The stages list contains the following fields: ■ `id`: Unique stage id. ■ `parents`: List of parent stage ids. ■ `size`: Number of tasks in this stage. ■ `type`: Type of stage, can be `load` or `compute` to specify whether a task gets its input data from a file system or consumes intermediate results from a previous stage as input data. ■ `function`: String of function executed in this stage, e.g. `"persist at SQLTest.scala:118"`. ■ `key`: Stable ID to identify this stage across instances of this application. ■ `input`: List of key-value pairs specifying the location (node) and the amount of input data (if available), e.g., `{"node01" : "1024"}` to specify that 1024 bytes of input data is located on `node01`. This data is only available at this point for stages of type `load`.

Table A.3: *List of HCL-SP's job events.*

Event	Description
`ready`	**Stage ready:** A stage is ready to be executed. This event contains the following additional fields: ■ `input`: List of key-value pairs specifying the location (executor) and the amount of input data. For stages of type `compute`, input data location is available at this time. A stage ready event automatically marks all of its tasks as ready.

Event	Description
`finished`	**Stage finished:** A stage has finished execution, i.e., all tasks have been executed. This event contains the following fields: ■ `status`: Can be either `succeeded` or `failed` to indicate whether a stage was executed successfully or not.

Table A.4: *List of HCL-SP's stage events.*

Event	Description
`finished`	**Task finished:** A task has finished execution. This event contains the following additional fields: ■ `status`: Can be either `succeeded` or `failed` to indicate whether a stage was executed successfully or not. ■ `executor`: Id of executor this task was executed on. ■ `shuffleReadRemote`: Amount of shuffle (input) data read from a remote location. ■ `shuffleReadLocal`: Amount of shuffle (input) data read from the same node. ■ `bytesRead`: Total number of bytes read (including broadcast and file system data). ■ `bytesWritten`: Total number of bytes written. ■ `cpuTime`: Total time used to execute the task on the executor. After a task has finished execution, the client (long) polls on this connection until HCL assigns another task (or command) for the executor.

Table A.5: *List of HCL-SP's task events.*

A.1.2 Test setup

All experiments use the test setup, software and configuration described in the following.

A.1.2.1 Cluster

Table A.13 lists the test cluster configuration. As I did not have full control over the test cluster and had to share it with other users and projects, the cluster used a variety of Linux Distributions and versions. Furthermore, nodes were added and removed during the lifetime of the cluster, due to upgrades and node failures, hence hardware resources are heterogeneous.

Node class	Quantity	CPU	Memory	OS
1	5	Intel Xeon E5-2650v2, 2.60GHz	160GiB	RHEL 7.5
2	2	Intel Xeon E5-2630v3, 2.40GHz	160GiB	RHEL 7.5
3	4 (3)	Intel Xeon E5-2640v3, 2.60GHz	256GiB	Fedora 26
4	4	Intel Xeon E5-2640v3, 1.20GHz	256GiB	CentOS 7.5

Table A.6: *Test cluster hardware and OS versions.*

All nodes are connected by a 56Gbps Infiniband network using a single Mellanox SX6036 switch. IPoIB was used for inter-node communication.

One node of node class 3 was exclusively used as control node where HCL, HDFS namenode and YARN master were executed. No executors were executed on the control node, hence only three nodes of node class 3 run executors.

A.1.2.2 Software

Table A.7 summarizes the software versions used throughout the evaluation.

Software	Version
Apache YARN	2.7.3
Apache HDFS	2.8.2
Apache Spark	2.2.1
Oracle HotSpot JVM	1.8.0_144

Table A.7: *Software versions.*

A.1.2.3 Settings

HCL, YARN's resource manager as well as HDFS's nameserver are executed exclusively on the control node. The HDFS file system is backed by 16 GB ram-disks on each of the 15 compute nodes. HDFS's balancer was invoked to distribute data across all nodes evenly.

Each compute node runs at most eight single-user-task executors[1] with a maximal heap size of 8GiB, resulting in a maximum of 112 executors in the cluster.

Parameter	Value (default)
spark.shuffle.service.enabled	true (false)
spark.sql.shuffle.partitions	56 (200)
spark.default.parallelism	64 (# of CPUs)
spark.serializer	KryoSerializer (JavaSerializer)
spark.sql.autoBroadcastJoinThreshold	20971520 (10485760)
spark.executor.cores	1 (1 with YARN)

(a) Spark settings

Parameter	Value
XX:MaxHeapSize (driver)	12g
XX:MaxHeapSize (executor)	8g
Garbage collector	XX:+UseG1GC

(b) JVM settings

Table A.8: *Relevant settings used throughout this evaluation, unless noted otherwise. Default values are given in braces.*

Spark settings are the result of performance tuning for vanilla Spark. `spark.shuffle.service.enable=true` is not necessary for HCL-SP but has been chosen to reduce the difference to the Mira setup. For `spark.default.parallelism` and `spark.sql.shuffle.partitions`, a balance between application runtime and resource utilization has been struck. Details about their impact is shown in Section 2.2.3.2. JVM settings reflect requirements by the test applications as well as performance tuning (garbage collector).

A.1.3 Example cluster definition file

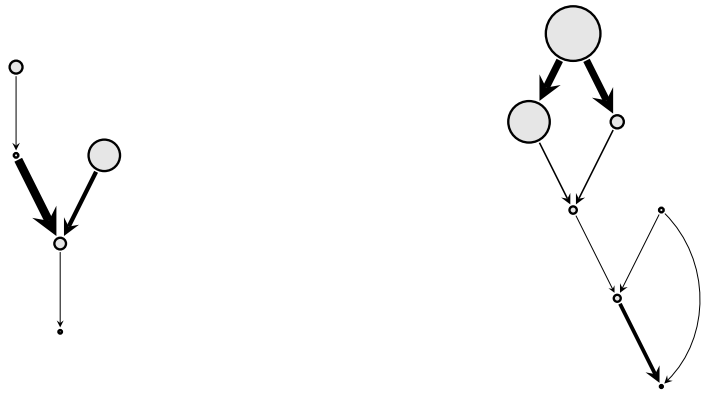
```
"type":"cluster","id":"example-cluster",
"graph": {
  "type":"tree",
  "vertices": [
    {
```

[1] Each executor typically spawns multiple background threads for system-related tasks, such as garbage collection and communication with the Spark driver

```
      "id":"node1.example.cluster.org",
      "nodeclass":"1",
      "architecture":"x86_64",
      "model":"Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz",
      "connectors":[{"id":"eth0","level":"1"}],
      "domains": [
        {
          "id":"0",
          "memory":"136564400128",
          "cores": [
            {"id":"0","pus": ["0","2","4","6","8","10","12","14","16","18",
                              "20","22","24","26","28","30"]},
            {"id":"1","pus": ["1","3","5","7","9","11","13","15","17","19",
                              "21","23","25","27","29","31"]}],
          "iodevs": [
            {"id":"45056","type":"storage","dev":"sda"},
            {"id":"16384","type":"gpu","dev":"card0","model":"Nvidia GTX 1080"},
            {"id":"24578","type":"net","dev":"eth0","ipv4":"10.0.0.1/24",
             "bw":"1000000000"}]
        }]
    },
    // more vertices
    ]
}
```

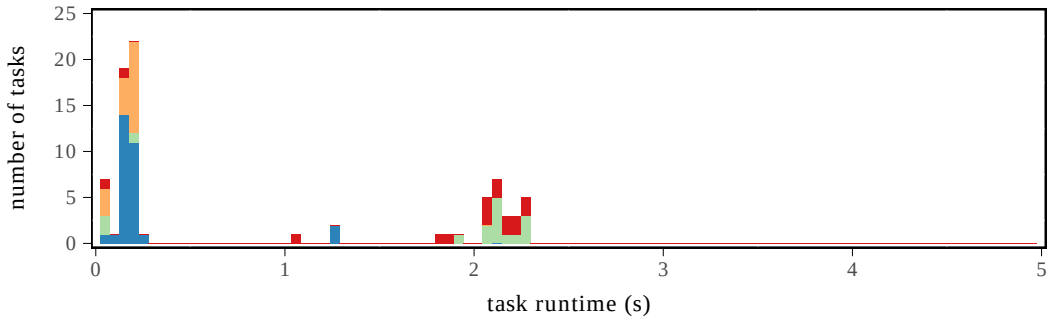
A.1.4 Test applications

Throughout the evaluation of HCL-SP and Mira, a Spark implementation [116] of the the TPC-DS benchmark suite is used, which "provides a representative evaluation of performance as a general purpose decision support system" [88]. The TPC-DS benchmark consists of ≈100 Structured Query Language (SQL) queries which vary in complexity, runtime, amount of processed data, DAG structure, depth and width. Figures A.1 and A.2 shows the DAG and the task runtime distribution for two queries across the entire applications as well as across individual stages as example of the variation within the set of TPC-DS queries.

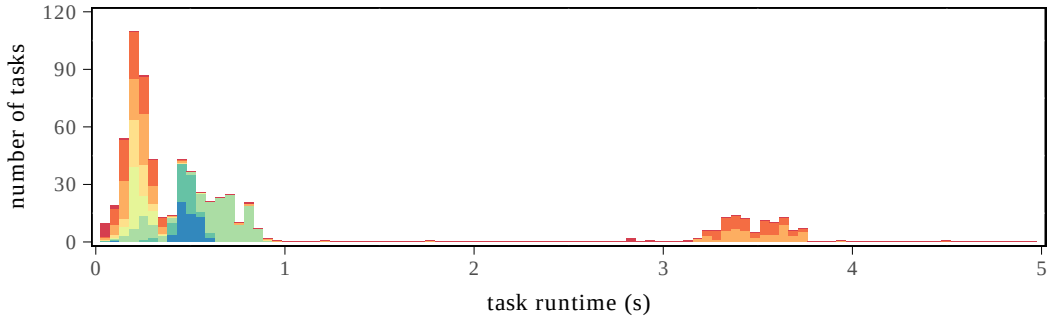


(a) TPC-DS query 41 **(b)** TPC-DS query 44

Figure A.1: *Example of the heterogeneity of application DAGs. Both figures show DAGs of TPC-DS queries. The size of nodes represents the total amount of work (accumulated task runtimes) performed in the corresponding stage. The widths of arrows corresponds to the amount of data transferred between two stages.*



(a) TPC-DS query 41



(b) TPC-DS query 44

Figure A.2: *Example of relative normalized task runtime distributions for two TPC-DS queries. The histograms show two things: The runtime distribution across all tasks and the runtime distribution within each stage. Stages are color-coded. The y-axis shows the normalized fraction of tasks on a linear scale.*

Spark's SQL engine compiles each SQL query into Scala code, which in turn is compiled into JVM byte code. As different code generated, compiled and executed for each query, each is be considered a different application.

