



Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Performanceanalyse von Big-Data-Anwendungen in Spark unter Nutzung der POP-Methodik

Performance Analysis using POP Methodology in Spark Big Data Applications Bachelorarbeit

communicated by Prof. Matthias S. Müller

Moritz Brückner
Matrikelnummer: 412238

Aachen, den 29. März 2023

Erstgutachter: Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter: Prof. Dr. rer. nat. Sandra Geisler (*)
Betreuer: Radita Tapaning Hesti Liem, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University

(*) Lehrstuhl für Informationssysteme und Datenbanken,
RWTH Aachen University

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 29. März 2023

Kurzfassung

Heutige Anwendungsprogramme müssen immer größeren Datenmengen gerecht werden und diese in einem akzeptablen Zeitrahmen mit limitierten Ressourcen verarbeiten können. Um diesen Ansprüchen nachzukommen, werden vielfach Frameworks wie *Apache Hadoop* oder *Apache Spark* verwendet, mit deren Hilfe eine Anwendung verteilt in einem Cluster-Netzwerk parallel ausgeführt werden kann.

Ein häufiges Problem solcher Frameworks ist, dass sowohl die Konfiguration der Anwendung als auch die Art der Anwendung und die Struktur ihrer Daten einen großen, schwer einzuschätzenden Einfluss auf die Performance der Anwendung hat. Zudem scheint sich der Trend abzuzeichnen, dass sich die ursprünglich größtenteils unabhängigen Disziplinen des Hochleistungsrechnens (HPC) und dem Rechnen mit Massendaten (Big Data) langsam annähern und immer mehr überlagern. Dadurch gewinnt die Anwendung von Apache Spark auf HPC-Systemen an Relevanz und infolgedessen auch die Untersuchung der Performance von Spark-Applikationen auf ebendiesen Systemen.

In dieser Arbeit wird die POP-Methodik, die ursprünglich für die Performanceanalyse von HPC-Anwendungen entwickelt wurde, auf Big-Data-Applikationen in Apache Spark angewandt. Kern der POP-Methodik ist es, einzelnen, die Performance beeinflussenden Aspekten einen Score zuzuordnen, mit dessen Hilfe ein umfassender und direkter Überblick über potenzielle Performance-Probleme einer Anwendung erlangt werden kann. Ziel dieser Arbeit ist die Untersuchung ausgewählter Spark-Benchmarks der HiBench-Benchmark-Suite, um mit den daraus gewonnenen Ergebnissen POP-Metriken für Spark-Applikationen abzuleiten. Zusätzlich zu den im HPC-Kontext verwendeten POP-Metriken werden weitere, Spark-spezifische Metriken vorgeschlagen, mit deren Hilfe die Bandbreite erkennbarer Probleme deutlich erweitert und eine präzisere Bestimmung ebendieser Probleme ermöglicht wird.

Diese Arbeit kommt zu dem Ergebnis, dass sich die POP-Methodik grundsätzlich erfolgreich auf Spark-Applikationen anwenden lässt, wenngleich in bestimmten Fällen gewisse Einschränkungen oder Annahmen notwendig sind. Auch wenn sich die Metriken durch die in dieser Arbeit durchgeführten Experimente nicht zweifelsfrei in ihrer Korrektheit und Vollständigkeit verifizieren lassen, scheint die hier vorgestellte Methodik geeignet zu sein, eine große Anzahl verschiedener Performance-Probleme zu erkennen. Dennoch sind weitere Untersuchungen notwendig, um einige der vorgenommenen Einschränkungen bzw. Annahmen zu eliminieren und sowohl einzelne Metriken als auch die Methodik insgesamt weiter zu verbessern und zu validieren.

Stichwörter: Apache Spark, POP-Methodik, Performanceanalyse, Big Data, High-Performance Computing, HiBench

Abstract

Today's software applications need to cope with ever increasing amounts of data while processing the data in a reasonable amount of time with limited resources. Specialized frameworks such as *Apache Hadoop* or *Apache Spark* are often used to meet those requirements, making it possible to run an application in a distributed and parallel manner on multiple compute nodes in a cluster network.

A common issue with these frameworks is that both the configuration of an applications as well as the kind of application and the structure of its data are strongly influencing the application's performance. In addition to that, there seems to be a current trend of convergence of the originally largely independent disciplines of high-performance computing (HPC) and big data, whose applications increasingly overlap. As a result, the application of Apache Spark on HPC systems is gaining relevance and, consequently, also the study of performance of Spark applications on these systems.

In this thesis, the POP methodology, originally developed for analyzing the performance of HPC applications, is applied to Spark big data applications. The core principle of the methodology is to assign a score to individual performance-influencing aspects, which can be used to obtain a comprehensive and direct overview of potential performance bottlenecks of an application. The aim of this thesis is to evaluate selected Spark benchmarks from the HiBench benchmark suite and to use the obtained results to derive POP metrics for Spark applications. Beyond the POP metrics that are used in the HPC context, additional Spark-specific metrics are proposed in order to significantly extend the range of identifiable problems and to allow for a more precise determination of these problems.

This thesis comes to the conclusion that, in principle, the POP methodology can be successfully applied to Spark applications, although in some cases certain limitations or assumptions are necessary. Even though it is not possible to verify the correctness and completeness of the proposed metrics beyond any doubt by means of the conducted experiments, the methodology presented in this thesis seems to be suitable for identifying a large number of different performance problems. Yet, further investigations are required in order to eliminate some of the limitations and assumptions made, and to improve and validate both individual metrics as well as the methodology as a whole.

Keywords: Apache Spark, POP Methodology, Performance Analysis, Big Data, High-Performance Computing, HiBench

Acknowledgements

Simulations were performed with computing resources granted by RWTH Aachen University under project thes1318.

Contents

List of Figures	xiii
List of Tables	xv
1. Introduction	1
2. Background	5
2.1. Apache Spark	5
2.2. Apache YARN	7
2.3. Hadoop Distributed File System (HDFS)	8
2.4. POP Methodology	8
3. Experimental Setup	13
3.1. Workloads	13
3.1.1. WordCount	14
3.1.2. PageRank	15
3.1.3. K-means Clustering	15
3.2. Configuration	17
3.3. Hardware	21
3.4. Software	21
4. Implementing the POP Methodology	23
4.1. Data Extraction	23
4.2. Thread Scheduling	24
4.3. Calculating POP Metrics for Spark Applications	29
4.3.1. POP Standard Metrics	30
4.3.2. Spark-specific POP Metrics	34
4.3.2.1. Cause-identifying POP Metrics	37
5. Workload Performance Analysis	43
5.1. Experiment Results	43
5.2. Limitations	46
6. Summary and Outlook	49
6.1. Summary	49
6.2. Outlook	49
A. Code Snippets	51

B. Experimental Results	55
B.1. WordCount	55
B.1.1. hibench.scale.profile: huge	55
B.1.2. hibench.scale.profile: gigantic	57
B.2. PageRank (hibench.scale.profile: huge)	58
B.3. K-means	61
B.3.1. hibench.scale.profile: huge	61
B.3.2. hibench.scale.profile: gigantic	64
Bibliography	69

List of Figures

2.1.	Example lineage graph of a Spark job	6
2.2.	Logical breakdown of a Spark application	7
2.3.	Physical architecture of a Spark application	7
2.4.	Hierarchical overview of all POP1 standard metrics	9
2.5.	Minimum waiting time for two processes p_1, p_2	10
2.6.	Transfer time for two processes p_1, p_2 using the example of a synchronization barrier	11
3.1.	The DAG constructed for the WordCount benchmark as shown in the Spark History Server Web UI	14
3.2.	The DAG constructed for the PageRank benchmark as shown in the Spark History Server Web UI	15
3.3.	The DAGs constructed for the k-means benchmark as shown in the Spark History Server Web UI	16
a.	Job 0: <code>count</code>	16
b.	Job 1: <code>takeSample</code>	16
c.	Job 2: <code>takeSample</code>	16
d.	Jobs 3-7: <code>collectAsMap</code>	16
e.	Job 8: <code>collect</code>	16
3.4.	Conceptual breakdown of the JVM heap space	19
4.1.	Anatomy of a task	27
4.2.	Example result of the simulated thread scheduler for the first stage of the WordCount benchmark	29
4.3.	Hierarchical overview of all POP efficiencies for Spark applications	31
4.4.	Amount of minor and major GC invocations per stage based on the memory fraction	36
5.1.	Task scheduling for the first stage of experiment [E6]	45
B.1.	Colormap used for printing POP scores in the appendix	55

List of Tables

- 3.1. List of configuration variables used for the experiments 17
- 3.2. `hibench.scale.profile` and the corresponding `Bytes Read` metric for
the first stage of the tested workloads 20

- 4.1. Task attributes used in this thesis 25
- 4.2. Thread attributes used in this thesis 26
- 4.3. Stage attributes used in this thesis 26
- 4.4. List of all Spark task locality levels 39

1. Introduction

The advent of big internet platforms as well as the growing interest in artificial intelligence in recent years have contributed to the rise of the field of big data, which is almost ubiquitous in today’s world and finds applications in various disciplines (cf. [1], [2]). New challenges arose for many applications as they need to cope with enormous and continuously growing amounts of data while processing the data in a reasonable amount of time with limited computational resources. For these difficult tasks, specialized distributed big data frameworks such as *Apache Hadoop* and *Apache Spark* were developed, making it possible to run an application in a distributed manner on multiple compute nodes in a cluster network in order to distribute the application’s load.

The performance of Spark applications strongly depends on their configuration as well as the used algorithms and implementations; in particular, “running a Spark application on all the available processors does not necessarily imply lower running time” [3]. Even if individual configuration parameters are understood very well in isolation, their impact “may vary from application to application and [...] from cluster to cluster” [4], and multiple parameters are often correlated [4], [5]. This makes it difficult to tune or even predict performance [4], [6], and despite recent scientific efforts on performance prediction of Spark applications (see p. 2), tuning performance by trial and error on actual applications still appears to be the way to go. Over the years, Spark introduced many performance optimization techniques and is constantly improving, however, “its configuration and fine-tuning remains a big challenge” [3]. So far, there only exist few approaches to systematically identify performance bottlenecks of arbitrary Spark applications (cf. [7]).

To fill in parts of this gap, this thesis presents a proof of concept on how the *POP methodology* [8] that was originally developed for applications in the field of high-performance computing (HPC) can be applied to arbitrary applications running in Apache Spark. The core idea of the POP methodology is to calculate efficiencies (*metrics*) for all performance-influencing aspects of an application that describe how much efficiency is lost due to each of these aspects. The resulting efficiencies give a clear and comparable overview about possible performance losses and provide a basis for further performance investigations.

In addition to that, this thesis presents an approach to deploy Spark on *CLAIX-2018*, the HPC compute cluster of the RWTH Aachen University. Hadoop and Spark applications are often run on low-cost *shared-nothing*¹ [9] *commodity hardware* [2], [10], but HPC clusters with more powerful hardware as well as shared memory

¹ I.e., nodes do not share memory or storage with each other.

1. Introduction

and storage appear to become more prominent targets for big data computing in recent years [2], [11]. Research suggests that Hadoop applications might benefit from *scaling-up* a cluster (i.e., improving the cluster hardware) instead of *scaling-out* (that is, adding more hardware to the cluster) [12], and some argue that on today’s clusters, disk locality—which is an advantage of shared-nothing clusters over HPC clusters—does not matter anymore as networking speeds are sufficiently fast [13]. Adapting the POP methodology to Spark applications as presented in this thesis makes it possible to systematically evaluate the effects of different cluster hardware on Spark applications and hopefully enables further research on this topic.

Related Work. Lots of research has been done already to evaluate the performance of Spark applications and to compare it to other big data frameworks like Hadoop, Flink, HAMR, and DataMPI [5], [14]–[19]. Whereas some studies mainly compare and investigate timings or hardware usage [15], [19], [20], others make use of dedicated tools for analysis such as *SparkMeasure* [21]–[23], *Ganglia* [24], [25], or other custom tools and visualizations [14], [26], [27]. Also due to the sheer endless complexity of possible combinations of Spark parameters, many recent publications deal with developing models to predict the performance of Spark applications, sometimes by means of machine learning [6], [28], [29]. Similar techniques have been developed for Hadoop (e.g., self-tuning configurations [30]).

Contrary to what one might at first expect from a data-intensive framework like Apache Spark, a lot of research comes to the conclusion that Spark applications are often CPU-bound [7], [14], [25], [31]. For instance, Ousterhout, Rasti, Ratnasamy, *et al.* claim that network optimizations can only improve the run time of Spark applications “by a median of at most 2%” [7]. Nevertheless, disk or network I/O can be a bottleneck in some cases as well ([14] resp. [25], [32]–[34]), and some papers find entirely different bottlenecks such as garbage collection [35]. Following this—as well as the fact that individual scientific publications usually evaluate very specific configurations for specific workloads under specific conditions (e.g., hardware)—, it is clear that one needs to consider the entire range of possible bottlenecks when developing a methodology to recognize performance issues for arbitrary Spark applications.

There exist many benchmarking suites for Spark, making it possible to test and compare various Spark workloads with similar characteristics as real-world applications. The work presented in this thesis makes use of *HiBench* [36], which, like *BigDataBench* [37], is a benchmarking tool that supports Spark among various other big data frameworks. *SparkBench* [38] is a benchmarking suite specifically designed for Spark applications.

The POP metrics [8] were originally developed for HPC applications (e.g., applications using MPI or OpenMP) and so far have not been adapted to applications outside of the HPC context. For both HPC and big data applications, however, there exist similar approaches (e.g., *PerfExpert* [39] for HPC). For Spark applications, Wang and Khan proposed some metrics in [6] for the sake of estimating and predicting performance that work similar to the POP metrics, and Nguyen, Khan,

Albayram, *et al.* developed a framework to extract performance metrics from Spark applications that recognizes significant execution changes “in response to changes in configuration settings” [40].

Thesis Structure. Chapter 2 contains a brief introduction to Apache Spark and other components from the Hadoop ecosystem that are important for this thesis, as well as the POP methodology that is used as the basis of the proposed performance evaluation framework. Next, I describe the setup and configuration that I used for my experiments on the CLAIX-2018 cluster (Chapter 3). In Chapter 4, I present how performance data can be extracted from Spark event logs and how the obtained information is used to derive meaningful POP metrics for Spark applications (using both the original POP metrics as well as new, Spark-specific ones). These metrics are motivated both by existing research as well as results from the experiments. In Chapter 5, I summarize and discuss my key findings from the experiments, followed by a brief conclusion of the thesis and a discussion of open research questions in Chapter 6.

2. Background

2.1. Apache Spark

The *Apache Spark* framework [41] is a distributed big data framework for the JVM (*Java Virtual Machine*) within the *Apache Hadoop* [42] ecosystem and is considered “one of the most active open-source Apache projects with a huge [...] community” [3]. It was developed as an alternative and extension to Hadoop’s implementation of the *MapReduce* algorithm [43], aiming to achieve faster execution times especially for iterative jobs in which a function is applied “repeatedly to the same dataset to optimize a parameter” [44]. A major flaw of Hadoop MapReduce is that working data cannot be cached in memory, so in each iteration the data must be reloaded from disk [44]. As a result, there are high latencies involved when computing iterative algorithms [44], [45]. To bypass this issue, Spark introduced the concept of so-called *resilient distributed datasets* (RDDs) whose data can be stored in memory or written to disk [44], [46]. RDDs are immutable collections of so-called *records*, which are serializable JVM objects. Records can either exist in deserialized form when the RDD is cached in memory or in serialized form independent of their location [46].

In the background, the data inside RDDs is split into *partitions* that can be distributed to different worker nodes. This allows for efficient execution of programs since—in the best case—the RDD’s data is close to where it is actually used. Spark provides a declarative API working similar to functional programming or SQL, which is used for the so-called *driver* program that is running on the main node. In the driver program, the user describes operations on RDDs, which are either *transformations* or *actions*: *Transformations* are operations that transform one or multiple RDDs (called *parent RDDs*) into a new RDD (e.g. by mapping each record of a parent RDD to a new record in the output RDD), whereas *actions* “return a value to the application or export data to a storage system” [46].