Due to incomplete changes in Spark's lineage graph that were required to compute the stable stage ID, only 90 queries could be executed, as others relied on Resilient Distributed Dataset (RDD) types for which the computation of the stable ID was infeasible due to the complexity of the Spark code.

Input data. The input data has been generated using a Spark version of the TPC-DS data generator utility [129]. The scaling factor parameter has been set to 100, which corresponds to ≈100GiB of raw table data (≈40GiB in the Parquet file format). All data was stored in the Parquet data format [117] on a distributed HDFS file system.

A.1.5 Supplemental results

This section provides enlarged plots of Section 3.4.5.5.

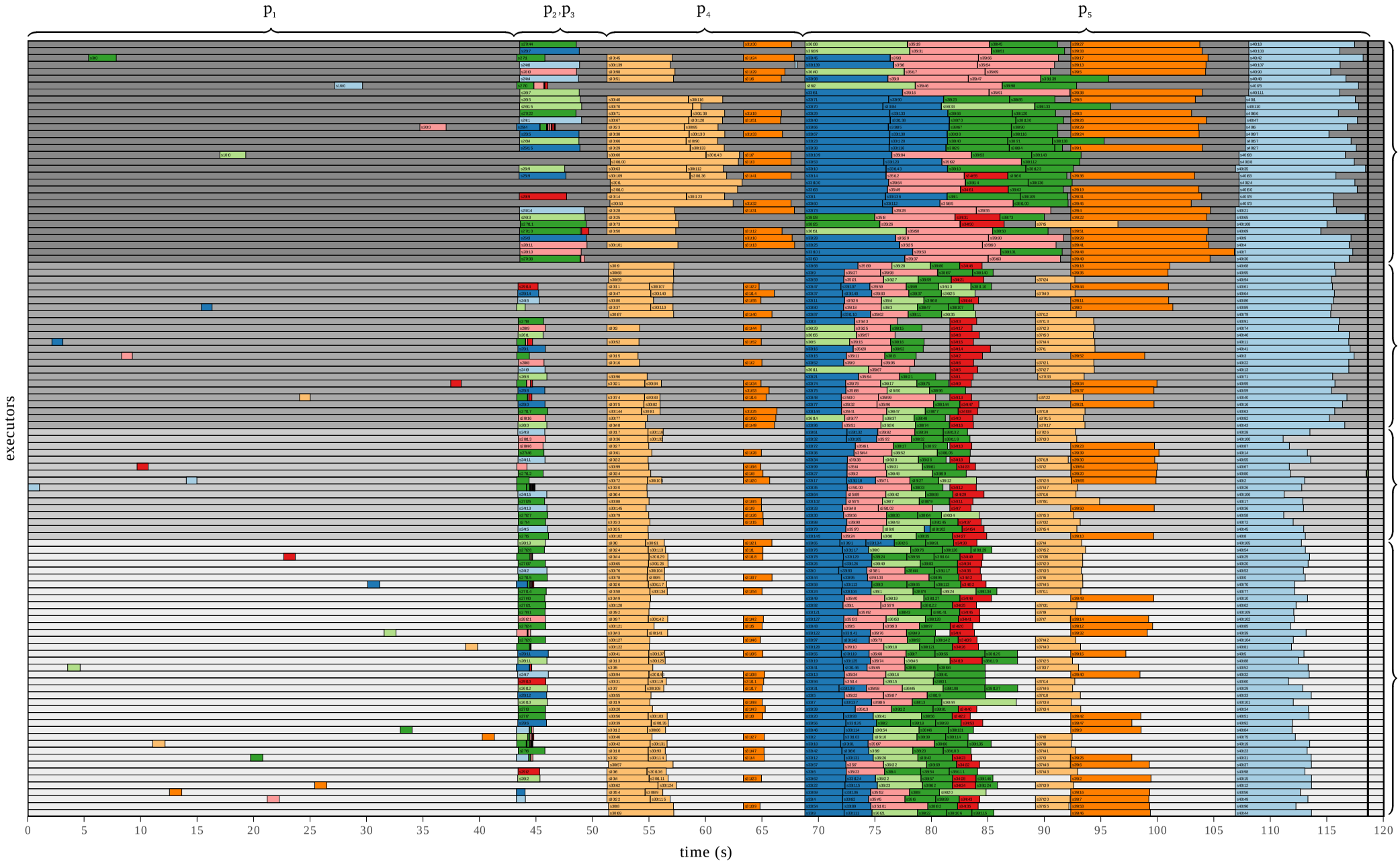


Figure A.3: *TPC-DS query 23a on vanilla Spark. Larger plot of Figure 3.16a.*

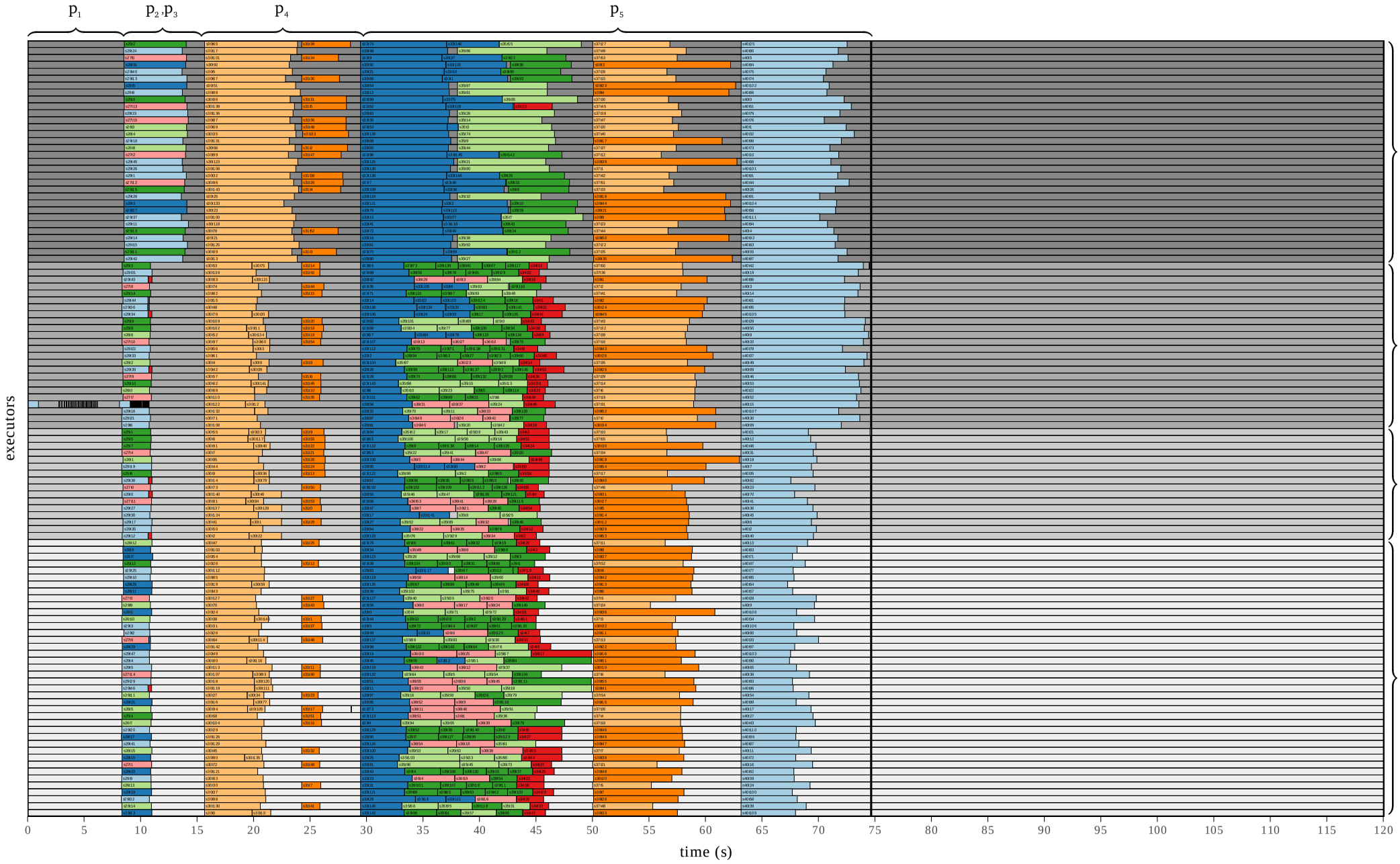


Figure A.4: *TPC-DS query 23a on Spark+HCL-SP. Larger plot of Figure 3.16b.*

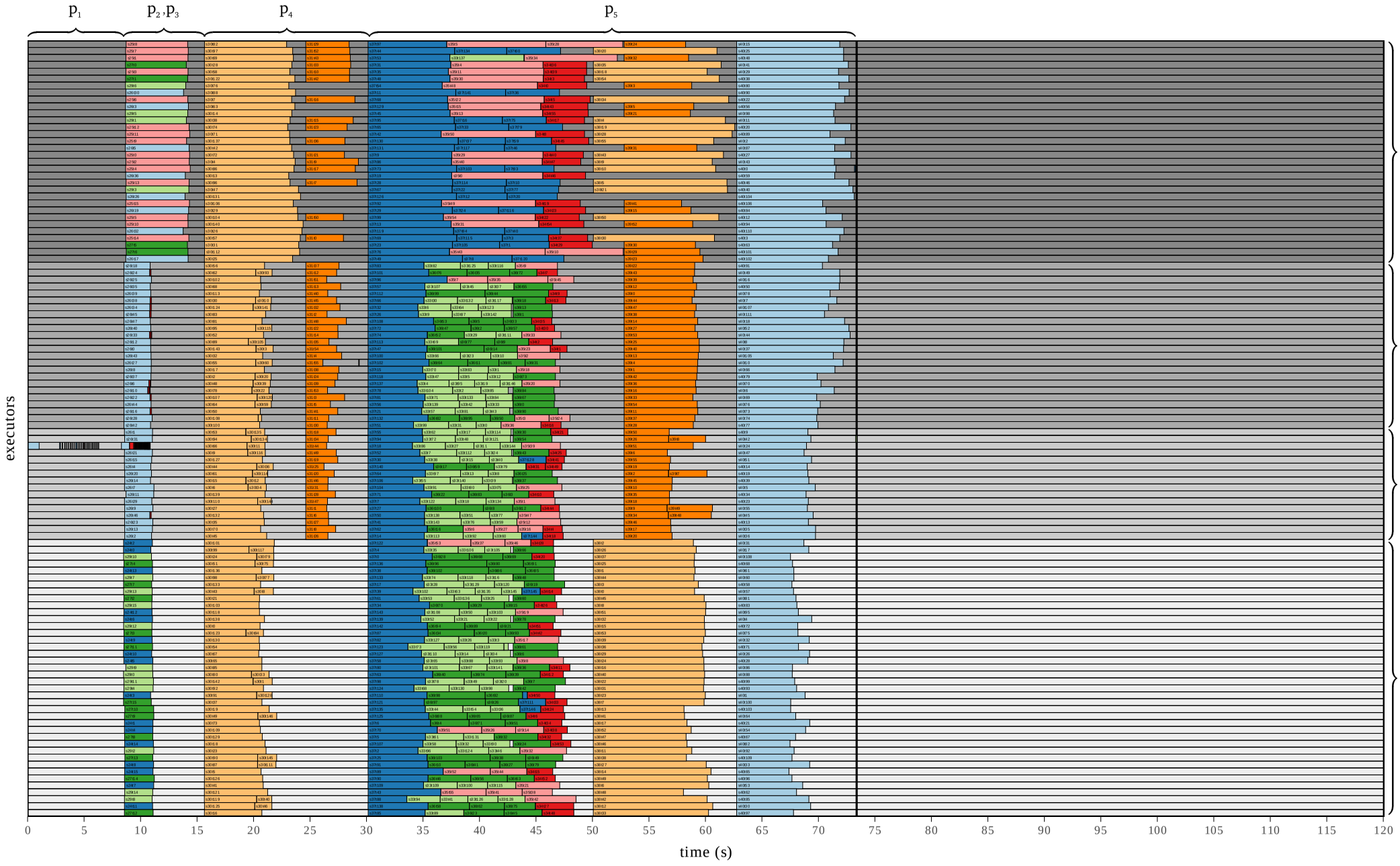


Figure A.5: *TPC-DS query 23a on Spark+HCL-SP (RP).*

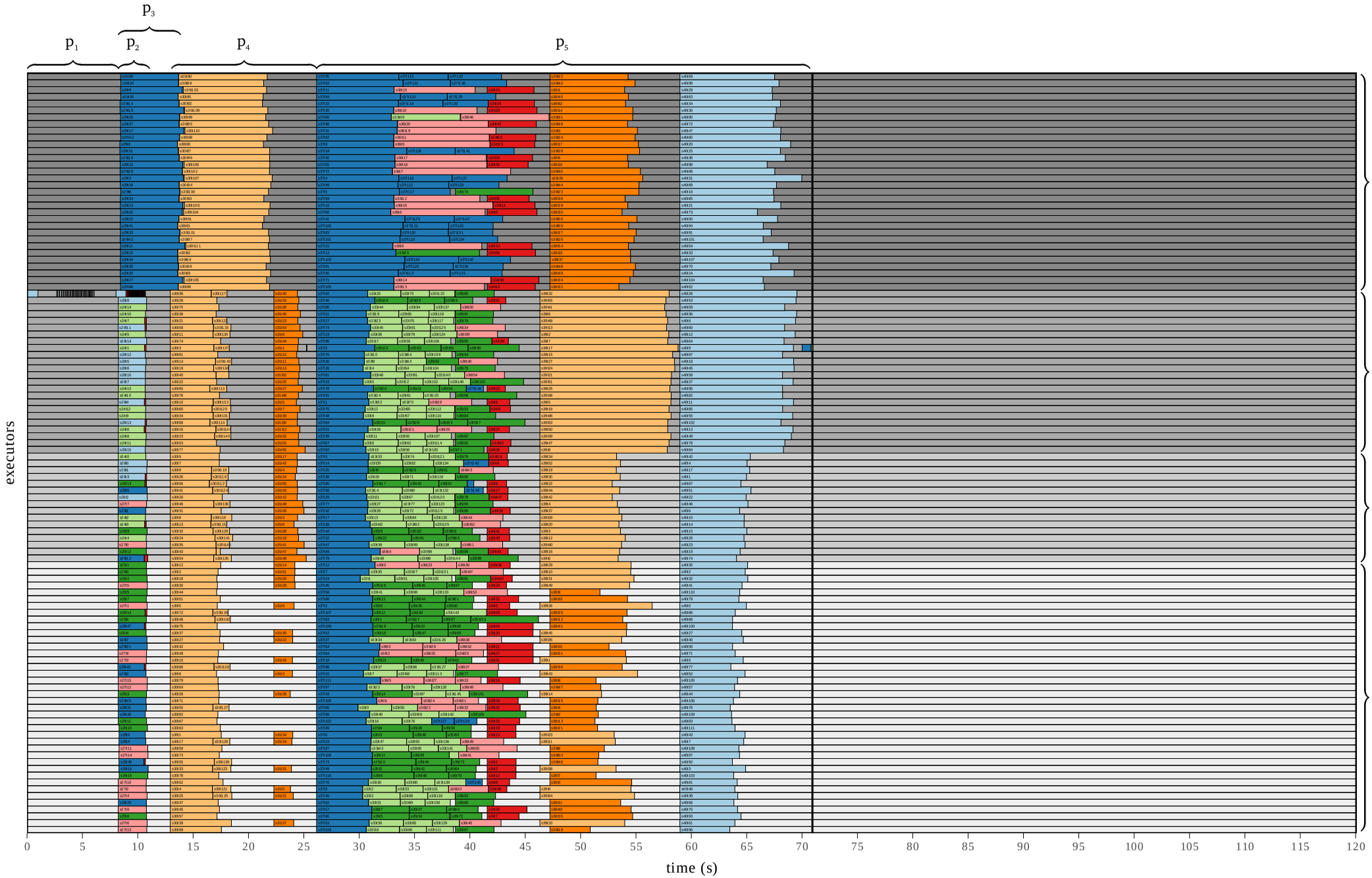


Figure A.6: *TPC-DS query 23a on Spark+HCL-SP (FP).*

A.2 Mira

A.2.1 Mira REST interface

Mira's REST interface is based on that of HCL-SP (Section A.1.1) and **extends** it as described in the following.

URL	Event types
`/apps/{app}/executors/{executor}`	*Assigned executor events* for the executor with the id `{executor}` that is currently assigned to application `{app}`. Executor ids are unique within an application. Executor instances that disconnect and reconnect to the same application are assigned a new (temporary) application-specific id. Table A.11 lists all events of this type.
`/executors/{executor}`	*Unassigned executor events* for the executor with the id `{executor}` that is currently unassigned. The executor id here is stable and identifies the executor instance. Table A.12 lists all events of this type.

Table A.9: *Mira's REST interface hierarchy as extension to the HCL-SP REST interface hierarchy (Table A.1).*

All events include an `event` field that specifies the event and a `timestamp` field that specifies the time the event was generated in microseconds. The meaning of the former is only unique in combination with the event type. In the following, all available events for each event type are listed:

Event	Description
`start`	**Application submitted:** A new application with the given id `{app}` is submitted. The `{app}` is a unique identifier of the application instance. This event contains the following additional fields: ■ `name`: Name of the application. This field is the same for all instances of an application. ■ `driver`: URL to the application driver scheduler interface, e.g., `spark://CoarseGrainedScheduler@10.0.0.25:40129`. The driver URL is passed to executors to connect to during driver assignment.

Table A.10: *List of Mira's application events as extension to the HCL-SP application events (Table A.2).*