Spark does not directly execute transformations, instead it lazily evaluates RDDs whenever there is an action that requires them to be calculated [44]. Each action issues a *job*, which builds an execution plan describing what operations need to be computed to get the result of the action. These execution plans form directed acyclic graphs (DAGs), the so-called *lineage graphs* (see Figure 2.1). By keeping track of the lineage of RDDs, Spark is able to reconstruct lost partitions in case of problems such as unresponsive compute nodes. If a problem occurs, Spark simply re-executes the operations that were required to compute the data stored in the lost partition(s) [44], [47].

2. Background

Because of using data partitioning, many operations can be efficiently executed in a distributed manner. In the best case, data within each partition of a transformation’s parent RDD is only used for calculating a single partition of the output RDD. This is called a *narrow transformation*. Narrow transformations are trivially parallelizable: They do not require any data to be transferred between partitions, so the partitions of the output RDD can be computed together with the partitions of the input RDDs. However, other transformations need to use a single partition of a parent RDD in multiple partitions of the output RDD, which is called a *wide transformation*. Wide transformations require *shuffling* of data, which essentially is a repartitioning that causes records to be transferred between nodes [46].

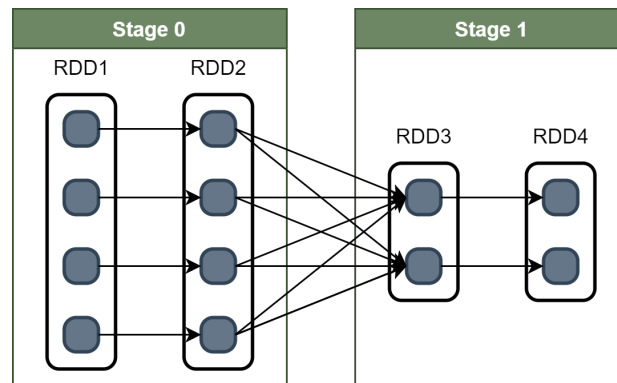


Figure 2.1.: Example lineage graph of a Spark job. The RDDs in the first stage each consist of four partitions, the RDDs in the second stage use two partitions. Arrows between partitions denote data dependencies. The transformation between RDD1 and RDD2 as well as the transformation between RDD3 and RDD4 are narrow transformations, whereas the transformation between RDD2 and RDD3 is a wide transformation that creates a stage boundary and requires shuffling.

When evaluating the lineage graph, Spark groups RDDs and narrow transformations between them into so-called *stages*, which contain “as many pipelined transformations [...] as possible” [44], effectively minimizing the required network traffic [48, *RDD > Partitions and Partitioning*]. In between stages, the data needs to be shuffled because of wide transformations, and in turn, new partitions are created. Stages that do not depend on each other can run in parallel.

The actual computation of the workload of a stage is done via serializable items of work called *tasks*, each representing a set of pipelined partitions among all RDDs in a stage (so, within a successfully completed stage, the amount of successful¹ tasks and the amount of partitions are equal). Tasks are scheduled to and executed within so-called *executors*, which are JVM processes running on the worker nodes [49, *Cluster Mode Overview*] that are able to run multiple tasks in parallel

¹ Tasks may fail or stop early due to various reasons, in which case new redundant tasks are created (see p. 34 for more details).

using multithreading [49]. Executors are unique to a Spark application, so each executor is only responsible for at most one application [49].

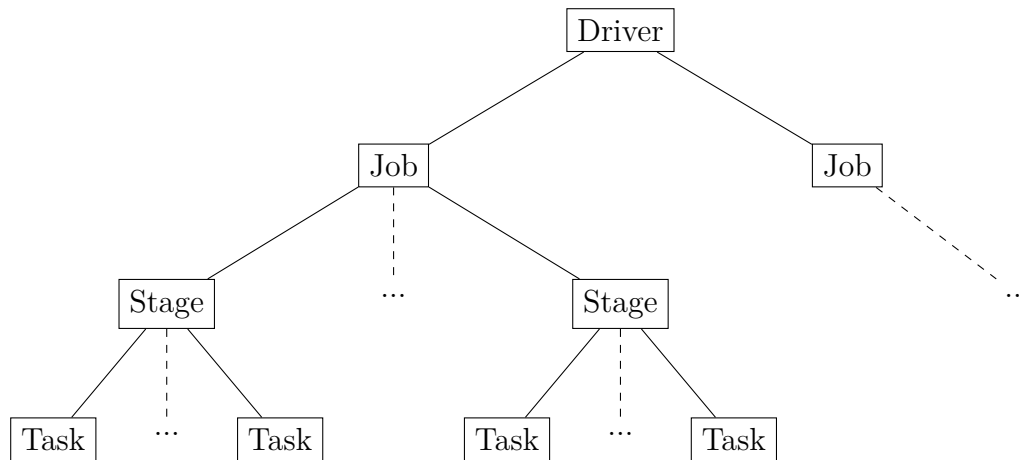


Figure 2.2.: Logical breakdown of a Spark application.

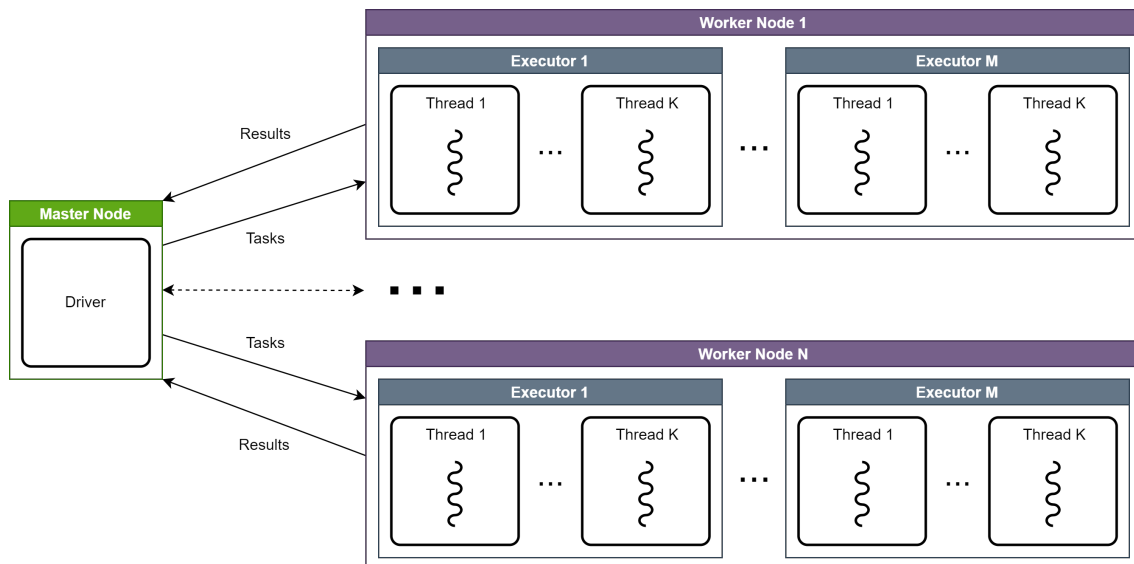


Figure 2.3.: Physical architecture of a Spark application.

2.2. Apache YARN

Apache YARN (*Yet Another Resource Negotiator*) [50] is a component of Hadoop to manage computational resources (CPUs, memory, storage, etc.) of applications on a cluster of compute nodes. Its core idea is to decouple “the programming model from the resource management infrastructure” [50], which emerged from various

2. Background

shortcomings of Hadoop when it had to serve shifting requirements and quickly growing demands [50].

A YARN cluster consists of a *Resource Manager* that is responsible for managing and scheduling a job’s resources, and multiple *Node Managers* that each control the execution of a single node and report its state to the Resource Manager [50]. Within each node of the cluster, the Resource Manager creates so-called *containers*, which are “logical bundle[s] of resources [...] bound to a particular node” [50]. An application that requires certain resources requests them from the Resource Manager, which then tries to schedule a corresponding container for the application [50].

The Spark applications that were tested as part of this thesis were configured to use YARN, which causes each Spark executor to run within its own YARN container [49, *Running Spark on YARN*] (please refer to Section 3.2 for more details).

2.3. Hadoop Distributed File System (HDFS)

The *Hadoop Distributed File System (HDFS)* [51] is a distributed file system for Apache Hadoop which partitions file data into so-called *blocks* that can be distributed among many nodes on a cluster (similar to partitions on an RDD). This allows for fast data access speeds because the data can be stored close to where it is processed, and it helps to balance the overall network and system load [52], [53]. An HDFS cluster consists of a *namenode* that manages the file system and file access, and one or multiple *datanodes* which are responsible for storing the data blocks [51]. Datanodes regularly send *heartbeats* to the namenode to communicate that they are alive and able to handle data requests [51]. For the case that a datanode fails, HDFS usually replicates each block a set amount of times (configured by the `dfs.replication` property) so that lost blocks can be recreated [51]. Using replicated blocks also has the advantage of distributing the data more uniformly so that data requests are more likely to access data close to where it is requested from [53].

Spark applications can be configured to use HDFS as the underlying file system. As explained in detail in Chapter 3, the experiments conducted as part of this thesis run workloads from the *HiBench* benchmark suite [36], which uses HDFS to store the input and output data of each workload.

2.4. POP Methodology

The *Performance Optimisation and Productivity Centre of Excellence in HPC* (short *POP*) [54] is an alliance of universities and institutions² that provides performance analysis and optimization services for software of “academic, research or commercial organizations in the European Union” [54] and conducts research on the performance of HPC applications. Any eligible organization can request performance audits,

² E.g., the HPC Group of the IT Center of the RWTH Aachen University.

resulting in a detailed report³ that addresses performance issues of the evaluated software and provides solutions to the found issues.

To assess the performance of applications in a standardized way, the POP project has developed a system of comparable performance metrics (*POP Standard Metrics*) that give a comprehensive summary of the factors that could limit the application's performance. Each metric is assigned a value that usually is defined in the range $[0, 1]$, representing an efficiency from 0% to 100% (higher values are better). The metrics are organized in a hierarchical manner (see Figure 2.4) with the *global efficiency* at the top, covering all other metrics. Sub-metrics are combined via multiplication, which propagates bad results up to the global metric. Only if all metrics at the leaves of the hierarchy tree are close to 1, the global efficiency is close to 1 as well. As long as the global efficiency shows problems, one can quickly tell what kinds of performance bottlenecks slow down the evaluated application by looking at what sub-metrics lead to the bad overall efficiency.

At the time of writing this thesis, there exist two versions of POP metrics: *POP1* [8] is a general-purpose set of metrics for arbitrary parallelism paradigms, whereas the newer *POP2* [55] is specifically targeted towards hybrid applications (in particular those using both OpenMP and MPI). For this reason, this thesis only considers POP1 and uses the terms *POP* and *POP1* interchangeably.

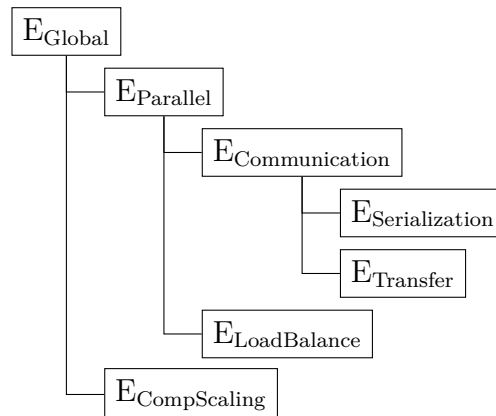


Figure 2.4.: Hierarchical overview of all POP1 standard metrics. Each metric that is not a leaf node of the the hierarchy tree is calculated by multiplying its sub-metrics together.

The POP standard metrics are defined for processes of an application, however, one can replace processes by threads or any other concurrently running entity. Let $\mathbf{P} := \{p_1, \dots, p_n\}$ for an $n \in \mathbb{N}^+$ denote the set of running processes of the evaluated application and $t_{\text{Comp}}: \mathbf{P} \rightarrow \mathbb{R}$ be a function that maps a process $p \in \mathbf{P}$ to the amount of time it spends with useful computation.

³ For a comprehensive list of completed reports see <https://co-design.pop-coe.eu/reports/> (visited on 03/23/2022).

2. Background

Serialization Efficiency. The *serialization efficiency* describes the shortest period of waiting time that each process of the application needs to wait on an ideal network without any latencies [8]. These waiting times can be caused by dependencies between processes that require a process to wait on others until the computation can continue [56] (see Figure 2.5). The serialization efficiency is defined as follows:

$$E_{\text{Serialization}} := \frac{\max_{p \in \mathcal{P}} t_{\text{Comp}}(p)}{\text{Total run time on ideal network}} \in [0, 1]. \quad (2.1)$$

If a process is actively computing for the entire duration of the program, the serialization efficiency equals 1 (100%) even if other processes may need to wait at some point in time.

In order to obtain timings of processes running in an ideal network, one can run the application in a dedicated simulator [8]. At the time of writing this thesis there exists no such simulator for Spark applications, however, results can be approximated (see p. 30 for more details).

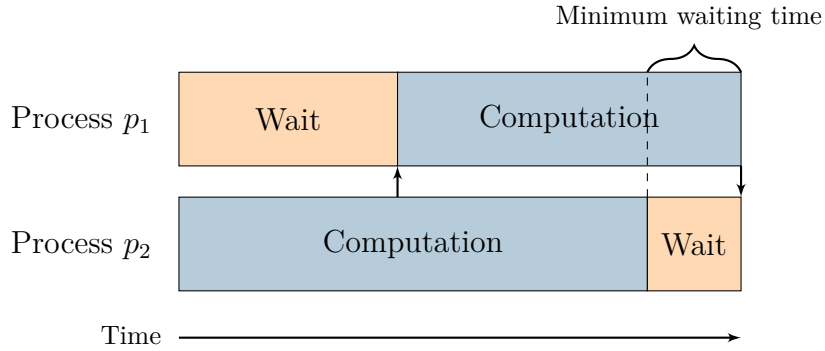


Figure 2.5.: Minimum waiting time for two processes p_1, p_2 . Arrows between processes denote the time at which a computing process resolves a dependency so that another waiting process can resume its computation. Since the serialization efficiency considers ideal networks, information about resolved dependencies is transferred instantaneously.

Transfer Efficiency. The *transfer efficiency* measures the loss in efficiency that is caused by network latencies and other communication-induced latencies. Given the run time of the application in a simulated ideal network, one can obtain the transfer efficiency E_{Transfer} as follows:

$$E_{\text{Transfer}} := \frac{\text{Total run time on ideal network}}{\text{Total run time on real network}} \in [0, 1]. \quad (2.2)$$

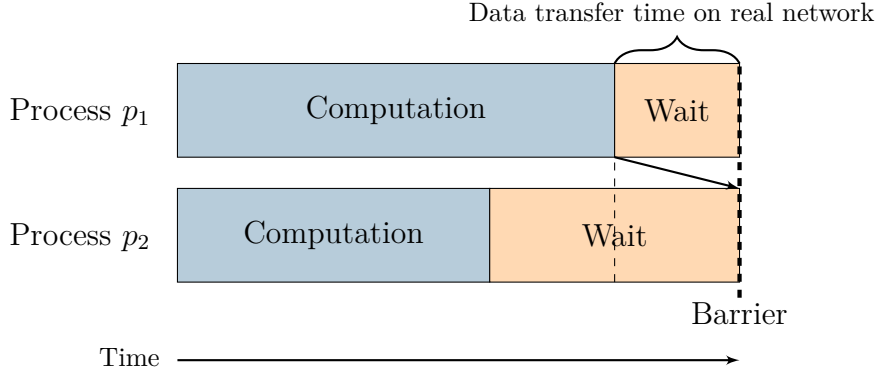


Figure 2.6.: Transfer time for two processes p_1 , p_2 using the example of a synchronization barrier. The processes are only allowed to proceed beyond the barrier if they received the information that all other involved processes did reach the barrier as well. The arrow between processes p_1 and p_2 denotes a data transfer.