Event	Description
`register`	**Register executor:** An executor is **re**-registering itself with an application after a previous assignment by the RM. This event contains the following additional fields: ■ `host`: Name of the host where this executor is running on.
`deregistered`	**Deregister executor confirmation:** An executor is confirming that it has removed itself from an application after a previous disconnect from driver command by the RM. This event contains no additional fields.

Table A.11: *List of Mira's assigned executor events.*

Event	Description
`register`	**Register executor:** An executor is registering itself with the RM and asks for a application driver (DRV) to connect to. This event contains the following additional fields: ■ `host`: Name of the host where this executor is running on.

Event	Description
`requestDriver`	**Request driver:** An unassigned executor is asking for a new driver assignment after having disconnected from its previously assigned driver. This event contains no additional fields.

Table A.12: *List of Mira's unassigned executor events.*

A.2.2 Test setup

A.2.2.1 Cluster

Node class	Quantity	CPU	Memory	OS
1	5	Intel Xeon E5-2650v2, 2.60GHz	160GiB	RHEL 7.5
2	2	Intel Xeon E5-2630v3, 2.40GHz	160GiB	RHEL 7.5
3	4 (3)	Intel Xeon E5-2640v3, 2.60GHz	256GiB	Fedora 26
4	4	Intel Xeon E5-2640v3, 2.60GHz	256GiB	CentOS 7.5

Table A.13: *Test cluster hardware and OS versions.*

All nodes are connected by a 56Gbps Infiniband network using a single Mellanox SX6036 switch. IPoIB was used for inter-node communication.

One node of node class 3 was exclusively used as control node where Mira, HDFS namenode and YARN master were executed. No executors were executed on the control node, hence only three nodes of node class 3 run executors.

A.2.2.2 Software

The following software versions (Table A.14) are used throughout the evaluation:

Software	Version
Apache YARN	2.7.3
Apache HDFS	2.8.2
Apache Spark	2.2.1
Oracle HotSpot JVM	1.8.0_144

Table A.14: *Software versions.*

In order to perform detailed performance analysis, Spark's DAG and task scheduler (e.g. for executor request, release and registration, task start and finish), as well as executor (e.g. startup and shutdown, task start and finish) were annotated with time-stamped log messages. These annotations are present in the vanilla, otherwise unmodified version of Spark, as well as in the Mira-integrated version, hence any potential performance implications thereof affects both versions equally.

A.2.2.3 Settings

Mira, YARN's resource manager as well as HDFS's name server are executed exclusively on the control node. The HDFS file system is backed by 16 GB ram-disks on each of the 15 compute nodes. HDFS's balancer was invoked to distribute data across all nodes evenly. Each compute node runs at most eight single-user-task executors[2] with a maximal heap size of 8GiB, resulting in a maximum of 112 executors in the cluster.