Communication Efficiency. The *communication efficiency* bundles the effects of data dependencies by combining the serialization efficiency with the transfer efficiency:

$$\begin{aligned} E_{\text{Communication}} &:= E_{\text{Serialization}} \cdot E_{\text{Transfer}} \\ &= \frac{\max_{p \in \mathbf{P}} t_{\text{Comp}}(p)}{\text{Total run time on real network}} \in [0, 1]. \end{aligned} \quad (2.3)$$

Load Balance Efficiency. The *load balance efficiency* describes how well the application's workload is distributed among processes. The POP standard defines the metric as follows:

$$E_{\text{LoadBalance}} := \frac{\frac{1}{|\mathbf{P}|} \sum_{p \in \mathbf{P}} t_{\text{Comp}}(p)}{\max_{p \in \mathbf{P}} t_{\text{Comp}}(p)} \in \left[\frac{1}{|\mathbf{P}|}, 1 \right]. \quad (2.4)$$

If the workload is completely balanced, it holds that

$$\frac{1}{|\mathbf{P}|} \sum_{p \in \mathbf{P}} t_{\text{Comp}}(p) = \max_{p \in \mathbf{P}} t_{\text{Comp}}(p),$$

so the resulting efficiency equals 1. However, with this version of the formula, the value of the worst possible outcome depends on the number of processes, which for cases with very few processes might hide a bad result and prevents a comparison between applications with different amounts of processes. For those cases, it might be desirable to normalize the results to $[0, 1]$:

$$E_{\text{LoadBalanceNorm}} := \begin{cases} \frac{E_{\text{LoadBalance}}^{-\frac{1}{|\mathbf{P}|}}}{1 - \frac{1}{|\mathbf{P}|}}, & \text{if } |\mathbf{P}| > 1 \\ 1, & \text{otherwise} \end{cases} \in [0, 1]. \quad (2.5)$$

2. Background

A disadvantage of the normalized efficiency is that the meaning of its values—although they are more comparable—is not as intuitive as it is for the original efficiency. For example, with the original formula, a load balance of 0.5 means that the available processing capacity is only 50 % used, whereas the normalized formula only regards the occupancy of all threads apart from the one with the longest useful computation time. Furthermore, the values of the normalized efficiency generally still depend on the number of processes, as adding more unused processes to the application still affects the resulting values in all cases apart from the worst case.

Parallel Efficiency. The POP project describes the *parallel efficiency* metric as revealing “the inefficiency in splitting computation over processes and then communicating data between processes” [8]. It is calculated by combining the communication efficiency with the load balance efficiency:

$$E_{\text{Parallel}} := E_{\text{Communication}} \cdot E_{\text{LoadBalance}} \in [0, 1]. \quad (2.6)$$

Computational Scaling Efficiency. In practice, increasing the amount of computational resources (nodes, CPU cores, processes, etc.) that are used to run a given program with a fixed input size (i.e., strong scaling) usually creates some computational overhead. There are various reasons for this overhead, such as accessing shared resources [8] or more competition within the available computational resources (e.g., memory limitations). To calculate the strength of the scaling-induced overhead, the POP project makes use of the *computational scaling efficiency*:

$$E_{\text{CompScaling}} := \frac{\sum_{p \in \mathbf{P}_R} t_{\text{Comp}}(p) \text{ for the reference case}}{\sum_{p \in \mathbf{P}} t_{\text{Comp}}(p) \text{ for the current configuration}} \in [0, \infty). \quad (2.7)$$

Compared to the other POP standard metrics, computational scaling does not only take a single run of the application into account. Instead, it compares the maximum useful computation time of all processes to a given *reference* case (using processes \mathbf{P}_R) that only utilizes a basis set of computational resources (e.g., one CPU only). It also is the only *atomic* metric (i.e., a metric without sub-metrics) of the POP standard metrics that cannot be represented in a $[0, 1]$ range since application runtimes can be both faster and slower than the specified reference case.

Global Efficiency. The *global efficiency* covers all other efficiencies and is the first efficiency to consider when looking at calculated POP metrics. Due to the multiplicative nature of efficiencies, the global efficiency is close to 0 when *any* other metric is close to 0, and close to 1 if *all* other metrics are close to 1.

$$E_{\text{Global}} := E_{\text{Parallel}} \cdot E_{\text{CompScaling}} \in [0, \infty) \quad (2.8)$$

3. Experimental Setup

To explore and evaluate the application of POP metrics to Spark application, I conducted various experiments on *CLAIX-2018*, which is the compute cluster of the RWTH Aachen University. The experiments run selected Spark benchmarks from the *HiBench* benchmark suite [36], which bundles many workloads commonly used in the big data world. I deployed the experiments with *Apptainer*¹ [57], which is a software to run scientific applications in a containerized and thus reproducible environment [58] that simplified the installation of Hadoop and Spark on the cluster.

Each experiment is performed as follows: After the cluster grants access to a specified amount of compute resources, an Apptainer container is started on each allocated node, differentiating between main and worker nodes by running different SCIF (*Scientific Filesystem* [59]) applications. These applications first call a Python script to create all necessary configuration files for Hadoop and Spark based on the experiment configuration stored in environment variables, and then they start the Hadoop cluster daemons (HDFS, YARN, and the Spark History Server, which is a service to inspect metrics of completed Spark applications). After that, the main node starts the HiBench benchmark and downloads the Spark event logs from the History Server (see Section 4.1) as well as various diagnostic files before the session is terminated.

Originally, it was planned to run each experiment three times in order to mitigate noise by averaging the results afterwards, however, the quota of the thesis project on CLAIX-2018 (24,000 core-hours in total, 3671 core-hours per month) sadly did not allow for that. The experiments requested resources and computation time from the cluster with the *Slurm* workload manager [60], and each experiment was executed using Slurm’s `--exclusive` switch in order to isolate it from other jobs on the cluster that might interfere with the running application. This switch causes Slurm to always allocate 48 CPUs per node on CLAIX-2018 even if not all cores are used, which in turn easily costs a few hundred core-hours per run on average. For that reason, the measurements presented in this thesis need to be taken with some care since there is no compensation for naturally arising fluctuations.

3.1. Workloads

With the goal of covering a wide range of real-world application types, I selected three different HiBench workloads for my experiments, which are briefly explained in this section.

¹ Formerly called *Singularity*.

3. Experimental Setup

3.1.1. WordCount

For each distinct word in a given text, the *WordCount* algorithm outputs the number of occurrences of that specific word within the text. Although it is often used as a “hello world” application due to its simplicity, the algorithm is of great interest for benchmarking since it corresponds with many real-world workloads [36], [61]. Spark divides HiBench’s WordCount implementation² into two stages (see Figure 3.1):

1. Read the input file into an RDD by using `sequenceToFile` and convert the input records to string with a call to `map`. Then, use `flatMap` to split each record into individual words and associate a count of 1 with each word using the `map` operation that maps each word to a key value pair $(word, 1)$.
2. Reduce the pairs of words with their associated counts by adding the counts together (using `reduceByKey`). Afterwards, save the resulting $(word, count)$ pairs to a file with the help of `map` to convert each pair to a string.

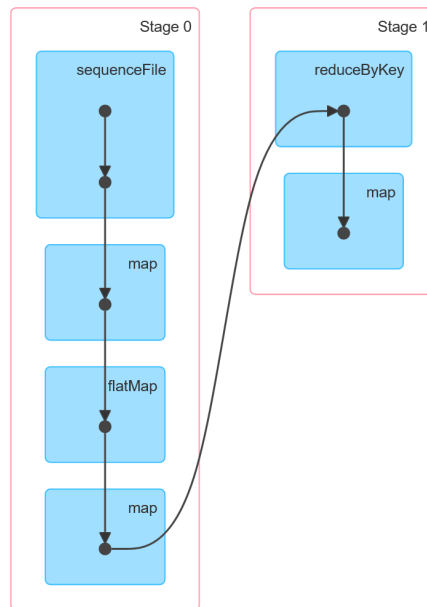


Figure 3.1.: The DAG (directed acyclic graph) constructed for the WordCount benchmark as shown in the Spark History Server web UI.

² See <https://github.com/Intel-bigdata/HiBench/blob/bf390d2e60ed05f77264820b79f4637910e9629f/sparkbench/micro/src/main/scala/com/intel/sparkbench/micro/ScalaWordCount.scala> (visited on 03/19/2013).

3.1.2. PageRank

The *PageRank* algorithm is a famous iterative algorithm to rank nodes in a graph based on the amount and rank of other nodes that point to them. Although its core idea was used in variations for different purposes earlier (cf. [62]), the algorithm was made famous by Page *et al.* for their work on ranking websites on the Google search engine [63]. Their work ultimately gave this algorithm its name.

Due to being an iterative algorithm, PageRank shows the effects of caching and thus is of great use for benchmarking [25]. Furthermore, it is “representative of one of the most significant uses of MapReduce - large-scale search indexing systems” [61]. To ensure comparability with real-world data, the generated HiBench input data that represents a graph network follows the natural *Zipf distribution* [61].

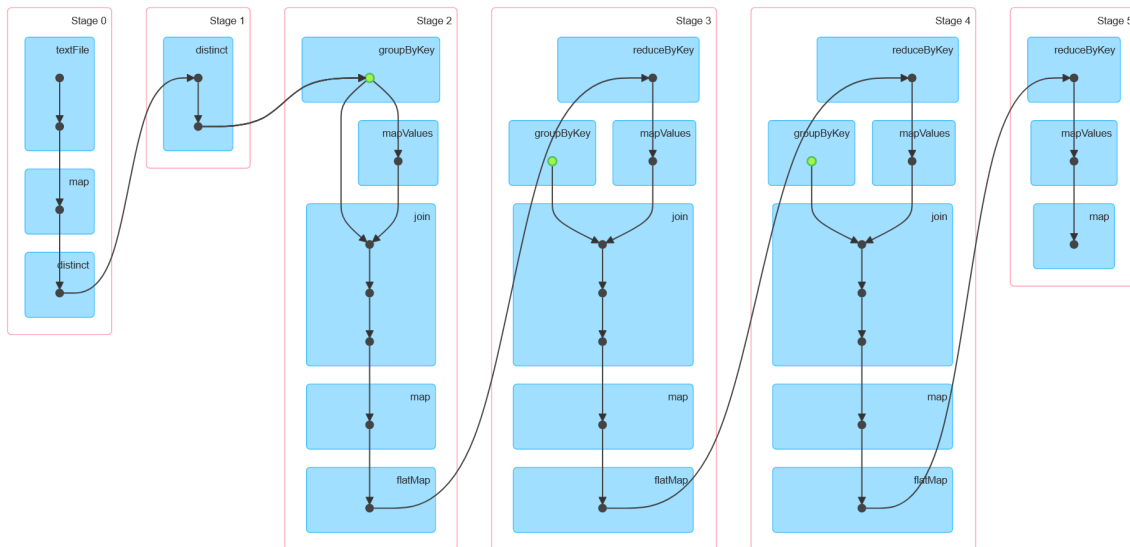


Figure 3.2.: The DAG (directed acyclic graph) constructed for the PageRank benchmark as shown in the Spark History Server web UI.

3.1.3. K-means Clustering

K-means clustering is a CPU-intensive [3] unsupervised machine learning and data mining algorithm that groups data samples of arbitrary dimensions into k -many *clusters* such that each data point in the input data set is closer to the center point of its cluster than to the center points of all other clusters. The algorithm works in an iterative manner; each iteration, the clusters’ center points are refined until the result is a close enough approximation of an exact result³, or until a certain amount of iterations is reached. In case of HiBench, the application always computes a fixed

³ The result varies depending on the initial selection of cluster center points. A solution of k -means always converges to a local optimum (i.e., given the cluster centers, each point is correctly assigned), but not necessarily to the global one (i.e., the clusters are optimally distributed to minimize the variance within a cluster) [64].

3. Experimental Setup

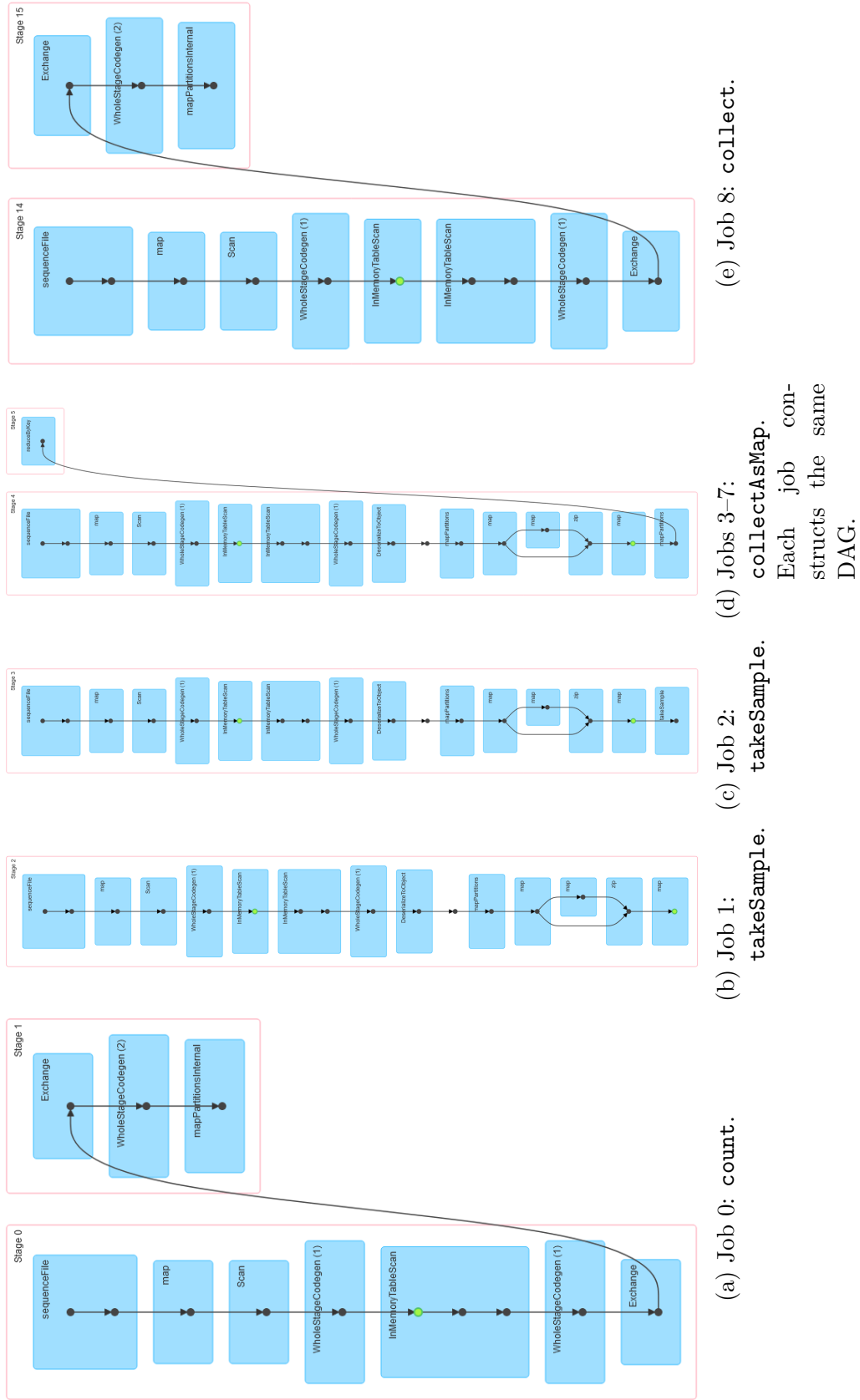


Figure 3.3.: The DAGs (directed acyclic graphs) constructed for the k-means benchmark as shown in the Spark History Server web UI.

amount of iterations⁴. Similar to PageRank, k-means clustering can be used to see the effects of data caching due to its iterative nature [25].

3.2. Configuration

The experiments were carried out with different configurations in order to provoke various performance bottlenecks of the benchmark applications. Due to the sheer amount of available configuration options and their possible combinations, only a few important variables were varied throughout the experiments (see Table 3.1). For a complete list of all used variable combinations, please refer to Appendix B. References to individual experiments are denoted by [E<num>] where <num> is a unique index for each experiment as listed in Appendix B (e.g., [E1]).

Variable	Value(s)
Number of worker nodes	4, 8, 16
Total number of executors	4, 8, 16
Cores per executor	4, 8, 16
Degree of parallelism	4, 8, 16
Spark memory fraction	0.1, 0.6, 0.9
HiBench input size	huge, gigantic
Spark driver memory	4 GB
Spark executor memory	4 GB
Speculative task execution (<code>spark.speculation</code>)	false
Job scheduler	FIFO
HDFS replication factor (<code>dfs.replication</code>)	1
HDFS block size (<code>dfs.blocksize</code>)	256 MiB

Table 3.1.: List of configuration variables used for the experiments. The number of nodes is configured by providing a list of worker hostnames in Hadoop’s `worker` file [65, Chapter *Hadoop Cluster Setup*], all other parameters are specified in various configuration files of HDFS, Spark, and HiBench.