Parameter	**Value**
XX:MaxHeapSize (driver)	12g
XX:MaxHeapSize (executor)	8g
Garbage collector	XX:+UseG1GC

(a) JVM settings

[2]Each executor typically spawns multiple background threads for system-related tasks, such as garbage collection and communication with the Spark driver

Parameter	Value (default)
spark.dynamicAllocation.enabled	true (false)
spark.shuffle.service.enabled	true (false)
spark.dynamicAllocation.minExecutors	1 (1)
spark.dynamicAllocation.executorIdleTimeout	1s (60s)
spark.sql.shuffle.partitions	56 (200)
spark.default.parallelism	64 (# of CPUs)
spark.serializer	KryoSerializer (JavaSerializer)
spark.sql.autoBroadcastJoinThreshold	20971520 (10485760)
spark.executor.cores	1 (1 with YARN)

(b) Spark settings

Parameter	Value (default)
yarn.nodemanager.resource.cpu-vcores	8 (unlimited)
yarn.resourcemanager.scheduler.class	CapacityScheduler (FairScheduler)
yarn...capacity.preemption.total_preemption_per_round	1.0 (0.1)
yarn...capacity.preemption.max_wait_before_kill	1s (15s)
yarn...capacity.preemption.monitoring_interval	1s (3s)
yarn.scheduler.capacity.root.queues	background, foreground (n/a)
yarn.scheduler.capacity.root.background.capacity	50 (n/a)
yarn.scheduler.capacity.root.background.maximum-capacity	99 (n/a)
yarn.scheduler.capacity.root.background.user-limit-factor	2.0 (n/a)
yarn.scheduler.capacity.root.foreground.capacity	50 (n/a)
yarn.scheduler.capacity.root.foreground.maximum-capacity	100 (n/a)
yarn.scheduler.capacity.root.foreground.user-limit-factor	2.0 (n/a)

(c) YARN settings

Table A.15: *Relevant settings used throughout this evaluation, unless noted otherwise. Default values are given in braces.*

Settings for Mira are based on those for HCL (Section A.1.2.3) but have been extended due to enable elastic scaling on Spark and YARN. Maximum background capacity settings for YARN were necessary as foreground applications would not launch with a value of 100. Furthermore, the resource scale-out factor α for Mira has been set to 1.0 to resemble vanilla Spark's behavior.

A.2.3 Test applications

Mira was evaluated using the TPC-DS benchmark suite, which is described in Section A.1.4. Additionally, the following micro-benchmarks were used.

A.2.3.1 Application to demonstrate executor acquisition delay

```
def main(args: Array[String]): Unit = {
  val ss = SparkSession.builder.appName("TestApp").getOrCreate()
  val sc = ss.sparkContext

  acquireExecutors(sc, 110, 10000) // 1. Acquire 'par' executors
}

def acquireExecutors(sc: SparkContext, par: Int, duration: Int): Unit = {
  // Build a dummy list with as many entries as executors to acquire. Spark needs to think
  // it processes data, otherwise it won't acquire executors.  The values are irrelevent.
  val data = List.newBuilder[Int]
  for (i <- 0 until par) { data += i }

  // Initiate the parallel execution. For each entry in 'data', one task is spawned. For
  // each task, one executor is acquired.
  val result = sc.parallelize(data.result(), data.result().size).map(v => {
    val t0 = System.currentTimeMillis()
    Thread.sleep(duration) // Wait for 'duration' seconds. It needs to sleep longer than
                           // Spark needs to acquire all executors, otherwise two or more
                           // tasks may be scheduled back-to-back on the same executor.
    val t1  = System.currentTimeMillis()
    (t1 - t0) // necessary return value. The specific value is irrelevant.
  }).collect()
}
```

Listing A.1: *Scala code of application to demonstrate executor acquisition delay*

A.2.3.2 Application to measure executor acquisition delay

```
def main(args: Array[String]): Unit = {
  val ss = SparkSession.builder.appName("TestApp").getOrCreate()
  val sc = ss.sparkContext
  // Warm-up driver, to make sure all executor acquisition paths in Spark (and YARN, if
  // used) are initialized and the JIT-cache is warm. We don't want to measure driver
  // initialization overheads.
  for (i <- 1 until 10) {
    acquireExecutors(sc, 1, 20000)
  }

  // Wait for 10 seconds to make sure Spark has released all executors again.
  Thread.sleep(10000,0)

  // Start actual test
  for (par <- 1 until 110) {
    acquireExecutors(sc, par, 20000) // 1. Acquire 'par' executors for 20s each.
                                     //    The 'acquireExecutors' function is defined
                                     //    in Listing A.1.
    Thread.sleep(10000,0)            // 2. Wait for 10 seconds to make Spark release
                                     //    executors.
  }
}
```

Listing A.2: *Scala code of application to measure executor acquisition delay*

A.3 Chicle

A.3.1 Applications

This section presents two state-of-the-art distributed ML training algorithms, Local SGD (lSGD) (a recent variant of Mini-batch SGD (mSGD)) and Communication-efficient distributed dual Coordinate Ascent (CoCoA), that have been ported to Chicle.

A.3.1.1 Communication-efficient distributed dual Coordinate Ascent (CoCoA)

CoCoA is a state-of-the-art communication-efficient framework for distributed training of support vector machines (SVMs), linear and logistic regression, lasso and sparse logistic regression models [46, 115]. Its stateful solver tasks execute efficient algorithms such as stochastic coordinate descent (SCD) and stochastic dual coordinate ascent (SDCA) and highly benefit from having random access to large amounts of data. Here, CoCoA was implemented with SCD solvers to train SVMs. Other solvers and models can be used as well by replacing the corresponding functions in CoCoA.

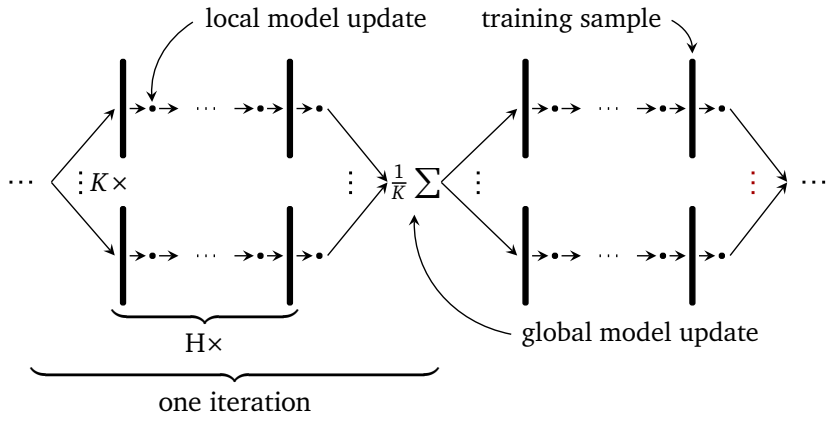


Figure A.7: *Conceptual depiction of CoCoA execution. H represents the number samples processed in each iteration (typically $H = 1/K$-th of the total training data) and K is number of tasks.*

Figure A.7 depicts the execution of CoCoA. In contrast to stateless mSGD solvers, where model updates are only applied to the global model *after* an iteration, in CoCoA, solvers apply updates to their local model *immediately* after "learning" from each training sample. This has the benefit that next samples of the same iteration can already benefit from learning done during previous samples. This allows CoCoA to identify correlations between training samples (and therefore "learn") locally, without the need for global communication and is also the reason why CoCoA benefits from having random access to all data on each node.

Implementation

The implementation of CoCoA on Chicle is based on the reference Spark implementation [125] and uses a CPU-based SCD as local solver to train a SVM. Simplified C++ code of the trainer and solver modules are shown in Listings A.3 and A.5. As CoCoA solves problems with a primal-dual structure, the duality-gap [46], i.e., the difference between the primal and dual solution, is used here as metric to determine how well the algorithm has converged to an optimal solution.

In-Memory Data Chunk Format. For CoCoA, the `CoCoAChunk` class has been derived from Chicle's base `Chunk` class. `CoCoAChunk` stores training samples in sparse vectors[3]. As sparse vectors are not fixed-sized, direct access to samples is not possible but an additional redirection step is required to account for variable offsets to each sample in the chunk. This is realized by a separate `samples` array with a pointer to the sparse vector for each training sample. As pointers are only valid within the task context where they were

[3] A sparse vector implementation was chosen as the evaluated datasets contain sparse one-hot encoded data. A dense vector implementation is possible as well would and result in a simpler, but less memory efficient in-memory representation for many of the selected datasets.

```
void Trainer::run(float target) {
  do {
    wait(event::policies_finished);  // wait until policies have completed
                                     // actions that must not be performed
                                     // during an iteration.
    signal(event::start_iteration);  // start next iteration on all workers
    wait(event::iteration_finished); // wait until all workers have finished

    // The shared vector (model) is updated in the background as each task
    // finishes the current iteration.  At this point, all shared vector updates
    // have already been applied.

    signal(event::update_shared_vector); // signal workers to update their
                                         // shared vector copy.

    float duality_gap = compute_duality_gap(); // compute duality-gap as
                                               // metric for convergence (Listing A.4).

  } while (duality_gap > target)
}
```

Listing A.3: *Simplified C++ code of CoCoA trainer*

```
void Trainer::compute_duality_gap() {
  // All primal and dual objectives are accumulated in the background.

  wait(event::objectives_updated);     // wait until all workers have updated their partial
                                       // primal and dual objectives.

  // get the accumulated values.
  double partial_do = get_partial_dual_objective();
  double partial_po = get_partial_primal_objective();
  double final_do   = 0.0;
  double final_po   = 0.0;

  int num_ft        = get_num_features(); // number of features in the dataset
  for (int ft = 0; ft < num_ft; ft++) {
    final_do += shared_vector[ft] * shared_vector[ft];
  }

  final_po = final_do / (lambda * lambda); // lambda = 0.01 in the experiments.

  final_do = ((final_do / (2.0 * lambda)) - partial_do) / get_num_examples();
  final_po = ((final_po * (0.5 * lambda)) + partial_po) / get_num_examples();

  double duality_gap = final_do + final_po;
}
```