One needs to be aware that Spark may decide to use a different configuration than the one set up by the user, meaning that one has to verify that the application in question is actually using the desired configuration. This can happen for example

⁴ See <https://github.com/Intel-bigdata/HiBench/blob/bf390d2e60ed05f77264820b79f4637910e9629f/sparkbench/ml/src/main/scala/com/intel/sparkbench/ml/DenseKMeans.scala#L113> (visited on 03/19/2023).

3. Experimental Setup

if the allocated hardware resources are not sufficient for the given configuration. In the conducted experiments, Spark sometimes decided to execute the application with a different amount of executors than configured, which is annotated above the affected experiments in the appendix.

Cores per executor. Each executor in Spark may utilize one or multiple cores, whose amount controls how many threads each executor can use to run tasks in parallel [66]. In HiBench applications, the amount of cores per executor is specified by the `hibench.yarn.executor.cores` property that directly maps to Spark’s `--executor-cores` parameter.

If the cluster is configured to use Hadoop YARN (as it is the case for the conducted experiments), each executor is a YARN *container*. In that case, the amount of cores does not necessarily refer to physical cores, but to YARN’s so-called *virtual cores* (*vcores*) that are configured with the `yarn.nodemanager.resource.cpu-vcores` property [66]. Virtual cores are abstract cores, their amount specifies the number of threads that a Spark application can use to run tasks [66].

The number of cores per executor (and thus the number of threads per executor) has a direct influence on the performance of Spark applications since more threads allow to run more tasks in parallel. However, too many threads can also be detrimental for performance: For example, Ryza comes to the conclusion that HDFS struggles with too many concurrent reads. In their experiments, five cores per executor appeared to be a good rule of thumb to achieve the best write throughput [66]. Similar problems with HDFS have been described in [67].

Degree of parallelism. The degree of parallelism specifies how many partitions Spark should create for distributing data. For the experiments, this is configured via the HiBench properties `hibench.default.map.parallelism` and `hibench.default.shuffle.parallelism`, which for Spark applications map to `spark.default.parallelism` and `spark.sql.shuffle.partitions`, respectively. The degree of map parallelism describes the partitioning of RDD data, whereas the degree of shuffle parallelism describes the amount of partitions during shuffle operations [49, *Spark Configuration: Execution Behaviour & Spark SQL*]. One needs to be aware that Spark only treats the configured level of parallelism as a *hint* (cf. [48, Internals: TaskSchedulerImpl > Default Level of Parallelism]), meaning that in some stages, Spark might decide to use the number of partitions of a parent RDD for a transformation, or the total amount of executor cores (cf. [49, *Spark Configuration: Execution Behaviour & Tuning Spark: Level of Parallelism*]).

The *effective* amount of partitioning of RDD data determines the amount of tasks, since for each partition, a unique task is created. In other words, the amount of parallelism “is equal to the number of tasks for each operation included in that stage” [3]. One needs to pay special attention to “operation included” in the previous quote: Since RDDs can persist in memory and get accessed by multiple stages, a stage might not need to compute the tasks for an already computed RDD again.

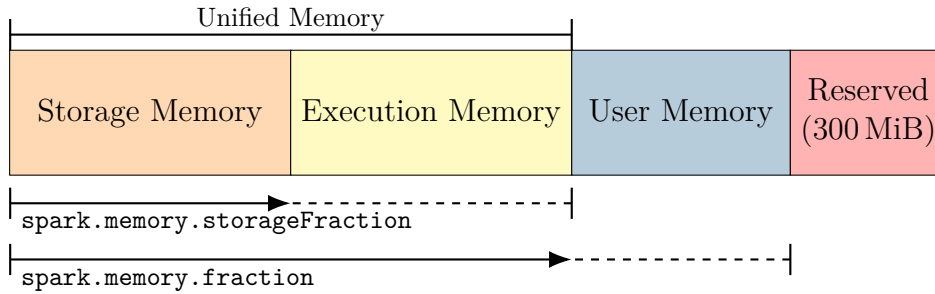


Figure 3.4.: Conceptual breakdown of the JVM heap space. Both the relative size and location of the displayed memory regions are exemplary and might differ in practice, and it is not documented whether the memory regions are contiguous.

Achieving the right level of parallelism is significant for the performance of a Spark application: If the degree of parallelism is less than the amount of total available cores, the application cannot fully utilize all of them. On the other hand, a too large degree of parallelism can lead to many small tasks, which eventually might lead to performance issues due to a high overhead of managing those tasks (cf. [68], [69]).

Memory fraction. Spark divides the JVM heap space on each executor into multiple memory regions with different purposes. Some amount of memory (hard-coded to 300 MiB⁵) is reserved to the application outside of any Spark execution purpose to ensure “sufficient memory for the system even for small heaps”⁶. In Spark version 3.0.0, the application terminates with an exception if the available heap size is less than 1.5 times the reserved memory size⁶, making sure that the user assigns at least a bare minimum of memory to each executor. Together with the so-called *user memory* region, the reserved memory is used “for user data structures, internal metadata in Spark, and safeguarding against OOM [*out of memory; M.B.*] errors in the case of sparse and unusually large records” [49, *Tuning Spark: Memory Management Overview*]. In other words, the user memory region together with the reserved memory is regular working memory for arbitrary application purposes.

The rest of the memory is the so-called *unified memory* region, whose size is configured by the `spark.memory.fraction` property. This property specifies the fraction (as a factor in $[0, 1]$) of the non-reserved memory that is allocated as unified memory (instead of being used as user memory; see Figure 3.4). By varying this setting, it is possible to provoke different memory pressure effects such as cache eviction, which is why it is varied throughout the conducted experiments.

⁵ The official Spark documentation does not make it clear whether the reserved size is 300 MiB or 300 MB (cf. [49, *Spark Configuration: Memory Management*] and [49, *Tuning Spark: Memory Management Overview*], but a look in the source code (see footnote 6) reveals the actual size.

⁶ Related source code: <https://github.com/apache/spark/blob/v3.0.0/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala#L194-L235> (visited on 03/18/2023).

3. Experimental Setup

The unified memory region is split into two sub-regions, *storage memory* and *execution memory*. Execution memory is the memory used in most RDD computations (“shuffles, joins, sorts and aggregations” [49, *ibid.*]), whereas storage memory refers to memory “used for caching and propagating internal data across the cluster” [49, *ibid.*]. The size of the storage region within the unified memory is configured by the `spark.memory.storageFraction` property (also a factor in $[0, 1]$), which for the conducted experiments is kept at the default value of 0.5 (cf. [49, *Spark Configuration: Memory Management*]).

Petridis, Gounaris, and Torres have demonstrated that the memory fraction has a high impact on performance and causes application crashes in extreme cases [4]. For instance, increasing the memory fraction might result in too little memory available for shuffling operations [4].

Input size. Before running a benchmark, HiBench generates pseudo-random input data based on user-provided size configurations called *scale profiles*. Each profile (namely `tiny`, `small`, `large`, `huge`, `gigantic`, and `bigdata`) is linked to preconfigured (but modifiable) workload-specific settings such as the number of pages for PageRank or the number of samples for k-means for example. For the conducted experiments, the scale profiles were used as-is without additional modifications.

Due to the differences in data generations for different workloads, it is not meaningful to compare distinct workloads by their scale profile (the input size might differ greatly even for the same scale profile). Thus, for comparing input data sizes, it is more appropriate to use the `Bytes Read` stage metric contained in the Spark event logs (see Table 3.2). For the conducted experiments, the value of this metric for the first stage directly corresponds to the size of the input data.

Although the input size is fairly small for the PageRank workload using `huge` as the scale profile, it was not possible to test the experiments with `gigantic` as the applications crashed, which is why this workload is only evaluated for the `huge` scale profile. `bigdata` caused crashes for all tested workloads.

Workload	<code>hibench.scale.profile</code>	<code>Bytes Read</code> (stage 0)
WordCount	<code>huge</code>	30.35 GiB
	<code>gigantic</code>	303.5 GiB
PageRank	<code>huge</code>	2.79 GiB
k-means	<code>huge</code>	18.5 GiB
	<code>gigantic</code>	37.1 GiB

Table 3.2.: `hibench.scale.profile` and the corresponding `Bytes Read` metric for the first stage (stage 0) of the tested workloads when using the default HiBench workload configurations. Note that the `Bytes Read` values vary slightly depending on the generated input data.

3.3. Hardware

The experiments that were conducted as part of this thesis (see Appendix B) were carried out on the RWTH Aachen University’s compute cluster *CLAIX-2018*. The compute nodes of the cluster are equipped with Intel® Skylake Platinum 8160 CPUs running at 2.1 GHz [70]. Each node consists of 2 sockets with 24 cores per socket and the total memory per node amounts to 192 GB [70]. The cluster uses a *Lustre*-based [71] file system with a read/write bandwidth of 150 GB/s [72].

3.4. Software

The experiments are conducted with HiBench commit `bf390d2e60` (link visited on 02/14/2023), which was configured to target Apache Spark 3.0.0 for Hadoop 3.2.3. *CLAIX-2018* runs Linux CentOS 7.9 and the Apptainer image used to run the experiments is based on the `debian:stretch` (Debian 9 [73]) Docker image. The OpenJDK version used in the Apptainer container is 1.8.0_332.

4. Implementing the POP Methodology

In this chapter, I describe the steps I took to analyze an already executed Spark application by means of extracting run-time data from the log files (see Section 4.1), preparing the data for analysis (Section 4.2) and calculating POP metrics for Spark applications (Section 4.3).

4.1. Data Extraction

Whenever a certain *event* happens in a running Spark application (that is, a new executor is added to the application, a new stage is submitted, or a new task is started, for example), Spark writes it to a log file if `spark.eventLog.enabled` is set to `true`. In the log file, each event is logged as a single line which is a valid JSON object literal that does not only contain the event type, but also further information about the event. For instance, each `SparkListenerTaskEnd` event (emitted when a task terminates) contains information about the task’s stage and executor as well as various task metrics such as runtimes and the amount of bytes the task has read. From the information contained in the event logs, one can gather comprehensive information about all stages, executors, and tasks of the application. Obtaining information about individual threads, however, is more involved and explained in Section 4.2.

In this thesis, I use the Spark event logs as the sole source of information regarding the application’s performance. Although there exist other possible approaches to collecting performance information (e.g., using the JDK’s Java Flight Recorder [26] or dedicated tools for monitoring distributed systems such as *Ganglia* [24], [25]), the event logs contain all the information required to calculate the POP metrics as proposed in Section 4.3.

I define \mathbf{S} to denote the set of all stages within the application, \mathbf{T} as the set of all threads within the application, and \mathbf{K} as the set of all tasks within the application. Furthermore, I denote the sets of all used threads resp. tasks within a stage $s \in \mathbf{S}$ as $\mathbf{T}_s \subseteq \mathbf{T}$ resp. $\mathbf{K}_s \subseteq \mathbf{K}$ and the set of all tasks within a thread of a stage $t \in \mathbf{T}_s$ as $\mathbf{K}_t \subseteq \mathbf{K}$. Metrics that belong to individual stages, threads, and tasks are denoted as attributes $x.\text{attr}$, which translates to “the value of the attribute `attr` that belongs to x ” (for an $x \in \mathbf{K} \cup \mathbf{T} \cup \mathbf{S}$). Some metrics directly come from the event logs, others are later computed from that information; see Tables 4.1, 4.2, and 4.3 for a complete list of attributes used in this thesis.

4. Implementing the POP Methodology

Sometimes, tasks might fail due to various reasons like running out-of-memory or executor failures. In that case, event logs of those tasks might not contain the same amount of information that they normally would for successful tasks even if those tasks used computational resources and thus affected the performance like any other task. This forces me to set some attributes of failed tasks to default values, since there is no way to retrieve the actual information. Thus, if an application has a large amount of failed tasks, the obtained POP results might vary from the correct metrics.

Accumulators. Some Spark metrics originate from *accumulators*, which is a special type of shared variable into which tasks and executors can merge values by means of a commutative and associative operation [49, *RDD Programming Guide: Accumulators*] (e.g., addition, which is used for all accumulators in Table 4.1). For each accumulator variable, each task owns a local instance of that accumulator, and after a task is completed, its local value is sent to the driver where it is merged to a global reference to the accumulator [48, *Shared Variables > Accumulators and Internals: Scheduler > DAGScheduler > Updating Accumulators of Completed Tasks*]. In Spark’s event logs, accumulators are listed with two value attributes per task, `update` and `value`. The `update` attribute describes the value that the task sent to the driver, whereas `value` is the total value of the global accumulator after being merged with the `update` value.

Accumulators are not completely reliable when they are updated within transformations: They might get updated multiple times by a single task if stages need to be recalculated due to stage failures, or if Spark speculatively executes tasks to counteract slowly running tasks [74]. For this reason, results from accumulators should be treated with some care.

4.2. Thread Scheduling

The formulae of the POP1 metrics are defined with processes in mind (a generic approach working for many parallelism paradigms such as MPI; cf. [8]), which for Spark applications would be equivalent to the JVM processes run by their executors. However, Spark allows for a more fine-grained view on the parallel distribution of work: Each executor can run multiple threads in parallel, which is configured with the `hibench.yarn.executor.cores` property as explained in Section 3.2. Spark provides information about which executor ran which task, but unfortunately, the same kind of information is not available for threads. To obtain that information, I

¹ See <https://github.com/apache/spark/blob/v3.0.0/core/src/main/scala/org/apache/spark/ui/jobs/StagePage.scala#L263-L296> and <https://github.com/apache/spark/blob/v3.0.0/core/src/main/scala/org/apache/spark/status/AppStatusUtils.scala#L30-L54> (both visited on 03/10/2023).

Task attribute	Description
$k.\text{launchTime}$	The timestamp in ms at which the task is launched. Equivalent Spark task metric: Launch Time
$k.\text{finishTime}$	The timestamp in ms at which the task result reaches the driver. Equivalent Spark task metric: Finish Time
$k.\text{totalDuration}$	$k.\text{finishTime} - k.\text{launchTime}$
$k.\text{execRunTime}$	The time in ms spent by the executor running the task. The equivalent Spark metric includes both $k.\text{shuffleFetchTime}$ and—although not documented— $k.\text{shuffleWriteTime}$ ¹ . Equivalent Spark task metric: * <code>internal.metrics.executorRunTime</code>
$k.\text{usefulCompTime}$	See Equation 4.3.
$k.\text{schedulerDelay}$	See Section 4.2.
$k.\text{getResultTime}$	The time in ms “that the driver spends fetching task results from workers” [49, <i>Web UI</i>]. Equivalent Spark task metric: Getting Result Time
$k.\text{execDeserializeTime}$	The time in ms spent by deserializing the task after it has reached the executor. Equivalent Spark task metric: * <code>internal.metrics.executorDeserializeTime</code>
$k.\text{resSerializeTime}$	The time in ms taken on the executor to serialize the result of a task. Equivalent Spark task metric: Result Serialization Time
$k.\text{shuffleFetchTime}$	The time in ms the task waits for remote shuffle data to arrive. Equivalent Spark task metric: * <code>internal.metrics.shuffle.read.fetchWaitTime</code>
$k.\text{shuffleWriteTime}$	The time in ms taken by the task writing data to be shuffled. Equivalent Spark task metric: * <code>internal.metrics.shuffle.write.writeTime</code>
$k.\text{shuffleBytesRead}$	The amount of bytes that the task read from shuffle data. Calculated from the Spark task metrics * <code>internal.metrics.shuffle.read.localBytesRead</code> + * <code>internal.metrics.shuffle.read.remoteBytesRead</code>
$k.\text{storageBytesRead}$	The amount of bytes read from storage. Equivalent Spark task metric: * <code>internal.metrics.input.bytesRead</code>
$k.\text{taskLocality}$	Relative location of the task’s data. See p. 38 and Table 4.4. Equivalent Spark task metric: Locality
$k.\text{jvmGCTime}$	The time in ms spent by garbage collection during the task’s runtime. Equivalent Spark task metric: * <code>internal.metrics.jvmGCTime</code>
$k.\text{taskFailed}$	Whether the task failed due to an exception (see p. 34). Equivalent Spark task metric: Failed
$k.\text{taskKilled}$	Whether the task was killed by the application (see p. 34). Equivalent Spark task metric: Killed
$k.\text{bytesSpilled}$	The amount of memory spilled to disk in bytes (see p. 40). Equivalent Spark task metric: Memory Bytes Spilled
$k.\text{peakMemory}$	The maximum execution memory this task used (see p. 40). Equivalent Spark task metric: Peak Execution Memory

Table 4.1.: Task attributes used in this thesis (for all tasks $k \in \mathbf{K}$). Spark metrics denoted with “*” are accumulators, whose `update` values are used for the task attribute. For a complete list of available Spark metrics please refer to [49, *Monitoring and Instrumentation*].