Listing A.4: *Simplified C++ code of CoCoA trainer duality-gap computation.*

```
void Solver::run(Dataset* dataset) {
  // 'done' and 'preempt' are set via RPCs by the trainer to indicate that training
  // or the current iteration is finished.

  double* shared_vector = get_model(); // get initial copy of the shared vector

  while (!done) {
    wait(event::iteration_started);

    int num_samples_processed = 0;
    int num_samples           = dataset->get_num_samples(); // # of local samples

    while (num_samples_processed < num_samples && !preempted) {
      Sample *sample = get_next_sample(); // randomly select next sample from all data
                                          // chunks.  Samples do not repeat within one
                                          // epoch.

      // Perform local training step with the SCD/SVM solver. The shared_vector
      // constitutes a valid update after each iteration of this loop (see Listing A.6).
      local_solver_method(shared_vector, sample);

      num_samples_processed++;
    }

    // Update shared vector and signal that iteration is finished. Tell trainer how
    // many samples have been processed and how many should have been processed.
    send(shared_vector, num_samples_processed, num_samples);
    signal(event::iteration_finished);

    // Compute partial dual objective: O(S) with S = number of samples across all data
    // chunks on this node (see Listing A.7).
    double delta_dual_objective = compute_dual_objective();

    // Wait until the trainer as collected and merged all updates to the shared
    // vector, then pull latest version.
    wait(event::update_shared_vector);
    shared_vector = get_model();

    // Compute partial primal objective: O(S * F) with S as above and F = number of
    // features (see Listing A.8).
    double delta_primal_objective = compute_primal_objective();

    // Send primal and dual objectives to the trainer so that it can compute the
    // duality-gap.
    send(delta_primal_objective, delta_dual_objective);
    signal(event::objectives_updated);
  }
}
```

Listing A.5: *Simplified C++ code of CoCoA solver*

```
void Solver::local_solver_method(double* shared_vector, Sample* ex) {
  double dp   = 0.0;
  double norm = 0.0;

  for (int i = 0; i < ex->size; i++) {
    int    ft   = ex->data[i].feature;
    double val  = ex->data[i].value;
    dp         += val * shared_vector[ft];
    norm       += val * val;
  }

  double label = ex->label;
  double delta = (lambda * label - dp) / sigma / norm;
  double alpha = ex->state + delta;
  double ulim  = (label == +1.0) ? 1.0 :  0.0;
  double llim  = (label == +1.0) ? 0.0 : -1.0;

  alpha        = fmax(fmin(alpha, ulim), llim);
  delta        = alpha - ex->state;
  ex->state   += delta;
  delta       *= sigma;

  for (int i = 0; i < ex->size; i++) {
    int    ft  = ex->data[i].feature;
    double val = ex->data[i].value;

    shared_vector[ft] += val * delta;
  }
}
```

Listing A.6: *Simplified C++ code for the local SCD solver. Based on the CoCoA reference implementation [125].*

```
double compute_dual_objective()
{
    double result = 0.0;

    for (Chunk* chunk : chunks) {
        for (Sample* ex : chunk->examples()) {
            result += ex->state * ex->label;
        }
    }

    return result;
}
```

Listing A.7: *Simplified C++ code to compute the dual objective.*

```
double compute_primal_objective()
{
    double result = 0.0;

    for (Chunk* chunk : chunks) {
        for (Sample* ex : chunk->examples()) {
            double dp = 0.0;
            for (int k = 0; k < ex->width(); k++) {
                long     feature = ex->data[k].feature;
                double   value   = ex->data[k].value;
                dp += (shared_vector[feature] / lambda) * val;
            }
            result += std::max(0.0, 1.0 - (ex->label * dp));
        }
    }

    return result;
}
```

Listing A.8: *Simplified C++ code to compute the primal objective.*

created, the `restore()` method of `CoCoAChunk` restores the `samples` array. Furthermore, SCD requires a single state value for each training sample. As a sample is moved from one node to another, this state has to be moved as well. In `CoCoAChunk` this problem is addressed by storing the state value within the chunk such that it is automatically transferred by Chicle if the sample (via the chunk) is transferred to another node, i.e., chunks do not have to be read-only but can contain mutable state. As with `Chunk`, the assumption is that the in-memory layout of the `CoCoAChunk` data structure is identical on all nodes and that no (de-)serialization (other than what was already mentioned) is required. The `CoCoAChunk` data structures is shown in Listing A.9.

A.3.1.2 Mini-batch and Local Stochastic Gradient Descent (SGD)

The second application that has been ported to Chicle is Local SGD (lSGD) [109], a recent variant of Mini-batch SGD (mSGD) algorithm, arguably the most widely used distributed ML training algorithm. Both algorithms are mainly used for the training of neural network (NN) variants, such as convolutional neural network (CNN), deep neural network (DNN) and recurrent neural network (RNN).

```
struct Datapoint {
    uint32_t feature;       // sparse vector entry index
    float    value;         // sparse vector entry value
}

struct Sample {
    uint32_t   size;        // number of datapoints
    float      label;       // ground truth for this sample
    double     state;       // per-sample state
    Datapoint *data;        // sparse vector
}

class CoCoAChunk : public Chunk {
    // all methods listed in Table 5.1 need to be implemented.

    size_t     size;        // sizeof(data)
    char      *data;        // contiguous memory region containing all data, i.e.,
                            // model, samples, datapoints
    size_t     num_samples; // number of samples
    Sample    *samples;     // pointers to samples in the 'data' array.
}
```

Listing A.9: *CoCoAChunk structure used by Chicle's CoCoA implementation for storing sparse vectors and per sample state.*

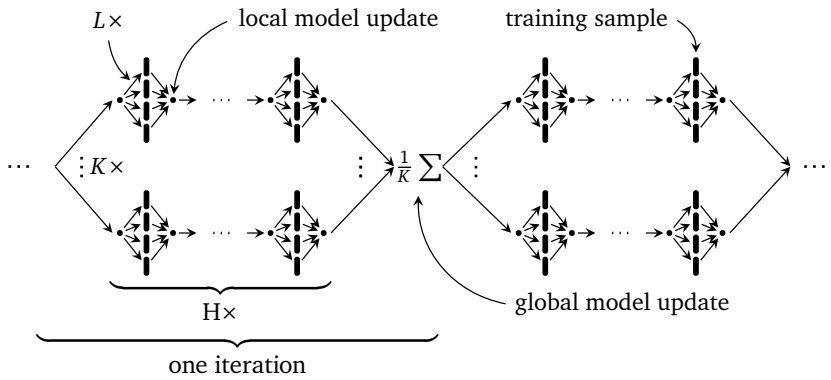


Figure A.8: *Conceptual depiction of mSGD and lSGD. The difference between mSGD and lSGD is that the former sets $H = 1$, i.e., does not perform any local model updates, while the latter sets $H > 1$, i.e., does locally update the model after independently processing L training samples.*

Figure A.8 shows a conceptual representation of mSGD and lSGD.

- In mSGD, $H = 1$ and $K \times L$ samples (= *mini-batch size*) are processed independently. Updates are only applied to the global model.
- In the lSGD variant, $H > 1$ and local updates are applied to local models after L training samples. Global updates occur after $K \times H \times L$ samples (= *global batch size*). Similar to SCD solvers in CoCoA, this allows the training algorithm to benefit

from learning done on previous training samples in the same iteration and can reduce the number of communication steps (iterations) necessary to converge.

Both variants use the same local solver algorithm (SGD), hence lSGD with H=1 is mSGD.

The number of training samples processed per iteration is $K \times H \times L$ in both cases. In both cases, L training samples are processed independently. Typically, $L > 1$ is chosen (e.g., $L = 16$) to take advantage of hardware parallelism on worker nodes and to amortize overheads (e.g. for communication and synchronization) across multiple training samples.

Implementation

The Chicle implementation of mSGD and lSGD is based on libtorch, the C++ backend library of PyTorch [85], that implements various ML algorithms. For instance, it provides implementations for forward and backward propagation for NNs networks, facilities to construct NNs, the SGD algorithm as well as the `Tensor` class[4] used by all provided algorithm implementations. Furthermore, libtorch provides CPU and GPU implementations of ML training algorithms which would allow Chicle utilizing GPUs.

Simplified C++ code of the trainer and solver modules are shown in Listing A.10 and Listing A.11.

```
1  void Trainer::run(float target) {
2    do {
3      wait(event::policies_finished);  // wait until policies have completed
4                                       // actions that must not be performed
5                                       // during an iteration.
6      signal(event::start_iteration);  // start next iteration on all workers
7      wait(event::iteration_finished); // wait until all workers have finished
8
9      // the model is updated in the background as each task finishes the
10     // current iteration.
11
12     float accuracy = compute_test_accuracy(); // compute test accuracy as
13                                               // metric for convergence
14
15   } while (accuracy < target)
16 }
```

Listing A.10: *Simplified C++ code of the mSGD and lSGD trainer.*

[4] Tensors store vectors and matrices.

```
void Solver::run(Dataset *dataset) {
  // 'done' and 'preempt' are set via RPCs by the trainer to indicate that training
  // or the current iteration is finished.

  while (!done) {
    wait(event::iteration_started);
    Model* model = get_model(); // fetch latest global model

    int L = get_L(); // number of samples to process independently
    int H = get_H(); // number of independent sample sets to process sequentially.
                     // H=1 for mini-batch SGD and H>1 for local SGD.

    int num_samples_processed = 0;
    int num_samples           = H * L; // number of sample to process.

    // The PyTorch API does not allow to efficiently train individual samples but
    // requires samples to be pre-grouped into 'Tensor' objects that contain the
    // intended number of samples 'L'.  dataset->restore(L) checks the correct
    // grouping and regroups samples, if necessary (i.e., if 'L' changes).
    dataset->restore(L);

    // without the following while loop, local SGD degrades to mini-batch SGD.
    for (int i = 0; i < H && !preempted; i++) {
      torch::Tensor *sample = get_next_sample(); // randomly select next sample.
                                                 // 'sample' actually contains 'L'
                                                 // training samples, pre-grouped
                                                 // as required by libtorch.

      // Forward propagation
      torch::Tensor output = model->forward(sample->data);
      // Compute loss (prediction error)
      torch::Tensor loss   = torch::loss(output, sample->target);
      // Backward propagation
      loss.backward();
      // Update local model
      sgd.step();

      num_samples_processed += L;
    }

    // Push model updates and signal that the iteration is finished. Tell trainer
    // how many samples have been processed and how many should have been
    // processed.
    send(shared_vector, num_samples_processed, num_samples);
    signal(event::iteration_finished);
  }
}
```

Listing A.11: *Simplified C++ code of the mSGD and lSGD solver.*

In-Memory Data Chunk Format. For mSGD and lSGD based on PyTorch, the `PyChunk` has been derived from Chicle's base `Chunk` class. `PyChunk` stores data such that it can be used with PyTorch's native `Tensor` objects. This is enabled by the fact that `Tensor` objects already store their data in a contiguous memory region and provide accessor functions to this memory where training samples are laid out back-to-back.