4. Implementing the POP Methodology

Thread attribute	Description
$t.\text{usefulCompTime}$	The total useful computation time of the thread in ms, as defined in Equation 4.4.
$t.\text{inputBytesRead}$	The total amount of input bytes all tasks of the thread read from storage and shuffle data: $\sum_{k \in \mathcal{K}_t} k.\text{storageBytesRead} + k.\text{shuffleBytesRead}$.

Table 4.2.: Thread attributes used in this thesis (for all threads $t \in \mathbf{T}$).

Stage attribute	Description
$s.\text{duration}$	The total run time of a stage in ms. Calculated with the two Spark metrics <code>completionTime</code> – <code>submissionTime</code> .

Table 4.3.: Stage attributes used in this thesis (for all stages $s \in \mathbf{S}$).

simulate a thread scheduler within each executor.

The scheduler, whose pseudocode implementation can be found in simplified form in Algorithm 1, first assigns as many threads to each used executor as there are cores configured, and then allocates individual tasks to threads based on the time they arrive at the scheduler and their executor ID (equivalent Spark task metric: `Executor ID`).

The arrival time of a task is the task’s launch time $k.\text{launchTime}$ plus the so-called *scheduler delay*. The description of what *exactly* is encompassed within the scheduler delay varies slightly even within official Spark sources. According to a tooltip in the web UI of the Spark History Server, the “Scheduler delay includes time to ship the task from the scheduler to the executor, and time to send the task result from the executor to the scheduler”². On the other hand, the Spark documentation describes it only as “the time the task waits to be scheduled for execution” [49, *Web UI*]. The documentation for the *IBM Spectrum Conductor* platform, which can be used to manage Spark applications, seems to support the latter description³, and the existence of the `Getting Result Time` task metric ($k.\text{getResultTime}$) also speaks against the tooltip in the web UI. Because of this—and because it would not be possible to split the scheduler delay into a pre-task delay and a post-task delay from the information available in the event logs—I will stick to the explanation of the Spark and Spectrum Conductor documentations and assume that the History Server UI tooltip is either wrong or imprecise. Following this, the scheduler delay is the time between launching a task and its arrival in serialized form at an executor. The

² See <https://github.com/apache/spark/blob/v3.3.0/core/src/main/resources/org/apache/spark/ui/static/stagepage.js#L351-L352> (visited on 02/10/2023).

³ See https://www.ibm.com/docs/en/spectrum-conductor/2.5.1?topic=performance-tuning-spark-application-tasks#concept_yqq_r1q_cy__title__2 (visited on 02/10/2023).

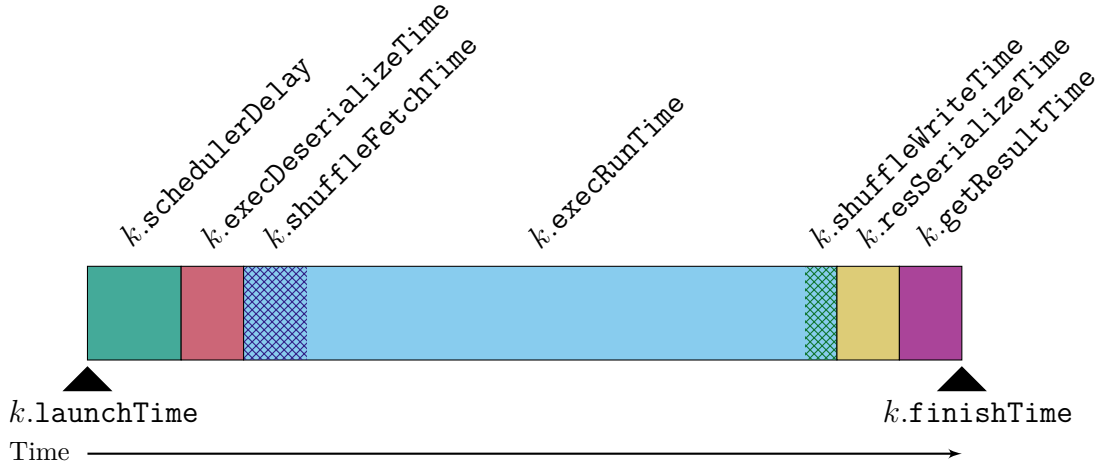


Figure 4.1.: Anatomy of a task $k \in \mathbf{K}$. The ratios of individual timings are exemplary. Note that $k.\text{shuffleFetchTime}$ and $k.\text{shuffleWriteTime}$ overlap with $k.\text{execRunTime}$ (see Table 4.1).

scheduler delay is not directly contained in the task metric in the event log, but since all other timings of a task are available, it can be calculated as follows for a task $k \in \mathbf{K}$:

$$\begin{aligned}
 k.\text{schedulerDelay} &:= k.\text{finishTime} - k.\text{launchTime} \\
 &\quad - k.\text{execDeserializeTime} - k.\text{execRunTime} \\
 &\quad - k.\text{resSerializeTime} - k.\text{getResultTime}.
 \end{aligned} \tag{4.1}$$

When there is no free thread for a task at the time of its arrival, the scheduler simulates additional waiting and outputs a warning so that the reason for the delay can be investigated. Since the arrival times of tasks are fixed by data contained in the event logs, having to wait implies either a problem with the scheduler or with the data itself. Although the issue did not arise in most of the conducted experiments, in some cases the scheduler had to simulate waiting by 2 milliseconds due to overlapping task timings ([E20], [E26], [E30])⁴. Yet, I am rather confident in the correctness of the implemented scheduler *as far as it is possible to be correct*: Because the timings from the event logs that I use for scheduling are provided with millisecond resolution, there is a chance for imprecisions and rounding errors that can slightly affect the results or even cause waiting time. Although being difficult to prove, I suspect this to be the cause of the observed delays as described above.

⁴ Originally, I suspected that not subtracting $k.\text{getResultTime}$ from $k.\text{finishTime}$ in the simulated scheduler (see Algorithm 1, line 28) was causing this delay, but implementing this did not solve the issue: In the referenced experiments, not a single task had a $k.\text{getResultTime}$ value larger than 0. It is not documented when exactly a thread is ready for further work after completing a task; in fact it could even be wrong to subtract $k.\text{getResultTime}$ as the driver might need to re-fetch the task result before a new task can start (in case fetching the task result failed before). Further research is required on this subject.

4. Implementing the POP Methodology

However, data precision is not the only limitation of the simulated scheduler. In general, we can only reliably map a task to an executor since that mapping is provided by the event logs, but there is no guarantee to which thread within an executor a task is scheduled. This depends on the order in which the algorithm iterates through the threads to check whether there are threads available for work. The results of a few POP metrics might vary marginally depending on to which thread a task is scheduled, but the scheduler will not introduce further delay since previous work of a free thread does not affect future execution. Also, the scheduler technically depends on partly undocumented and unspecified Spark internals that might change at any point in time.

A much bigger issue is that in some rare cases, Spark might decide to run stages with less executors as being available and instead allocates more cores to the remaining executors ([E22], [E28]). In most cases this is not problematic: The scheduler first only assigns threads to executors that are actually used by tasks in the current stage, and in turn it leaves some unassigned threads that can be claimed by overlapping tasks for their executors. However, if this case occurs in combination with the aforementioned precision issues that can cause some tasks to slightly overlap, the scheduler might allocate more threads per executor than actually used⁵ and thus cause waiting times for threads of other executors since they can no longer use as many threads as they did in reality. There is no information available in the event logs to systematically notice these issues, they usually make themselves felt by causing long (multi-second range) task waiting times because not all tasks can be scheduled on the thread to which they belong. This problem could be (unreliably) resolved either by introducing a clever heuristic that differentiates task overlapping from situations in which executors use a different amount of cores, or manual user intervention in problematic cases.

From the result of the thread scheduler, one can obtain the amount of threads that are actually used within a stage $s \in S$ (\mathbf{T}_s). Since Spark might spawn less parallel tasks than there are threads (e.g., due to a too small degree of parallelism), \mathbf{T}_s differs from the set of available threads \mathbf{T} in many cases. Many of the POP metrics defined in Section 4.3.2 calculate averages over a set of threads and specifically consider \mathbf{T}_s instead of \mathbf{T} in order to decorrelate the results from the stages' load balance, which would otherwise affect those metrics in case of unused threads.

⁵ This effect can be seen in the web UI of the Spark History Server, which in the *Event Timeline* on the *Stage Detail* pages simply stacks all tasks as they fit vertically and thus often requires more vertical lanes for tasks than there are threads. This is a side effect of 1) ignoring the scheduler delay for the sake of illustration purposes (which is justified but increases the chance of overlapping tasks) and 2) creating the timeline graph with the *vis.js* framework [75], which automatically calculates the stacking based on the given timings and the current zoom level (tasks outside the viewport are not taken into account). This is also why it is not possible to label the y-axis within each executor section on the timeline.

4.3. Calculating POP Metrics for Spark Applications

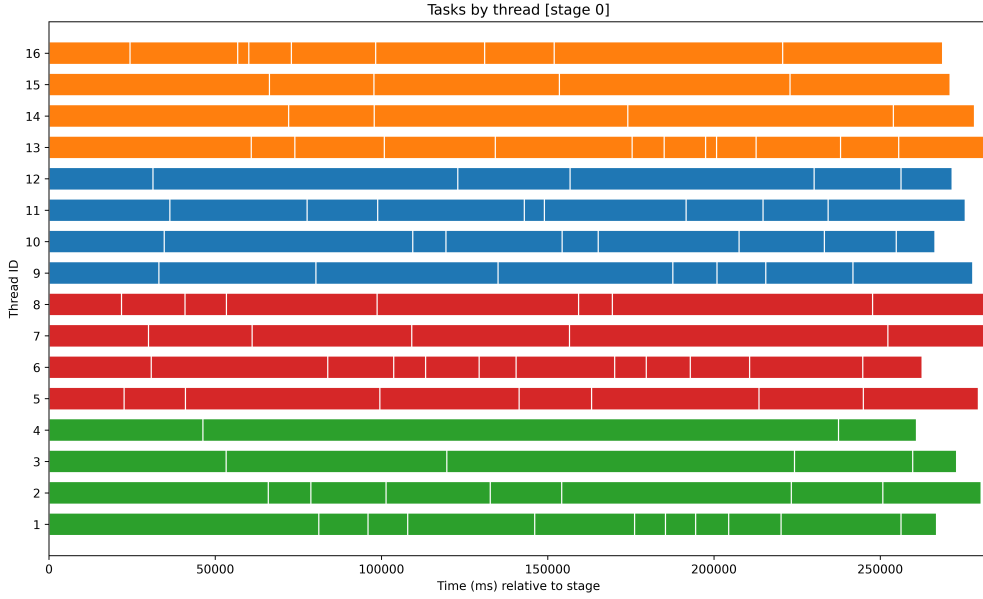


Figure 4.2.: Example result of the simulated thread scheduler for the first stage of the WordCount benchmark (experiment [E2]). Each colored block represents a single task, with each distinct color representing a unique executor.

4.3. Calculating POP Metrics for Spark Applications

Since Spark applications are naturally divided into stages that 1) do not interact with each other while running and 2) may differ greatly in terms of their purpose for the application, I calculate the POP metrics per stage. Generally, multiple stages or jobs can run in parallel if they do not depend on each other. However, in case of the experiments that were conducted as part of this thesis, the stages and jobs did depend on each other, which in turn caused all stages to run sequentially. This is convenient for the calculation of POP metrics since stages do not compete for computational resources, and for the formulae presented in this chapter I thus assume that all stages run in isolation from each other. For applications that run multiple jobs or stages in parallel, it might be required to approach the calculation of some metrics in a different way.

Since different stages of an application often differ greatly in terms of their duration, it does not make sense to treat the resulting metrics of each stage equally important. A stage that runs for less than a second usually is not worth optimizing, whereas the performance of a stage that runs for many minutes, hours, or even days has to be taken much more serious. To prevent misinterpretations of POP results for Spark applications, I also provide each efficiency as an average weighted by the duration of each stage. This way, stages with longer runtimes contribute much stronger to the final result. The weighted average E^\varnothing of an arbitrary POP

4. Implementing the POP Methodology

efficiency E is computed as

$$E^\emptyset = \frac{1}{\sum_{s \in \mathcal{S}} s.\text{duration}} \sum_{s \in \mathcal{S}} (s.\text{duration} \cdot E^s), \quad (4.2)$$

with E^s denoting the value of E with regard to stage $s \in \mathcal{S}$.

Most of the formulae presented in this section contain ratios of measurements which potentially could cause a division by zero. For many POP metrics these cases luckily almost never happen in reality, as the run time of tasks or threads—they often make up the denominator of said ratios—is usually non-zero. However, some formulae like Equation 4.19 are prone to this issue, and even if the risk is low in other cases, it makes sense to define default values for the formulae just in case. Since a division by zero in those formulae happens when Spark does nothing with regards to the metrics in question (i.e., if tasks do not read any input bytes from storage in a stage), I set those metrics to 1 in those cases because there is no efficiency to lose from *nothing*. For the sake of readability, handling the default values of the following formulae is omitted in the mathematical notation.

In the following sections, I use $s \in \mathcal{S}$ to denote the currently evaluated stage.

4.3.1. POP Standard Metrics

In this section, I propose an approach to apply the POP1 standard metrics as introduced in Section 2.4 to Spark applications, which—apart from the serialization and transfer efficiencies—is mostly a matter of replacing processes in the original equations with threads.

Serialization Efficiency, Transfer Efficiency. As mentioned in Section 2.4, calculating the serialization efficiency and the transfer efficiency requires running a simulation of the application to obtain true stage runtimes on an ideal network. This poses a problem, since no such simulation exists for Spark; and due to Spark’s complexity, it seems fairly questionable whether creating one would be worth the work. However, it is still possible to get approximate results by simulating how the run time of each task and its scheduling *could* change if all network-related task timings were omitted. This is similar to the *blocked time analysis* approach in [7] that removes all network and disk-related task timings (*blocked times*) and then re-schedules all tasks. This comes close to an isolation of networking effects, however, there are at least two causes for inaccuracy: First, Spark might schedule tasks differently in practice with reaction to changed network properties (e.g., the locality of the data of a task does not matter anymore), and more importantly, tasks might actually do network-related work within the task’s useful computation time (e.g., by fetching storage data over the network), the duration of which cannot be extracted from the event logs. This likely keeps some network-related effects invisible, so the event logs alone are not sufficient to *completely* determine whether networking is a bottleneck.

4.3. Calculating POP Metrics for Spark Applications

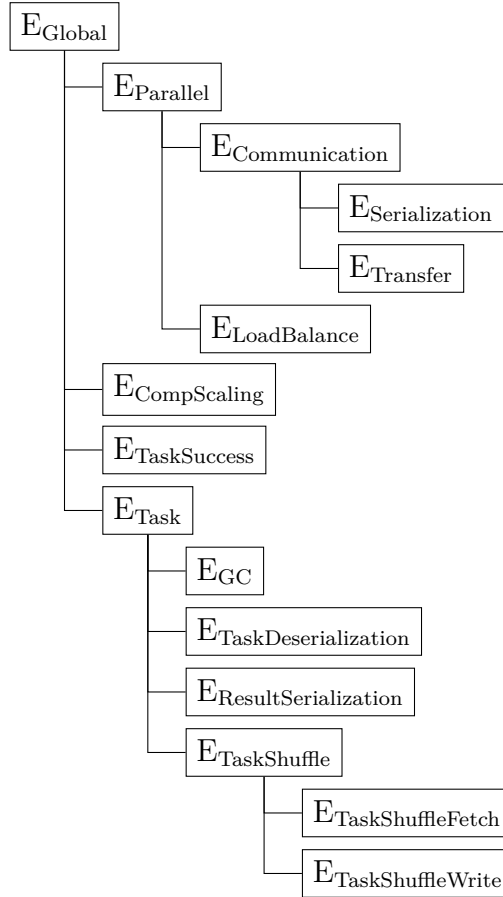


Figure 4.3.: Hierarchical overview of all POP efficiencies for Spark applications, including Spark-specific metrics (see Section 4.3.2). *Cause-identifying* metrics (see Section 4.3.2.1) are not included in this figure as they do not contribute to the global efficiency.