Furthermore, `Tensor` objects can be constructed from externally managed copies of that memory without the need to copy the data during construction. Moreover, training samples from multiple `Tensor` objects can be merged into a single `Tensor` object or split up into multiple `Tensor` objects without the need to copy or transform data. This allows efficient storage and transfer of training samples using the mechanisms provided by Chicle.

```
1
2  struct Sample {
3      torch::Tensor data;     // tensor containing training data
4      torch::Tensor label;    // tensor containing labels, i.e., ground truth.
5  }
6
7  class PyChunk : public Chunk {
8      // all methods listed in Table 5.1 need to be implemented.
9
10     void restore(int L);
11
12     size_t     size;        // sizeof(data)
13     char      *data;        // contains all data (samples 'data' and 'label')
14     size_t     num_samples; // number of samples
15     int[4]     dimensions;  // dimensions of the stored tensor objects: number of
16                             // samples per tensor + 3 spacial dimensions.
17     Sample    *samples;     // pointers to samples which are backed by memory in the
18                             // 'data' array. The 'samples' array itself is stored outside
19                             // of the chunk memory and is therefore not transmitted.
20 }
```

Listing A.12: *`PyChunk` class used by mSGD and lSGD to store PyTorch `Tensor` objects.*

The `PyChunk::restore(int L)` method is called by the receiving task to construct `Tensor` objects from the training samples stored in the chunk. As PyTorch requires multiple training samples to be pre-grouped for efficient training, each `Tensor` object is constructed such that it contains L training samples, corresponding to the L parameter of mSGD and lSGD. Furthermore, the `samples` and `dimensions` arrays are restored.

CNNs used in the evaluation

A simple CNN with Rectified Linear Unit (ReLU) activation, composing of two convolutional layers with max-pooling, followed by 3 fully connected layers was used throughout

the evaluation. This CNN was adjusted for the specific dataset for the number of color channels and image dimensions.

Listings A.13 and A.14 show the source code that defines both CNNs used for the CIFAR-10 and Fashion-MNIST datasets respectively. The source code is based on `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

Listing A.13: *Python/PyTorch code that defines the CNN used for the CIFAR-10 dataset.*

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

Listing A.14: *Python/PyTorch code that defines the CNN used for the Fashion-MNIST dataset.*

A.3.2 Description of minor experiments

This section contains brief descriptions for minor experiments of which results are used in the main chapter but haven't been described there.

A.3.2.1 Experiments for Figure 5.1a

The data for this plot was generated using PyTorch and the CNN in Listing A.13 on the CIFAR-10 dataset [10]. The training was performed on a single node with mini-batch sizes of 2^n, with $n = 3 \dots 11$. Training was considered complete once a test accuracy of 61% had been reached. This test accuracy represents the highest test accuracy that was achieved with all mini-batch sizes. The plotted data represents the average number of epochs to converge over 5 training runs for each mini-batch size.

A.3.2.2 Experiments for Figure 5.1b

The data for this plot was generated using CoCoA on Chicle on the Criteo dataset [70] with 16, 32 and 64 data partitions on 16 nodes. Apart from that, the test setup was as described in Section A.3.3. The plotted data represents the average number of epochs to converge over 5 training runs for each partition size.

A.3.2.3 Experiments to evaluate the performance of CoCoA on Spark

Spark is not considered a competitive ML training framework here and not used to compare Chicle against. Dünner et al. [103] have compared Sanp ML against Spark and various other distributed ML training frameworks and shown its inferior performance, albeit using test setups that are not directly comparable. In order to confirm the qualitative statement, that Spark is significantly slower than a C++/MPI-based implementation of CoCoA, the CoCoA reference implementation for Spark [125] is used to train the same four datasets (Table A.18) on the same 16-node test cluster as Chicle (Section A.3.3). The same vanilla Spark version (Section A.1.2.2) and settings (Section A.1.2.3) as in the HCL evaluation was used. As the reference CoCoA implementation was written for an older version of Spark with a slightly different API, minor syntactic changes had to be implemented.

This evaluation compares the iteration time of CoCoA on Spark with CoCoA on Chicle. Results are shown in Table A.16.

Dataset	Chicle	Spark	Difference
Higgs	186ms	2022ms	10.9×
Criteo	1834ms	12501ms	6.8×
KDDA	1336ms	30707ms	23.0×
Webspam	1804ms	26584ms	14.7×

Table A.16: *Iteration runtime comparison of CoCoA on Chicle and on Spark.*

The evaluation results confirm that Spark is significantly (up to 23.0×) slower than Chicle, which makes a fair comparison of uni-tasks vs. micro-tasks impossible.

The reasons for the lower performance of Spark are manifold and include the use of interpreted languages (Scala), data serialization, copy and transfer overhead and inherent properties of Spark's execution model. For instance tasks in Spark cannot explicitly retain state across iterations. Any state that needs to be retained has to be transferred to the driver and broadcast back to tasks for next iteration. In the case of CoCoA, there is per-sample state (one double value) that is only read and updated by the task that processes a specific sample. As tasks process the same samples in every iteration, this state could remain local. In this experiment, the size of the state ranges from 2.8MiB (Webspam) to 368MiB (Criteo). In Chicle, it is stored as part of the data chunk and remains local, except if the data chunk is moved to another task, which happens rarely.

Similar observations w.r.t. the performance of Spark for ML training have also been made for other algorithms [77, 62].

A.3.3 Test setup

All Chicle experiments use the test setup described in the following.

A.3.3.1 Cluster

Table A.17 lists the test cluster configuration. As I did not have full control over the test cluster and had to share it with other users and projects, the cluster used a variety of Linux Distributions and versions. Furthermore, nodes were added and removed during the lifetime of the cluster, due to upgrades and node failures, hence hardware resources are heterogeneous.

Node class	Quantity	CPU	Memory	OS
1	5	Intel Xeon E5-2650v2, 2.60GHz	160GiB	RHEL 7.5
2	4	Intel Xeon E5-2630v3, 2.40GHz	160GiB	RHEL 7.5
3	5	Intel Xeon E5-2640v3, 2.60GHz	256GiB	Fedora 26
4	4	Intel Xeon E5-2640v3, 2.60GHz	256GiB	CentOS 7.5

Table A.17: *Test cluster hardware and OS versions.*

Due to difficulties deploying the libtorch dependency of Chicle and PyTorch across all given OS versions, Chicle and the PyTorch reference application were executed inside Docker containers [49], with a Fedora 28 base image, during all experiments. Due to Message Passing Interface (MPI) deployment issues, the Snap ML reference application was run on the host OS and not inside Docker containers.

All nodes are connected by a 56Gbps Infiniband network via a single Mellanox SX6036 switch. A single control node (from node class 3) runs the Chicle driver and does not participate in any distributed computation. No GPUs were available.

For load balancing tests, all nodes of node class 4 have been slowed down to 1.20GHz. Nodes of other node classes did not allow the manual adjustment of the CPU frequency due to their configuration.

A.3.3.2 Datasets

CoCoA. Table A.18 lists all datasets used in the evaluation of CoCoA. All datasets were acquired from the libsvm project [108] and selected according to the following critea:

- **Size**: The largest datasets that still fit inside a single node's memory have been chosen. This is a limitation of Chicle's data loader.
- **Preprocessing**: Only datasets that did not require preprocessing, e.g., unsupervised label learning, have been selected.

- **Density**: The selected datasets contain such with dense and sparse vectors.

Dataset	Samples	Features	In-mem. size	File size	Chunks	Density
Higgs [41]	11M	28	2.5GiB	7.4GiB	1209	92.11%
KDDA [16]	8.4M	20M	2.6GiB	2.5GiB	1263	1.8e-04%
Webspam [4]	350k	17M	10GiB	24.0GiB	5032	0.02%
Criteo [70]	46M	1M	15GiB	25.0GiB	7346	3.9e-3%

Table A.18: *Overview of datasets used in the evaluation of CoCoA.*

lSGD/mSGD. Datasets used in all lSGD and mSGD experiments are listed in Table A.19. Datasets have been selected as they are commonly used in ML publications [62, 114, 112, 124] and still trainable without GPUs within the test time limit (6 minutes) (which, for instance, excludes the also widely used imagenet dataset).

Dataset	# Samples	# Features	# Cat.	Size	# Chunks
CIFAR-10 [10]	50 k / 10 k	3072	10	586MiB	2381
Fashion-MNIST [89]	60 k / 10 k	784	10	180MiB	723

Table A.19: *Overview of datasets used in the evaluation of mini-batch SGD with number of training/test samples, number of features and categories, as well as the total in-memory size of all training samples and number of data chunks used to store them.*

A.3.4 Supplemental implementation information

This section lists supplemental implementation-related information, such as APIs, event description and minor functions that are used but not described in the main chapter.

A.3.4.1 Event bus events

This section lists events handled by the event bus.

Event	Sender	Receiver	Description
register worker	worker	trainer, policies	First event sent from a worker after it connects to the driver. Contains: worker name and address.
worker registered	trainer	trainer, policies	A worker has registered itself with the system. Contains: worker id and name and address.
worker added	trainer	trainer, policies	A worker was added to the current application. Contains: worker id.
remove worker	resource manager	trainer, policies	A worker will be removed from the current application. Contains: worker id.
worker removed	trainer	trainer, policies	A worker was removed from the current application. Contains: worker id.
connect to worker	trainer	solver	Directs a solver (worker) to connect to another solver (worker). Contains: worker id, worker address.

(a) Worker management

Table A.20: *Overview of the most important event types in Chicle, grouped by function. Sender and receiver modules may coincide as some use the event bus for internal event signaling and coordination across multiple threads.* *Continued on page 258.*

Event	Sender	Receiver	Description
dataset loaded	trainer	trainer, policies	Notifies policies that the training dataset has been loaded.
chunk assigned	trainer	solver, policies	A data chunk was assigned to a solver which is supposed to retrieve it. Contains: chunk id, current location of the chunk
chunk added	trainer, solver	trainer, solver, policies	Confirms that a data chunk has been added to a task. Contains: chunk id, new location of the chunk
remove chunk	trainer	solver, policies	A data chunk should be removed from a task. Contains: chunk id
all chunks assigned	trainer	trainer, policies	Confirms that all data chunks have been assigned to tasks and that training can start/continue.

(b) Chunk management

Table A.20: *Continued from page 257.*

Event	Sender	Receiver	Description
update shared vector	trainer	solver	Direct solver to update the shared vector (model) from the trainer.
start iteration	trainer	solver	Directs solver tasks to start next iteration. Contains: iteration number as well as number of samples to process
task finished	solver	trainer, policies	A task has finished the current iteration. Contains: task id, iteration number, number of processed samples, percentage of completed samples, task runtime.
finish iteration	policy	solver	Instructs a solver to finish the current iteration and return the latest results. Used by the preemption policy to mitigate stragglers.
iteration finished	trainer	trainer, policies	Current iteration has finished. Contains: iteration number.
epoch finished	trainer	trainer, policies	Current epoch has finished. Contains: epoch number.