Unfortunately, the original POP metrics assume that the useful computation time of a process (or of a thread in case of Spark) does not change because of networking effects. This means that in both the original equations for the serialization efficiency and the communication efficiency (Equations 2.1 and 2.3), $t_{\text{Comp}}(p)$ is identical. When re-scheduling tasks on an ideal network, one would naturally ignore data locality and schedule each task as soon as any executor thread is available anywhere on the cluster. However, this would potentially result in different values for $t_{\text{Comp}}(p)$ on ideal and real networks, so in order to avoid this conflict, each task must be scheduled to the same thread regardless of the network in which the application is executed⁶. This further decreases the accuracy of the calculations, since Spark might schedule tasks differently if it was executed on an ideal network.

⁶ Incidentally, this reduces the complexity of the ideal network scheduler: It no longer needs to map tasks to threads, so its work is logically reduced to a mere approximation of the duration of a stage on an ideal network.

4. Implementing the POP Methodology

To calculate the total run time of a stage on an ideal network (see Algorithm 2), I do not only consider the maximum time any thread spends with useful computation as this would most likely always yield a serialization efficiency close to 1 (tasks within a stage have no dependencies between each other and thus 1) do not need to wait for each other and 2) can be scheduled immediately when there is a thread available). Instead, I delay each thread in the ideal network scheduling by the launch time of the first task in that thread relative to its stage, to make sure that any delays that happen between the submission of a stage and the launching of its first task are taken into account for calculating the results. Likewise, I retain the time span between the last task on the real network scheduling and the completion of its stage. Technically, this approach is not required and one could simply ignore the serialization efficiency for Spark applications, but it allows to see how much time a stage spends outside of any actual computation, even if that time might be different on a “real ideal” network. In some way this is similar to the serial component of *Amdahl’s Law* [76], which seems to be a meaningful way to use the serialization efficiency in Spark applications.

For the thread scheduler that simulates an ideal network, I define the useful computation time of a task $k \in \mathbf{K}$ as follows

$$\begin{aligned} k.\text{usefulCompTime} &:= k.\text{execRunTime} \\ &+ k.\text{execDeserializeTime} + k.\text{resSerializeTime}, \end{aligned} \quad (4.3)$$

and the corresponding thread attribute for a thread $t \in \mathbf{T}$ as

$$t.\text{usefulCompTime} := \sum_{k \in \mathbf{K}_t} k.\text{usefulCompTime}. \quad (4.4)$$

The above, perhaps somewhat unorthodox looking Equation 4.3 excludes the scheduler delay as well the time spent by fetching task results, however, the network-related $k.\text{shuffleFetchTime}$ (part of $k.\text{execRunTime}$) is included by purpose since it has its own efficiency $E_{\text{TaskShuffleFetch}}$ to obtain specific shuffle-related POP results (see Equation 4.16). If $k.\text{shuffleFetchTime}$ was excluded in the above definition, the scheduler would ignore it and in turn it would be included twice in the global efficiency.

Once the run time of the current stage in an ideal network has been calculated with `SCHEDULETHREADSIDEALNETWORK` as defined in Algorithm 2, the serialization efficiency can be calculated as follows:

$$E_{\text{Serialization}} = \frac{\max_{t \in \mathbf{T}_s} t.\text{usefulCompTime}}{\text{SCHEDULETHREADSIDEALNETWORK}(s)} \in [0, 1]. \quad (4.5)$$

The transfer efficiency is calculated as

$$E_{\text{Transfer}} = \frac{\text{SCHEDULETHREADSIDEALNETWORK}(s)}{s.\text{duration}} \in [0, 1]. \quad (4.6)$$

Communication Efficiency. The communication efficiency simply is the product of $E_{\text{Serialization}}$ and E_{Transfer} , however, one can calculate it without having to calculate the sub-metrics first as explained in Section 2.4:

$$E_{\text{Communication}} = \frac{\max_{t \in \mathbf{T}_s} t.\text{usefulCompTime}}{s.\text{duration}} \in [0, 1]. \quad (4.7)$$

Load Balance Efficiency. The load balance efficiency for Spark applications can be obtained by calculating the following:

$$E_{\text{LoadBalance}} = \frac{\frac{1}{|\mathbf{T}|} \sum_{t \in \mathbf{T}} t.\text{usefulCompTime}}{\max_{t \in \mathbf{T}} t.\text{usefulCompTime}} \in \left[\frac{1}{|\mathbf{T}|}, 1 \right]. \quad (4.8)$$

Note that the load balance is computed not only over all threads in the current stage (\mathbf{T}_s) but over all available threads (\mathbf{T}). This makes it possible to recognize configuration issues like a too small degree of parallelism that might cause Spark to create fewer partitions than threads (see Chapter 5). If stages of the evaluated application ran in parallel, it might be reasonable to use \mathbf{T}_s instead to only account for threads that were actually used by the current stage.

Transforming the unnormalized load balance metric into the normalized one as discussed in Section 2.4 works in the same way as Equation 2.5, one only has to replace $\frac{1}{P}$ by $\frac{1}{T}$ (or $\frac{1}{T_s}$). For the experiment results listed in Appendix B, I use the unnormalized original metric $E_{\text{LoadBalance}}$ to calculate the parallel efficiency in order to stay close to the POP specifications (see Figure 2.4). Due to the size of the tables, I do not include the results for the normalized metric, but given the amount of threads available in the experiments, the values of the normalized and unnormalized metrics did not differ much.

Parallel Efficiency. The calculation of the parallel efficiency does not change from the original definition in Equation 2.6. Just as with the original POP metrics, I use the unnormalized $E_{\text{LoadBalance}}$ for the parallel efficiency.

Computational Scaling Efficiency. As explained in Section 2.4, the computational scaling efficiency requires the application to run with a bare minimum of computational resources (i.e., a *reference* configuration) for comparison, in order to measure the effect of scaling these resources. For the experiments, I executed the workloads on one node with one executor using one core each, and a degree of parallelism of one, which amounts to one thread and one partition in the default case (Spark might still decide to use a different amount of partitions, however). All other configurations (e.g., the memory fraction, input size, as well as the driver/executor memory) remained unchanged in order to only consider strong scaling factors.

Since the POP metrics I propose for Spark are calculated for stages, one needs to adapt this concept to stages of a reference configuration. I define s_R to be the stage in the reference workload that corresponds to the currently evaluated stage s

4. Implementing the POP Methodology

(i.e., both stages have the same index). Then, the computational scaling efficiency for Spark applications is computed as follows:

$$E_{\text{CompScaling}} = \frac{s_R.\text{duration}}{s.\text{duration}} \in (0, \infty). \quad (4.9)$$

Global Efficiency. Because I define a set of POP metrics specifically for Spark applications in Section 4.3.2, the global efficiency does no longer only include E_{Parallel} and $E_{\text{CompScaling}}$ as in Equation 2.8, but also the Spark-specific metrics as defined in Section 4.3.2:

$$E_{\text{Global}} = E_{\text{Parallel}} \cdot E_{\text{CompScaling}} \cdot E_{\text{TaskSuccess}} \cdot E_{\text{Task}} \in (0, \infty). \quad (4.10)$$

4.3.2. Spark-specific POP Metrics

Although the POP standard metrics already cover a large area of possible performance bottlenecks, they alone do not give complete overview over the performance of a Spark application. The Spark event logs expose a lot of detailed information that can be used to obtain a broader image of performance bottlenecks of Spark applications, so in this section I propose additional POP metrics specifically designed for Spark applications.

Task Success Efficiency. In some applications it might happen that certain tasks do not terminate successfully. Spark distinguishes between two types of unsuccessful tasks: *failed* tasks terminate by themselves due to problems within their own scope of execution, whereas *killed* tasks are terminated from outside.

Failed tasks might occur for example when an executor runs out of memory (see [E17]). Killed tasks on the other hand can be a symptom of tasks that take an unusually long amount of time to complete (so-called *straggler tasks*) if Spark is configured to perform *speculative execution*. With speculative execution (not enabled for the experiments in this thesis, see Table 3.1), Spark spawns redundant copies of tasks if it concludes that the copies might finish sooner as their slow-running equivalents. If this assumption holds true, the slow running tasks are killed once their counterpart terminates successfully (cf. [77]).

Thus, the occurrence of many unsuccessful tasks is often an indicator of underlying problems, and it is self-evident that such situations can be critical for performance since it implies a waste of computational resources. To identify such cases, I propose a *task success efficiency*

$$E_{\text{TaskSuccess}} := \frac{\sum_{k \in \mathbf{K}_s} [\neg(k.\text{taskFailed} \vee k.\text{taskKilled})]}{|\mathbf{K}_s|} \in [0, 1], \quad (4.11)$$

where $[X]$ for a logical statement X is the *Iverson bracket* notation that evaluates to 1 if X is true and to 0 otherwise.

Task Efficiency. As shown in Figure 4.1, the duration of a task does not only consist of useful computation, instead it includes a lot of additional timings that ideally should be as short as possible in order to achieve the best efficiency. Most of the network-related task timings are already dealt with by the transfer efficiency, however, there exist other timings that so far are not part of any metric. The ratios of these additional timings are bundled in the *task efficiency*, whose sub-efficiencies are explained on the next pages. If the task efficiency is low, only a small amount of time is spent on average with useful computation in the tasks of the current stage.

$$E_{\text{Task}} := E_{\text{GC}} \cdot E_{\text{TaskDeserialization}} \cdot E_{\text{ResultSerialization}} \cdot E_{\text{TaskShuffle}} \in [0, 1] \quad (4.12)$$

GC Efficiency. In order to free previously allocated memory, Spark applications rely on garbage collection in the JVM processes. Most of the available implementations of JVM garbage collectors make use of the *mark-and-sweep* method, which works by frequently interrupting currently executing threads, searching for and marking unreferenced memory (*mark* phase), and then freeing it in the *sweep* phase. These frequent interrupts slow down the execution of programs by some degree, which makes garbage collection a potential significant performance bottleneck of Spark applications (cf. [35], [78], [79]). If the amount of available memory gets sufficiently small, the frequency of garbage collection might increase in order to keep enough memory available for execution (*GC pressure*), which in turn also increases the GC-induced overhead.

Garbage collection in the JVM comes in two qualities: *minor* collections and *major* collections. The minor garbage collection takes care of relatively young objects on the heap, whereas the major garbage collection frees objects that were used for a longer time and already survived multiple minor garbage collections [80]. Especially the number of major collections within an application is important, as “[m]ajor collections usually last much longer than minor collections because a significantly larger number of objects are involved” [80].

Investigations have shown that the amount of time spent with garbage collection in Spark directly correlates with the input data size, but with a larger-than-linear growth [79]. Garbage collection is an especially important performance factor for applications accessing many different RDDs as this increases the chance of evicting objects from memory [49, *Tuning Spark: Garbage Collection Tuning*]. For instance, Kang *et al.* conclude “that garbage collection accounts for almost 16–47% of the total elapsed time of running iterative algorithms on Spark” [35], Awan *et al.* come to similar results (48% for k-means, slightly lower values for WordCount) [79].

Apart from the application’s workload, a high garbage collection overhead can also be a consequence of Spark internals or the JVM itself: For example, Spark’s *Unified Memory Manager*, which is used to manage the heap memory, usually allocates more execution memory than needed [78], [81]. This improves the performance for operations on the execution memory region, but on the other hand can result in a longer time spent with garbage collection (among other issues) [78], [81]. Also, garbage

4. Implementing the POP Methodology

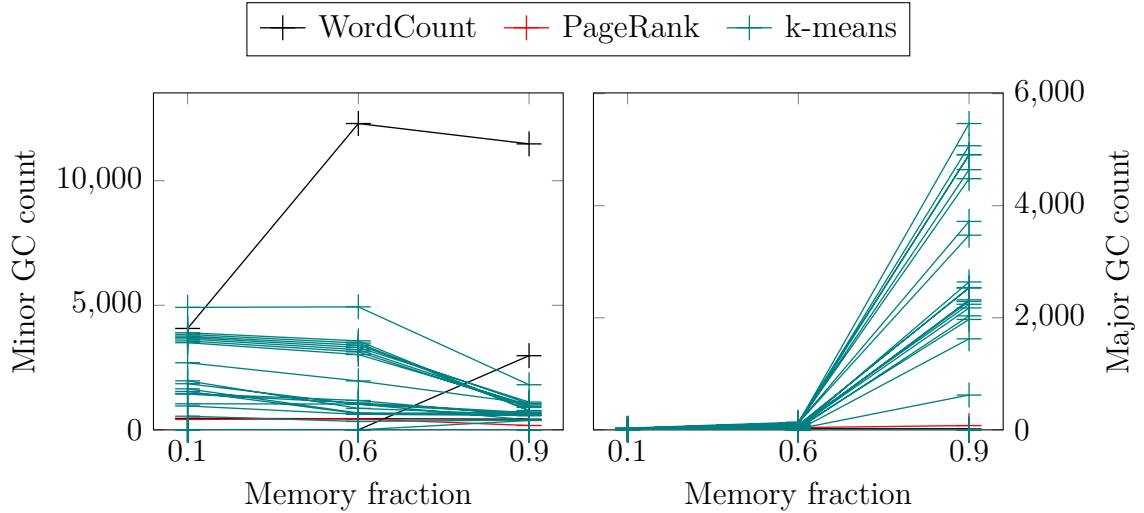


Figure 4.4.: Amount of minor and major GC invocations per stage based on the memory fraction for all conducted experiments in which only the memory fraction was varied. Data points connected by a line belong to the same stage using the same parameters apart from the memory fraction.

collection itself tends to be “memory hungry” [31], which reduces the amount of memory an executor can use for its computation purposes and in turn artificially increases the GC pressure.

To notice cases in which garbage collection appears to be a bottleneck, I propose the following *GC efficiency*:

$$E_{GC} := 1 - \frac{\sum_{k \in K_s} k.jvmGCtime}{\sum_{k \in K_s} k.totalDuration} \in [0, 1]. \quad (4.13)$$

To further investigate garbage collection issues, one can make use of dedicated JVM flags to obtain debug logs about the garbage collector [82]. These can be visualized in tools like *GCViewer* [83].

In the experiments, increasing the memory fraction for iterative workloads also increased the amount of time spent in garbage collection, resulting in a decreased GC efficiency (see [E16], [E22], [E28]). This appears to be caused by an increase in GC pressure due to little available user memory that in turn causes the garbage collector to become active more often. The amount of minor GC invocations slightly decreased with a higher memory fraction, but the amount of major GC invocations increased, especially for the case of 0.9 as the memory fraction (cf. Figure 4.4).

Task Deserialization Efficiency, Result Serialization Efficiency. The time taken by deserializing a task on an executor can have a major performance impact as well. For instance, in *Azure Databricks Data Science & Engineering* (a data analytics tools based on Spark), a certain kind of library (a so-called *cluster-installed library*) is “only installed on the executors when the first tasks are submitted” [84]. The time

4.3. Calculating POP Metrics for Spark Applications

spent by installing those libraries is then counted as part of a task’s deserialization time [84]. Although cases like this are probably rare in practice, it makes sense to define a metric that recognizes them:

$$E_{\text{TaskDeserialization}} := 1 - \frac{\sum_{k \in \mathbf{K}_s} k.\text{execDeserializeTime}}{\sum_{k \in \mathbf{K}_s} k.\text{totalDuration}} \in [0, 1]. \quad (4.14)$$

Likewise, it could theoretically happen that the amount of time a task spends with serializing the result is a bottleneck, so I also define a metric for that purpose:

$$E_{\text{ResultSerialization}} := 1 - \frac{\sum_{k \in \mathbf{K}_s} k.\text{resSerializeTime}}{\sum_{k \in \mathbf{K}_s} k.\text{totalDuration}} \in [0, 1]. \quad (4.15)$$

In the conducted experiments, there were no cases of exceptionally long deserialization or serialization times.

Task Shuffle Efficiency. To make performance problems caused by shuffling noticeable, I define two metrics for the amount of time spend with fetching and writing shuffle data, respectively. These metrics are combined in the superordinate *task shuffle efficiency*.

$$E_{\text{TaskShuffleFetch}} := 1 - \frac{\sum_{k \in \mathbf{K}_s} k.\text{shuffleFetchTime}}{\sum_{k \in \mathbf{K}_s} k.\text{totalDuration}} \in [0, 1] \quad (4.16)$$

$$E_{\text{TaskShuffleWrite}} := 1 - \frac{\sum_{k \in \mathbf{K}_s} k.\text{shuffleWriteTime}}{\sum_{k \in \mathbf{K}_s} k.\text{totalDuration}} \in [0, 1] \quad (4.17)$$

$$E_{\text{TaskShuffle}} := E_{\text{TaskShuffleFetch}} \cdot E_{\text{TaskShuffleWrite}} \in [0, 1] \quad (4.18)$$

4.3.2.1. Cause-identifying POP Metrics

The POP metrics that were introduced so far all deal with performance bottlenecks. Each decrease in value directly corresponds with a loss of efficiency compared to an optimal case, and the mathematical “weight” of those metrics in the resulting global efficiency does not matter because the loss in efficiency itself controls the weight. This is the essence of the original POP methodology that I want to preserve with the metrics proposed so far, however, the information contained in Spark’s event logs allows to get more insight about *potential causes* of performance problems. Thus, in this section, I propose three additional metrics that are motivated by known reasons for performance problems in Spark, however, their values do not say much about the actual *importance* of the recognized problems. Consequently, it is not possible to determine useful weights for them, so I do not include the following metrics in the global efficiency and only use them as accompanying metrics for further investigations.

4. Implementing the POP Methodology

Input Balance. A common cause of load-balance-related performance problems in real-world Spark applications is a poor distribution of data among partitions (and thus potentially among executors or nodes), called *data skew* [85], [86]. If amount of data reads is distributed unevenly between partitions, some tasks within a stage may take longer to compute than others, which in many cases introduces unnecessary waiting time to the system. This is known to have a major impact on an application’s performance in some cases [85], [86].

In order to catch issues caused by poor data distribution, I propose an *input balance* metric that describes the balance of all used threads of the current stage in terms of the amount of read input bytes:

$$E_{\text{InputBalance}} := \frac{\frac{1}{|\mathbf{T}_s|} \sum_{t \in \mathbf{T}_s} t.\text{inputBytesRead}}{\max_{t \in \mathbf{T}_s} t.\text{inputBytesRead}} \in \left[\frac{1}{|\mathbf{T}_s|}, 1 \right]. \quad (4.19)$$

As already mentioned before, \mathbf{T}_s is used instead of \mathbf{T} in order to decorrelate the input balance as much as possible from the load balance in case not all threads are utilized by the current stage. This way, the input balance metric can be used to investigate reasons for a bad load balance if an insufficient utilization of threads can be ruled out.

One could easily adapt this metric to compute the balance over all tasks within a stage instead of just computing the balance between threads, but then the resulting efficiency would not only consider data skew (a *spatial* effect), but also the *temporal* distribution of work which is not particularly relevant to the application as long as the work is equally distributed among threads. Similar to the transformation of the load balance efficiency in Equation 2.5, the input balance can be normalized to $[0, 1]$ as well, if desired.

Task Locality. Generally speaking, the closer data is to a node, the faster the node is able to access it due to less latency and less network congestion [52]. Thus, data locality can be a potential performance bottleneck, especially on clusters with slow network hardware. RDDs are equipped with a function `getPreferredLocations` that returns a sequence of preferred hostnames for a given partition, which, when creating tasks, Spark’s `DAGScheduler` together with the `TaskSetManager` use to determine the actual location for a task [48, *Internals: Scheduler > DAGScheduler* and *Internals: Scheduler > TaskSchedulerImpl*]. Once a task is created and pending execution, the `TaskSetManager` assigns a *locality* level to it that describes where the task is located relative to the data it is accessing (see Table 4.4).

To detect potential issues related to data locality, I propose a *task locality* metric, which assigns a score $\in [0, 1]$ to each task locality level and then computes the average score of all tasks within a stage:

$$E_{\text{TaskLocality}} := \frac{1}{|\mathbf{K}_s|} \sum_{k \in \mathbf{K}_s} \text{LocalityScore}(k.\text{taskLocality}) \in [0, 1]. \quad (4.20)$$

Locality Level	Description
PROCESS_LOCAL	The task's data is in the same JVM process as the task.
NODE_LOCAL	The task's data is on the same node as the task.
RACK_LOCAL	The task's data is on the same server rack as the task.
ANY	The task's data is not on the same server rack as the task, but anywhere else on the network.
NO_PREF	The access time for the task's data does not depend on the task's location.

Table 4.4.: List of all Spark task locality levels. Adapted from [49, *Tuning Spark: Data Locality*].

LocalityScore is a function that maps a given locality level to its score, defined by the following function graph:

$$\begin{aligned} &\{(\text{PROCESS_LOCAL}, 1.0), \\ &\quad (\text{NODE_LOCAL}, 0.\bar{6}), \\ &\quad (\text{RACK_LOCAL}, 0.\bar{3}), \\ &\quad (\text{ANY}, 0), \\ &\quad (\text{NO_PREF}, 0.5)\}. \end{aligned} \tag{4.21}$$

In general, the PROCESS_LOCAL level is preferable for tasks as it should result in the fastest data access times. Because the data locality is not specified for NO_PREF, it is not possible to estimate data access latencies in that case. For this reason, I set its score to 0.5.

There are many causes for data locality problems: For example, the partitioning of the input data in HDFS can lead to cases in which all the input data of an application is stored only on the namenode (an extreme instance of data skew). In the conducted experiments, it is a common pattern that $E_{\text{TaskLocality}}$ is low especially in the first stage(s). Even though I am not able to verify this at the time of writing, I suspect that the HiBench benchmark suite creates the input data only in HDFS blocks on the namenode. All datanodes are registered and live according to the output of the `hdfs dfsadmin -report` command.

Although the task locality metric is a good example of a POP metric that can be used to notice misconfigurations of the cluster or application, task locality issues might not only emerge by an application's non-optimal configuration, but also by the scheduling of multiple competing Spark jobs on a cluster: To improve both data locality and throughput on multi-user clusters, Zaharia *et al.* have proposed *Delay Scheduling*, which lets jobs wait for a small duration of time if they cannot be

4. Implementing the POP Methodology

launched local to their data [87]. Delay Scheduling is now part of Apache Spark [88] and can be configured by the `spark.locality.wait.□` properties that default to 3s of additional waiting [49, Chapter *Spark Configuration*].

Memory Spill. When the size of partitions exceeds the available memory of the executor to which they belong, Spark temporarily *spills* the entire data of some of the executor’s partitions to disk. Since spilled data needs to be loaded into memory again in order to be processed and not all spilled data can fit into memory at once, data spills can cause serious performance issues [89]. To find issues caused by spilled data, I propose a dedicated *memory spill* metric.

Ideally, one would want the memory spill metric to describe how much memory was spilled to disk compared to the total required memory on an ideal machine with an infinite amount of working memory. This would allow to define a POP metric with values in $[0, 1]$:

$$\frac{\text{Number of bytes spilled in the current stage } s}{\text{Total required memory to store all data in RAM}} \in [0, 1].$$

The value of the denominator is almost logically equivalent⁷ to the total amount of unique memory that was allocated in any point in time in order to store the partition data. Unfortunately but unsurprisingly, Spark does not track and provide that information, which makes it impossible to calculate the idealistic version of the metric. Instead, one needs to find other values for the denominator of the metric’s formula to approximate the result.

The only Spark metrics related to a task’s memory usage represent the amount of memory used at a fixed point in time (that is, either the amount of currently allocated memory or the maximum used memory so far). This poses a problem, since memory might get spilled multiple times over the execution time of a task and thus the amount of bytes spilled in a stage has no upper bound and might grow larger than the memory usage at any point in time, resulting either in a POP metric outside the normalized range or a hard cutoff (if clamping values) that would not allow for arbitrary degrees of “badness” in the results. There exist various heap-related metrics for *executors*, however, they show the same kind of problem. In addition, using them would count spilled tasks twice in the result as such tasks contribute to the Heap memory metrics before the executor runs out of memory, and they add to the amount of spilled memory after their data gets spilled to disk. Luckily, we know from the `k.bytesSpilled` task metric how much memory a spilled task would require if it was kept in memory (at least the metric is a very close approximation of that), so we only need to find a replacement for the overall amount of memory required by tasks whose partition is not spilled to disk.

This replacement does not exist in the available data, so one might come to the conclusion that the only meaningful and practical way to define a memory spill

⁷ Apart from implementation details; the application might technically allocate slightly different amounts of memory depending on various factors.

4.3. Calculating POP Metrics for Spark Applications

metric is to count how many tasks were spilling memory and then compare it with the total amount of tasks in a stage. Unfortunately, this quantitative approach does not differentiate between the potential performance impact of different spilled tasks (i.e., a task spilling a few megabytes of data likely is much less of a performance problem than a task spilling dozens of gigabytes). For this reason, I think that is more useful to give up some accuracy in favor of gaining more insight about the strength of data spilling by using the task's `k.peakMemory` metric to (roughly!) estimate the amount of execution memory a task needs in total over time. This does not yield the most precise results as the value is likely too small, but on the other hand this gives large amounts of spilled bytes some extra weight. In addition, spilled data can also originate from the storage memory region of the heap (cf. [89]), but no proper task-based metric exists for that purpose (`k.peakMemory` only considers execution memory [49, Chapter *Monitoring and Instrumentation*]).

I define the memory spill metric as follows:

$$E_{\text{MemorySpill}} := 1 - \frac{\sum_{k \in \mathbf{K}_s} k.\text{bytesSpilled}}{\sum_{k \in \mathbf{K}_s} \text{MemUsedBytes}(k)} \in [0, 1], \quad (4.22)$$

with

$$\begin{aligned} \text{MemUsedBytes}(k): \mathbf{K} &\rightarrow \mathbb{N}_0, \\ k &\mapsto \begin{cases} k.\text{bytesSpilled} & \text{if } k.\text{bytesSpilled} > 0 \\ k.\text{peakMemory} & \text{otherwise.} \end{cases} \end{aligned} \quad (4.23)$$

The metric evaluates to 1 if no bytes were spilled in the current stage, and to 0 if all tasks were spilled. The higher the peak execution memory of all non-spilled tasks is, the less influence spilled tasks have in the result. The higher the amount of spilled bytes is compared to a fixed peak memory, the lower the resulting efficiency is.

5. Workload Performance Analysis

Although some experimental results were already presented in the last chapter, I want to give a brief overview about my findings for the tested workloads (see Appendix B) as well as observed limitations of both the conducted experiments and the proposed POP metrics. As already mentioned in Chapter 3, the quota of the thesis project on the compute cluster did not allow for redundant experiments, which is why the results are not averaged from multiple runs. Accordingly, please treat the following findings with some care.

5.1. Experiment Results

Generally, the network performance of CLAIIX-2018 does not seem to be an issue according to the calculated POP metrics, similar to what is claimed in [7] and [13] for clusters in general. However, one needs to keep in mind that CLAIIX-2018 is an HPC cluster specifically equipped with fast networking hardware, so the results might vary on clusters running on commodity hardware. More importantly, the POP metrics only consider task timings as provided in the Spark event logs, which do not contain information about the time spent by waiting for storage data to arrive on an executor. For this reason, the current set of metrics simply might not be able to recognize some disk and network related bottlenecks, as explained in detail on the following pages.

Throughout the experiments, the $E_{\text{TaskLocality}}$ exhibits rather bad values due to what was observed on p. 39. This could be a side effect of running the applications on an HPC cluster, more validation is required here.

WordCount. The WordCount workload generally seems to run fairly well apart from some exceptions, the experiments show very good communication and task efficiencies, and memory spilling is not an issue at all. The memory fraction does not seem to have a high impact on the WordCount workload, indicating—together with the absence of spilling memory—that memory does not need to be considered an issue for WordCount.

However, the second stage shows a bad load balance in many cases, which appears to be caused by configuring a too small degree of parallelism compared to [E5], [E12]. This results in an incomplete utilization of available threads, since the `reduceByKey` transformation “hash-partitions the output with the number of partitions (*i.e.* the default parallelism)” [14]. Although it should be possible to achieve a much better utilization of threads in the second stage by configuring a higher

5. Workload Performance Analysis

degree of parallelism, this factor was insignificant for the conducted experiments as the second stage ran for less than a second and thus does not play an important role in improving performance. This fact can be seen very well when looking at the averaged metrics, which are barely influenced by very short stages and should be considered first when looking for performance bottlenecks.

Another interesting observation in the second stage is that the computational scaling often shows values smaller than 1, which indicate that the reference stage was faster than the actually evaluated stage. Given the short runtimes of those stages, however, this could be caused by the overhead of handling multiple threads, and one can expect some statistical noise too. Some overhead can be seen as well in the task deserialization efficiency, which is likely caused by the fact that even a small absolute deserialization overhead takes a decent amount of relative time for very short tasks.

Similar effects as with the generally observed $E_{\text{TaskLocality}}$ values can be seen for $E_{\text{InputBalance}}$, whose values are mostly mediocre (and sometimes even fairly problematic [E14]) in the first stage apart from some exceptions like [E7]. It is not clear what causes this imbalance, although it is possibly related to the task locality (more research is required here).

The first (longer running) stage runs much better than the second one, however, it seems that the computational scaling in that stage is—to some extent—correlated with the task locality. This could indicate that there exists a bottleneck that the current set of POP metrics cannot identify. After analyzing hardware utilization metrics, Shi, Qiu, Minhas, *et al.* came to the conclusion that the first stage of the WordCount workload is disk-bound [25], but unfortunately, Spark does not provide the time spent by reading input data from storage in the event logs. This makes disk I/O as well as some network-related work mostly invisible in the current set of metrics, so other sources of information need be considered in order to derive a metric for this purpose in the future, as discussed in Section 5.2.

With `huge` as the configured input size, the load balance decreases if more hardware parallelism is added [E2], [E6], [E7]. This is caused by straggler tasks, which in the conducted experiments are strongly correlated to the task locality: The slow tasks in [E6] and [E7] have a task locality of `RACK_LOCAL` whereas the task locality of the remaining tasks is `NODE_LOCAL` (see Figure 5.1). For [E2], on the other hand, the task locality is `RACK_LOCAL` for *all* tasks. The WordCount experiments with an input size of `gigantic` have the same issue, however, they had enough small tasks to mostly catch up the slow straggler tasks, in turn the amount of computation time was more evenly distributed among threads. Although the metrics encompassed within E_{Task} as well as the transfer efficiency might at first look as if there was no causality between the task locality and the load balance in this case (all tasks spent most of their time within active computation), this might again be caused by the missing inclusion of storage read times in the metrics, as mentioned in the previous paragraph. Even if it currently cannot be verified for sure whether there is a causal relationship between the task locality and the duration of tasks in the experiments, it certainly looks as if data locality—contrary to what some publications claim—still

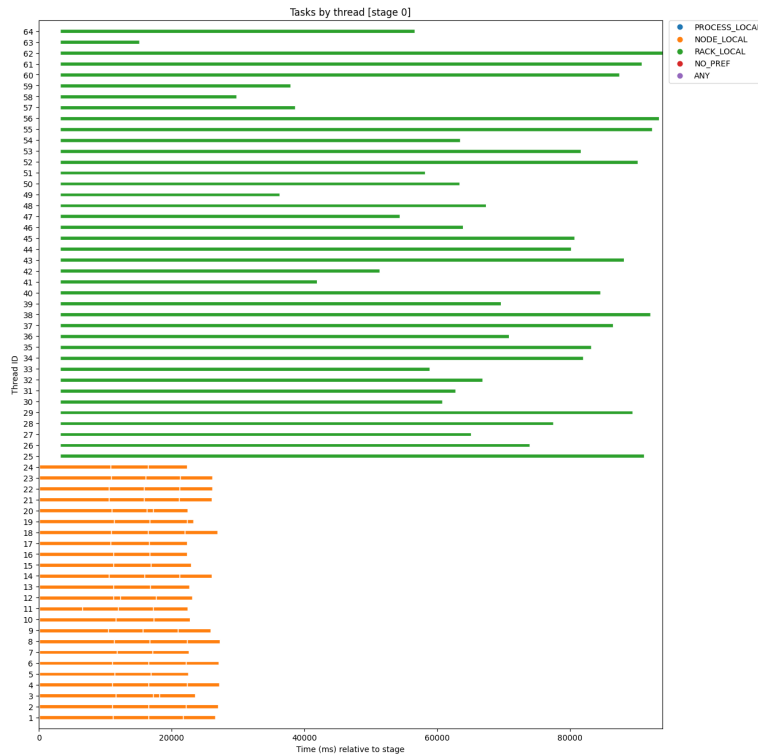


Figure 5.1.: Task scheduling for the first stage of experiment [E6]. The tasks are colored by their task locality; orange tasks are `NODE_LOCAL`, whereas green tasks are `RACK_LOCAL`. The x-axis describes the time span of tasks in ms relative to the stage, the y-axis describes the thread id.

matters to some degree on modern hardware, even on HPC clusters.