(c) Training coordination

Table A.20: *Continued from page 258.*

A.3.4.2 Communication subsystem API

Table A.21 lists the API of the communication subsystem used in Chicle. For each peer (trainer, worker), there is one endpoint instance, which defines the source and destination for each API call.

Method	Description
`struct ibv_mr* register(char* region, size_t size)`	Registers a memory region with the remote direct memory access (RDMA) subsystem for one-sided transfers. Returns a pointer needed to manage the region and initiate one-sided operations.
`void deregister(struct ibr_mr *mr)`	De-registers the memory region managed via `mr`.
`void send(Rpc request)`	Serialize and send a request `rpc`.
`void respond(Rpc request, Rpc response)`	Serialize and send a response to a prior request.
`void read(OneSidedRpc request, int worker_id)`	Issue a one-sided RDMA read request to worker `worker_id`.

Table A.21: *Methods provided by Chicle's communication subsystem.*

A.3.4.3 Rebalance (load balancing) policy

This section lists minor functions of the rebalance policy.

```
1 void addRuntime(Worker* worker, float runtime) {
2   // worker->runtimes is a ring-buffer of size I and represents the sliding window.
3   worker->runtimes.push_back(runtime);
4   // 'median' computes the median of all values stored in the ring-buffer.
5   worker->medianTaskRuntime = median(worker->runtimes);
6 }
```

Listing A.15: *Simplified C++ code of the rebalance policy's task runtime prediction algorithm.*

```
void adjustRuntime(Worker* worker, float runtime) {
  // Runtime is adjusted for all values in the history such that the new median task
      runtime
  // reflects the to be expected runtime with a changed number of chunks.
  for (int i = 0; i < I; i++)
    worker->runtimes[i] += runtime;

  worker->medianTaskRuntime = median(worker->runtimes);
}
```

Listing A.16: *Simplified C++ code of the rebalance policy's task runtime prediction algorithm.*

```
void workerFinishedEventHandler(int iteration, Worker* worker,
                                float completionRate, float runtime) {
  // Straggler mitigation works by preempting running tasks.  A preempted task processed
  // fewer samples than planned, making the measured runtime misleading.  Correct measured
  // runtime (τ_k) for the completed fraction (σ_k) of training samples.
  float normalizedRuntime = runtime / completionRate; // T'_k = τ_k / σ_k

  // Add latest runtime measurement to the sliding window and update median task runtime.
  addRuntimes(worker, normalizeRuntime);
}
```

Listing A.17: *Simplified C++ code of the rebalance policy's event handlers.*

```
void iterationFinishedEventHandler(int iteration) {
  // initiate chunk rebalancing, if necessary
  rebalance();

  // signal trainer that rebalancing has been done and it can start the next iteration.
}
```

Listing A.18: *Simplified C++ code of the rebalance policy's event handlers.*

Bibliography

[1] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[2] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[3] H. Topcuoglu, S. Hariri, and Min-You Wu. "Task scheduling algorithms for heterogeneous processors". In: *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*. 1999, pp. 3–14. DOI: 10.1109/HCW.1999.765092.

[4] Steve Webb, James Caverlee, and Calton Pu. "Introducing the Webb Spam Corpus: Using Email Spam to Identify Web Spam Automatically." In: *CEAS*. 2006.

[5] Michael Isard et al. "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. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[6] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM*. OSDI'04 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[7] L Pinedo Michael. *Scheduling: theory, algorithms, and systems*. Springer, 2008.

[8] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 29–42. URL: http://dl.acm.org/citation.cfm?id=1855741.1855744.

[9] Michael Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 261–276. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629601. URL: http://doi.acm.org/10.1145/1629575.1629601.

[10] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[11] Bernard Metzler, Fredy Neeser, and Philip Frey. "Softiwarp". In: *Open Fabrics Alliance Sonoma Workshop*. 2009.

[12] Ashish Thusoo et al. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1626–1629. ISSN: 2150-8097. DOI: 10.14778/1687553.1687609. URL: http://dx.doi.org/10.14778/1687553.1687609.

[13] Ganesh Ananthanarayanan et al. “Reining in the Outliers in Map-reduce Clusters Using Mantri”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 265–278. URL: http://dl.acm.org/citation.cfm?id=1924943.1924962.

[14] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. “DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm”. In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Feb. 2010, pp. 27–34. DOI: 10.1109/PDP.2010.56.

[15] S Sonnenburg et al. “The SHOGUN machine learning toolbox”. In: *Journal of Machine Learning Research* 11.Jun (2010), pp. 1799–1802.

[16] Hsiang-Fu Yu et al. “Feature engineering and classifier ensemble for KDD cup 2010”. In: *KDD Cup*. 2010.

[17] Matei Zaharia et al. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling”. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys ’10. Paris, France: ACM, 2010, pp. 265–278. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[18] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[19] Alekh Agarwal and John C Duchi. “Distributed delayed stochastic optimization”. In: *Advances in Neural Information Processing Systems*. 2011, pp. 873–881.

[20] Dong Chen et al. “The IBM Blue Gene/Q interconnection network and message unit”. In: *SC’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, pp. 1–10.

[21] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” In: *NSDI*. Vol. 11. 2011, pp. 22–22.

[22] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[23] Benjamin Recht et al. “Hogwild: A lock-free approach to parallelizing stochastic gradient descent”. In: *Advances in neural information processing systems*. 2011, pp. 693–701.

[24] Sameer Agarwal et al. “Reoptimizing data parallel computing”. In: *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 281–294.

[25] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[26] Andrew D Ferguson et al. “Jockey: guaranteed job latency in data parallel clusters”. In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, pp. 99–112.

[27] Zhonghong Ou et al. “Exploiting hardware heterogeneity within the same instance type of Amazon EC2”. In: *Presented as part of the*. 2012.

[28] Charles Reiss et al. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 7.

[29] Alexey Tumanov et al. “Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: ACM, 2012, 25:1–25:7. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391254. URL: http://doi.acm.org/10.1145/2391229.2391254.

[30] Ganesh Ananthanarayanan et al. “Effective Straggler Mitigation: Attack of the Clones.” In: *NSDI*. Vol. 13. 2013, pp. 185–198.

[31] James Cipar et al. “Solving the Straggler Problem with Bounded Staleness”. In: *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. Santa Ana Pueblo, NM: USENIX, 2013. URL: https://www.usenix.org/conference/hotos13/solving-straggler-problem-bounded-staleness.

[32] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[33] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-aware scheduling for heterogeneous datacenters”. In: *ACM SIGPLAN Notices*. Vol. 48. 4. ACM. New York, NY, USA: ACM, Mar. 2013, pp. 77–88. DOI: 10.1145/2499368.2451125. URL: http://doi.acm.org/10.1145/2499368.2451125.

[34] Qirong Ho et al. “More effective distributed ml via a stale synchronous parallel parameter server”. In: *Advances in neural information processing systems*. 2013, pp. 1223–1231.

[35] Derek G. Murray et al. “Naiad: A Timely Dataflow System”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738. URL: http://doi.acm.org/10.1145/2517349.2522738.

[36] Kay Ousterhout et al. “Sparrow: distributed, low latency scheduling”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. ACM. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[37] Kay Ousterhout et al. “The Case for Tiny Tasks in Compute Clusters.” In: *Proceedings of the 14th Workshop on Hot Topics in Operating Systems*. Vol. 13. HotOS ’16. Santa Ana Pueblo, NM: USENIX, 2013. URL: https://www.usenix.org/conference/hotos13/case-tiny-tasks-compute-clusters.

[38] Malte Schwarzkopf et al. “Omega: Flexible, Scalable Schedulers for Large Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386. URL: http://doi.acm.org/10.1145/2465351.2465386.

[39] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[40] Hamid Arabnejad and Jorge G Barbosa. “List scheduling algorithm for heterogeneous systems by an optimistic cost table”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.3 (2014), pp. 682–694.

[41] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. “Searching for exotic particles in high-energy physics with deep learning”. In: *Nature communications* 5 (2014), p. 4308.

[42] Eric Boutin et al. “Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 285–300. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

[43] Henggang Cui et al. “Exploiting Bounded Staleness to Speed Up Big Data Analytics”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 37–48. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui.

[44] Christina Delimitrou and Christos Kozyrakis. “Quasar: resource-efficient and QoS-aware cluster management”. In: *ACM SIGPLAN Notices*. ASPLOS ’14 49.4 (2014), pp. 127–144. DOI: 10.1145/2541940.2541941. URL: http://doi.acm.org/10.1145/2541940.2541941.

[45] Robert Grandl et al. “Multi-resource packing for cluster schedulers”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 455–466.

[46] Martin Jaggi et al. “Communication-efficient distributed dual coordinate ascent”. In: *Advances in neural information processing systems*. 2014, pp. 3068–3076.

[47] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server.” In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[48] Yucheng Low et al. “Graphlab: A new framework for parallel machine learning”. In: *arXiv preprint arXiv:1408.2041* (2014).

[49] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.

[50] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. “Wrangler: Predictable and faster jobs using fewer resources”. In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[51] Martn Abadi et al. *TensorFlow: Large-scale machine learning on heterogeneous systems, 2015*. Tech. rep. 2015.

[52] Paris Carbone et al. “Apache Flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[53] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[54] Pamela Delgado et al. “Hawk: Hybrid Datacenter Scheduling”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 499–510. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/delgado.

[55] Ionel Gog et al. “Musketeer: All for One, One for All in Data Processing Systems”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 2:1–2:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741968. URL: http://doi.acm.org/10.1145/2741948.2741968.

[56] Konstantinos Karanasos et al. “Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters”. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 485–497. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos.

[57] Frank McSherry, Michael Isard, and Derek G. Murray. “Scalability! But at what COST?” In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

[58] Kay Ousterhout et al. “Making sense of performance in data analytics frameworks”. In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 293–307.

[59] Bikas Saha et al. “Apache TEZ: A unifying framework for modeling and building data processing applications”. In: *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*. ACM. 2015, pp. 1357–1369.

[60] Abhishek Verma et al. “Large-scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 18:1–18:17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964. URL: http://doi.acm.org/10.1145/2741948.2741964.

[61] Kewen Wang and Mohammad Maifi Hasan Khan. “Performance prediction for apache spark platform”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE. 2015, pp. 166–173.

[62] Martín Abadi et al. “Tensorflow: a system for large-scale machine learning.” In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[63] *Amazon EC2 Pricing*. https://aws.amazon.com/ec2/pricing/. https://aws.amazon.com/ec2/pricing/: Amazon, Mar. 2016. URL: https://aws.amazon.com/ec2/pricing/.

[64] Brendan Burns et al. “Borg, omega, and kubernetes”. In: *Queue* 14.1 (2016), p. 10.

[65] Ionel Gog et al. “Firmament: Fast, Centralized Cluster Scheduling at Scale”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, Nov. 2016, pp. 99–115. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog.

[66] Robert Grandl et al. “Altruistic scheduling in multi-resource clusters”. In: *Proceedings of OSDIâĂŹ16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 2016, p. 65.

[67] Robert Grandl et al. “GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, Nov. 2016, pp. 81–97. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_graphene.

[68] Aaron Harlap et al. “Addressing the straggler problem for iterative convergent parallel ML”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM. 2016, pp. 98–111.

[69] Scott Hendrickson et al. “Serverless Computation with openLambda”. In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’16. Denver, CO: USENIX Association, 2016, pp. 33–39. URL: http://dl.acm.org/citation.cfm?id=3027041.3027047.

[70] Yuchin Juan et al. “Field-aware factorization machines for CTR prediction”. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM. 2016, pp. 43–50.

[71] Nitish Shirish Keskar et al. “On large-batch training for deep learning: Generalization gap and sharp minima”. In: *arXiv preprint arXiv:1609.04836* (2016).

[72] David Lion et al. “Don’t Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems.” In: *OSDI*. 2016, pp. 383–400.

[73] Jean-Pierre Lozi et al. “The Linux scheduler: a decade of wasted cores”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 1.

[74] Animesh Trivedi et al. “On the [ir] relevance of network performance for data processing”. In: *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.

[75] Alexey Tumanov et al. “TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys ’16. London, United Kingdom: ACM, 2016, 35:1–35:16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901355. URL: http://doi.acm.org/10.1145/2901318.2901355.

[76] Jagath Weerasinghe et al. “Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers”. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2016, pp. 9–17.

[77] Jinliang Wei, Jin Kyu Kim, and Garth A Gibson. “Benchmarking Apache Spark with Machine Learning Applications”. In: *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA* (2016).

[78] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: http://doi.acm.org/10.1145/2934664.

[79] F. Abel et al. “An FPGA Platform for Hyperscalers”. In: *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*. Aug. 2017, pp. 29–32. DOI: 10.1109/HOTI.2017.13.

[80] Celestine Dünner et al. “Understanding and optimizing the performance of distributed machine learning applications on apache spark”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 331–338.

[81] Priya Goyal et al. “Accurate, large minibatch SGD: training imagenet in 1 hour”. In: *arXiv preprint arXiv:1706.02677* (2017).

[82] Aaron Harlap et al. “Proteus: agile ml elasticity through tiered reliability in dynamic resource markets”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 589–604.

[83] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2017, pp. 1–12.

[84] Michael Kaufmann and Kornilios Kourtis. “The HCl Scheduler: Going all-in on Heterogeneity”. In: *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*. HotCloud’17. Santa Clara, CA: USENIX Association, July 2017. URL: https://www.usenix.org/conference/hotcloud17/program/presentation/kaufmann.

[85] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[86] Samuel L Smith et al. “Don’t decay the learning rate, increase the batch size”. In: *arXiv preprint arXiv:1711.00489* (2017).

[87] Ehsan Totoni, Subramanya R Dulloor, and Amitabha Roy. “A Case Against Tiny Tasks in Iterative Analytics”. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. ACM. Whistler, BC, Canada: ACM, 2017, pp. 144–149. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3103004. URL: http://doi.acm.org/10.1145/3102980.3103004.

[88] TPC. *TPC BenchmarkâĎćDS (TPC-DS)*. http://www.tpc.org/tpcds/. http://www.tpc.org/tpcds/, Mar. 2017. URL: http://www.tpc.org/tpcds/.

[89] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[90] Haoyu Zhang et al. “SLAQ: Quality-driven Scheduling for Distributed Machine Learning”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 390–404. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3127490. URL: http://doi.acm.org/10.1145/3127479.3127490.

[91] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 2018.

[92] Apache. *Apache Apex*. accessed 2018/08/18. 2018. URL: https://apex.apache.org.

[93] Apache. *Apache Flink Jira 4319*. accessed 2018/08/06. 2018. URL: https://issues.apache.org/jira/browse/FLINK-4319.

[94] Apache. *Apache Hadoop*. accessed 2018/06/23. 2018. URL: https://hadoop.apache.org/.

[95] Apache. *Apache OpenWhisk*. accessed 2018/06/23. 2018. URL: https://openwhisk.apache.org/.

[96] Apache. *Apache Pig*. accessed 2018/08/16. 2018. URL: https://pig.apache.org/.

[97] Apache. *Apache Spark Job Scheduling*. accessed 2018/08/06. 2018. URL: https://spark.apache.org/docs/2.2.1/job-scheduling.html.

[98] Apache. *Apache Spark Standalone*. accessed 2018/08/18. 2018. URL: https://spark.apache.org/docs/2.2.1/spark-standalone.html.

[99] Apache. *Apache Storm*. accessed 2018/08/06. 2018. URL: http://storm.apache.org.

[100] Apache. *Apache Storm Jira 2284*. accessed 2018/08/06. 2018. URL: https://issues.apache.org/jira/browse/STORM-2284.

[101] Ryan R Curtin et al. “mlpack 3: a fast, flexible machine learning library.” In: *J. Open Source Software* 3.26 (2018), p. 726.

[102] Celestine Dünner et al. “A Distributed Second-Order Algorithm You Can Trust”. In: *arXiv preprint arXiv:1806.07569* (2018).

[103] Celestine Dünner et al. “Snap ML: A hierarchical framework for machine learning”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 250–260.

[104] Sanghamitra Dutta et al. “Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD”. In: *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. Ed. by Amos Storkey and Fernando Perez-Cruz. Vol. 84. Proceedings of Machine Learning Research. Playa Blanca, Lanzarote, Canary Islands: PMLR, Sept. 2018, pp. 803–812. URL: http://proceedings.mlr.press/v84/dutta18a.html.

[105] Nikolas Ioannou et al. “Parallel training of linear models without compromising convergence”. In: *NeurIPS 2018 Systems for ML workshop* (2018).

[106] Michael Kaufmann, Thomas Parnell, and Kornilios Kourtis. “Elastic CoCoA: Scaling In to Improve Convergence”. In: *NeurIPS 2018 Systems for ML workshop* (Dec. 2018).

[107] Michael Kaufmann et al. “Mira: Sharing Resources for Distributed Analytics at Small Timescales”. In: *Big Data (Big Data), 2018 IEEE International Conference on*. IEEE. 2018.

[108] *LibSVM*. accessed 2019/08/17. 2018. URL: https://www.csie.ntu.edu.tw/~cjlin/libsvm/.

[109] Tao Lin, Sebastian U Stich, and Martin Jaggi. “Don’t Use Large Mini-Batches, Use Local SGD”. In: *arXiv preprint arXiv:1808.07217* (2018).

[110] Hongzi Mao et al. “Learning scheduling algorithms for data processing clusters”. In: *arXiv preprint arXiv:1810.01963* (2018).

[111] Edward Oakes et al. “SOCK: Rapid Task Provisioning with Serverless-Optimized Containers”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[112] Yanghua Peng et al. “Optimus: an efficient dynamic resource scheduler for deep learning clusters”. In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 3.

[113] Liudmila Prokhorenkova et al. “CatBoost: unbiased boosting with categorical features”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 6638–6648.

[114] Aurick Qiao et al. “Litz: Elastic Framework for High-Performance Distributed Machine Learning”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 631–644. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/qiao.

[115] Virginia Smith et al. “CoCoA: A general framework for communication-efficient distributed optimization”. In: *Journal of Machine Learning Research* 18 (2018), p. 230.

[116] Animesh Trivedi. *TPC-DS Benchmark Suite for Spark*. accessed 2018/08/16. 2018. URL: https://github.com/zrlio/sql-benchmarks.

[117] Apache. *Apache Parquet*. accessed 2019/06/07. 2019. URL: https://parquet.apache.org/.

[118] Apache. *Apache Spark Tuning Guide*. accessed 2019/08/17. 2019. URL: http://spark.apache.org/docs/2.2.1/tuning.html#level-of-parallelism.

[119] Boost. *Boost C++ Library*. accessed 2019/05/10. 2019. URL: https://www.boost.org.

[120] IBM. *Platform Load Sharing Facility*. accessed 2019/07/05. 2019. URL: https://www.ibm.com/us-en/marketplace/hpc-workload-management.

[121] Intel. *Intel Core i9-7980XE Specification*. accessed 2019/06/15. 2019. URL: https://ark.intel.com/content/www/us/en/ark/products/126699/intel-core-i9-7980xe-extreme-edition-processor-24-75m-cache-up-to-4-20-ghz.html.

[122] Can Karakus et al. "Redundancy Techniques for Straggler Mitigation in Distributed Optimization and Learning." In: *Journal of Machine Learning Research* 20.72 (2019), pp. 1–47.

[123] NVidia. *NVIDIA Turing GPU Architecture Whitepaper*. accessed 2019/06/15. 2019. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[124] Christopher J Shallue et al. "Measuring the Effects of Data Parallelism on Neural Network Training". In: *Journal of Machine Learning Research* 20.112 (2019), pp. 1–49.

[125] Virginia Smith. *CoCoA for Spark*. accessed 2019/05/10. 2019. URL: https://github.com/gingsmith/cocoa.

[126] *SoftRoCE*. accessed 2019/06/15. 2019. URL: https://github.com/SoftRoCE.

[127] Spark. *Spark Standalone Resource Scheduling*. accessed 2019/06/07. 2019. URL: https://spark.apache.org/docs/2.2.1/spark-standalone.html#resource-scheduling.

[128] *Top 500 list*. accessed 2019/08/14. 2019. URL: https://top500.org/list/2019/06/.

[129] Animesh Trivedi. *TPC-DS Parquet File Generator*. accessed 2019/06/07. 2019. URL: https://github.com/zrlio/parquet-generator.

[130] *Softlayer GPU Accelerated Computing*. http://www.softlayer.com/GPU. URL: http://www.softlayer.com/GPU.