PageRank. The PageRank workload generally seems to struggle a lot with the tested configurations. In many stages, all tasks are spilled to disk, which suggests that the amount of memory allocated to each executor might be too low and too few tasks are spawned. The small amount of tasks (and in turn also partitions) causes similar load balancing problems as already observed in the second stage of the WordCount workload, this effect is particularly strong in stages 2–5 that use `reduceByKey` and thus only have as much partitions as configured. Using 8 nodes appears to be a sweet spot for both the task locality and the amount of memory spilled [E18], for the latter I suspect that the distribution of 8 tasks to 8 executors does not exceed the memory of the executors whereas in [E19] with 16 configured executors and 16 tasks, Spark decided to only use 10 executors in reality.

The PageRank workload seems to have a fairly constant amount of garbage collection invocations throughout all tries although increasing the memory fraction also slightly increased the amount of major garbage collection invocations as observed on p. 36. With a memory fraction of 0.9, the workload even crashed due to running out of memory [E17], but even the incomplete first stage in that experiment already

5. Workload Performance Analysis

represents the highest count of major GC invocations I measured in the PageRank experiments. Interestingly, the relative time spent with garbage collection increased after adding more hardware parallelism [E18], [E19], but no reason for this increase can be directly deduced from the POP metrics. The duration of individual stages even increased in some cases, thus speaking against a constant absolute amount of time spent with garbage collection. More work needs to be done to investigate the reasons for this.

Although each stage apart from the first and last stage reads about 3 GB of data during shuffle operations, the shuffle efficiency as well as the closely related transfer and serialization efficiencies show almost perfect results for all tested PageRank applications. This—in contrast to the observed load balance issues—supports the common claim that network latencies do not greatly affect the performance of Spark applications.

K-means. The k-means workload generally has similar problems as PageRank regarding high memory fractions (this suggests that garbage collection is an especially important factor for the performance of iterative algorithms) and similar load balancing problems as the other workloads due to partitioning and the use of `reduceByKey`. Even the default memory fraction of 0.6 appears to cause some decent garbage collection overhead compared to a memory fraction of 0.1 [E20], [E20], suggesting that the application uses quite a lot of user memory.

The middle stages that correspond to the `collectAsMap` jobs (see Figure 3.3) exhibit a striking alternating pattern in many efficiencies. The odd-numbered stages mostly use much fewer partitions than the even-numbered ones (again due to a too low configuration value in combination with using `reduceByKey`), and thus the load balance suffers. Since the odd-numbered stages only took a fraction of a second to complete, various timings encompassed in the task efficiency E_{Task} took a big part of a task’s total duration in many experiments. In those stages, a similar computational scaling as with the WordCount workloads can be observed, potentially due to the same reasons.

The input balance varies a lot throughout the K-means experiments, but it exhibits especially low values for [E25]. Similar to the WordCount workload, this is caused by straggler tasks in two executors that appear to take a much longer time to complete than others. However, this is not related to the task locality, in stage 3 for example, all tasks have the best possible task locality of `PROCESS_LOCAL`. More investigations are required to find out what is causing the straggler tasks, and it is possible that the reason is not detectable by the current set of POP metrics.

5.2. Limitations

In this section, I give a brief summary about the most important limitations of the proposed framework, some of which were already mentioned on a handful of occasions throughout the thesis.

Spark applications likely run slightly different in a real-world scenario than in experiments due to overhead that is caused by collecting performance metrics. The creation of event logs requires both a small amount of computational resources and working memory, and sending accumulator metrics to the driver can cause performance problems in some cases [74]. Any additional logging such as collecting hardware metrics (e.g., by using LIKWID [90]) or garbage collection statistics (by passing certain parameters to the JVM) to conduct a more in-depth analysis of Spark applications also likely introduces further overhead.

Only a small set of available configuration parameters and workloads was tested in this thesis, which might not have provoked all possible performance issues in the experiments. Likewise, the conducted experiments alone do not suffice to validate that each metric 1) correctly identifies all cases of its corresponding bottleneck (completeness), and 2) *only* reacts to the bottleneck to which it should correspond (correctness). Furthermore, the missing averaging of test results as mentioned in Section 3 sadly leads to an unknown amount of noise and distortion in the results. However, there is a good chance that the noise level was sufficiently low and results are still comparable to some degree, as in each experiment, the compute nodes were allocated exclusively to the application.

Further potential causes of inaccuracies in the results are the use of accumulators to gather Spark metrics as explained in Section 4.1, the usage of default values for some missing attributes of unsuccessful tasks, and the uncertainty about what exactly counts as scheduler delay (see Section 4.2). In the experiments, the latter only amounted to a few milliseconds, so the impact of my decision to treat the scheduler delay only as the delay caused by sending a task to an executor is probably completely negligible.

As already mentioned in Section 4.2, Spark does not expose information about which thread executed a task. Thus, a custom thread scheduler simulation is required, which 1) causes marginal inaccuracies in the POP results and 2) might break completely when the event logs contain slightly overlapping task timings and, at the same time, Spark uses a different amount of cores per executor than configured. Furthermore, the thread scheduling only roughly approximates the effects of an ideal network for the serialization and transfer efficiencies, however, as there obviously is no such thing as an ideal network in reality, it is disputable whether there is a non-approximative solution anyways.

Compared to the other metrics, the computational scaling efficiency is a mathematical outlier by construction in terms of the range of its values. This is due to the fact that it neither really describes a bottleneck nor a cause for performance issues, but instead it expresses additional information about how well an application scales with the amount of parallel resources. For that reason, I am somewhat sceptical whether it makes sense to include scaling effects in the global efficiency, however, as this is the case for the original POP metrics, I abide by the decision of the original authors and treat it like any other metric in the results.

By far the biggest limitation, however, is that the proposed set of metrics that is based on Spark’s event logs still does not yield a fully complete image of all possible

5. Workload Performance Analysis

performance bottlenecks. For instance, the event logs do not contain any information about how much time a task or executor spent with reading input data from storage (contrary to shuffle data fetches, which are considered by the $E_{\text{TaskShuffleFetch}}$ metric). Although some of the *cause-identifying* metrics proposed in Section 4.3.2.1 can probably *hint* at certain bottlenecks as demonstrated in Section 5.1, one needs to consult other sources of information to see whether a stage is actually disk-bound, for example. Similar problems might arise with applications that are limited by their CPU cache utilization or by network effects outside of task timings reported by Spark (e.g., the time it takes to send input data from storage over the network), which also cannot be identified by the current set of metrics. Originally, it was planned to collect various hardware statistics with LIKWID in order to get more detailed information about such bottlenecks and the factors influencing the computational scaling efficiency, but unfortunately, there was an issue that caused LIKWID's output logs to always be empty¹, which could not be resolved in a timely manner.

¹ See <https://github.com/RRZE-HPC/likwid/issues/512>.

6. Summary and Outlook

6.1. Summary

In this thesis, I presented an initial proof of concept for using the POP methodology to systematically find bottlenecks of arbitrary Apache Spark applications. When I began working on this thesis, it was not evident whether this is actually possible, because the POP methodology was originally developed for software in the context of high-performance computing. Over time, however, it became clear that it is indeed viable and one can derive meaningful results from the proposed metrics. Although some metrics only yield approximative results and there is no fully reliable way of mapping tasks to the threads that executed them, it is possible to adapt all standard POP metrics to Spark applications. To obtain a broader and more detailed view on possible performance bottlenecks that can affect Spark applications, I proposed a Spark-specific extension of the original POP metrics.

The POP methodology is especially useful when analyzing large workloads with thousands of tasks, since it reduces the complexity of the analysis to a fixed set of factors that one can consult to quickly find entry points for further performance investigations. Generally, it is possible to use the proposed set of metrics to identify specific performance bottlenecks, and the additional *cause-identifying* metrics provide further information to pinpoint the reasons for a bottleneck as demonstrated in Section 5.1, extending the set of use cases of the original POP metrics. Beyond improving individual applications, it should be possible with the proposed framework to systematically evaluate and compare the effects of different configurations and hardware when running Spark applications. While this approach seems to work well in principle, the framework still needs validation and the development of further metrics in order to yield a complete and correct overview about all possible performance bottlenecks. Given the widespread use of Apache Spark as well as the increasing convergence of HPC and big data, I hope that this thesis is a useful contribution that enables further research on gaining insight about the performance of Apache Spark applications.

6.2. Outlook

This thesis only constitutes a starting point for investigating Spark performance with the help of POP metrics. Further research can build upon this and study both individual (and possibly additional) metrics as well as the application of the POP

6. Summary and Outlook

framework on Spark applications as a whole. The observed limitations as discussed in Section 5.2 can act as further inspiration for possible research topics.

More experiments with other parameters and workloads are needed to solidify the presented methodology, the conducted experiments only covered a portion of potential Spark applications and the proposed framework as a whole lacks battle-hardening. A particularly interesting and yet open question is how well the proposed POP metrics work for applications running multiple stages in parallel. In those cases, the current metrics might indicate performance issues where there are none due to each stage only having access to some of the available resources. Related to that, it might be worthwhile to also consider a more high-level view on Spark applications to evaluate effects that are related to whole stages or executors. For example, the framework proposed in this thesis does not consider the time Spark spends outside of stages (e.g., program initialization; cf. [40, Equation 1]) or losses in efficiency caused by re-executing individual stages in case of failed execution attempts.

This thesis did not evaluate the calculation and usage of POP metrics of running applications for real time analysis, which according to Gopalani and Arora are “almost ubiquitous [...] in the industry” [17]. As long as information about individual events is emitted by Spark while it runs an application, calculating POP metrics is technically possible, but both individual metrics as well as the overall approach might need to be adjusted to obtain meaningful results for running applications. Closely related to this topic is Spark’s streaming mode that only works on chunks of data at a time [17], which is also not considered in this thesis and requires further investigation.

Instead of relying only on the Spark event logs to gather performance information, it is possible to collect more data from hardware performance counters (e.g., by using LIKWID), garbage collection statistics or benchmark reports, among others. These additional data sources might be especially meaningful given that many Spark applications seem to be CPU-bound. By combining measurements from multiple sources, it should be possible to get a more in-depth view on application performance that can help to validate the completeness, correctness, and usefulness of individual POP metrics.

To make the proposed framework accessible to a broader audience outside of scientific research, it might be worthwhile to implement it as a standalone application once the proposed metrics are better understood and validated.

A. Code Snippets

The following is a collection of algorithms referenced in the thesis. In addition to the task, thread, and stage attributes defined in 4.1, I define (for any $k \in \mathbf{K}$, $t \in \mathbf{T}$, and $s \in \mathbf{S}$):

- $k.\text{arrivalTime}$: The timestamp in ms at which the task arrives at the executor as explained in Section 4.2. Calculated as $k.\text{launchTime} + k.\text{schedulerDelay}$.
- $k.\text{executorId}$: The id of the task’s executor $\in \mathbb{N}^+$ as stored in the Spark task metric `Executor ID`.
- $k.\text{threadId}$: The id of the thread $\in \{0, \dots, |\mathbf{T}| - 1\}$ to which a task is scheduled by Algorithm 1.
- $t.\text{index}$: The index of a thread $\in \{0, \dots, |\mathbf{T}| - 1\}$.
- $s.\text{submissionTime}$: The timestamp at which the stage is submitted (in ms).
- $s.\text{completionTime}$: The timestamp at which the stage is completed (in ms).

Furthermore, I define the following functions:

- `EMPTYLIST(size, default)`: Create an empty list of size *size* and initialize each element with the value *default*.
- `SORTSET(set, attr)`: Sort the given *set* in ascending order by the given *attr* as key, and return the result as a list.
- `MAX(numA, numB)`: Return the larger of both input parameters.
- `MAXLIST(list)`: Return the largest item from the given list.

Algorithm 1 – Simplified thread scheduling simulation for a Spark application that was executed on a real network (without simulation of waiting or error handling as discussed in Section 4.2).

Input s : the stage whose tasks to schedule
Input numCoresExec: the configured number of cores per executor

```

1: function SCHEDULETHREADS( $s$ , numCoresExec)
2:   threadsTime  $\leftarrow$  EMPTYLIST( $|\mathbf{T}|$ , 0.0)  $\triangleright$  Next available time slot for each thread
3:   threadsExecutorIds  $\leftarrow$  EMPTYLIST( $|\mathbf{T}|$ , -1)  $\triangleright$  Executor id for each thread
4:
5:   usedExecutorIds  $\leftarrow$   $\emptyset$   $\triangleright$  Executors actually used by tasks in this stage
6:   for  $k$  in  $K_s$  do
7:     usedExecutorIds  $\leftarrow$  usedExecutorIds  $\cup$   $\{k.executorId\}$ 
8:   end for
9:
10:   $t \leftarrow 0$ 
11:  for  $e$  in usedExecutorIds do
12:    for  $i$  in 0 to numCoresExec do
13:      threadsExecutorIds[ $t + i$ ]  $\leftarrow e$ 
14:    end for
15:     $t \leftarrow t + \text{numCoresExec}$ 
16:  end for
17:
18:  for  $k$  in SORTSET( $K_s$ , arrivalTime) do
19:    for  $t$  in  $\mathbf{T}$  do
20:      threadExecutor  $\leftarrow$  threadsExecutorIds[ $t.index$ ]
21:
22:      if  $k.executorId \neq \text{threadExecutor} \wedge \text{threadExecutor} \neq -1$  then
23:        continue  $\triangleright$  Only assign task to threads from the same executor
24:      end if
25:
26:      if  $k.arrivalTime \geq \text{threadsTime}[t.index]$  then
27:         $k.threadId \leftarrow t.index$ 
28:        threadsTime  $\leftarrow k.finishTime$ 
29:        threadsExecutorIds  $\leftarrow k.executorId$ 
30:        break
31:      end if
32:    end for
33:  end for
34: end function

```

Algorithm 2 – Simplified thread scheduling simulation for a Spark application that approximates the scheduling on an ideal network (without error handling).

Input s : the stage whose tasks to schedule

Output: the total run time of s on an ideal network

```
1: function SCHEDULETHREADSIDEALNETWORK( $s$ )
2:   threadsTime  $\leftarrow$  EMPTYLIST( $|\mathbf{T}|$ , -1.0)  $\triangleright$  Next available time slot for each thread
3:   maxTaskFinish  $\leftarrow$  0  $\triangleright$  Latest point in time at which a task finishes
4:
5:   for  $k$  in SORTSET( $\mathbf{K}_s$ , launchTime) do
6:     if threadsTime[ $k$ .threadId] = -1.0 then
7:       threadsTime[ $k$ .threadId]  $\leftarrow$   $k$ .launchTime -  $s$ .submissionTime
8:     end if
9:
10:    threadsTime[ $k$ .threadId]  $\leftarrow$  threadsTime[ $k$ .threadId] +  $k$ .usefulCompTime
11:    maxTaskFinish  $\leftarrow$  MAX(maxTaskFinish,  $k$ .finishTime)
12:  end for
13:
14:  stageCompletionOverhead  $\leftarrow$   $s$ .completionTime - maxTaskFinish
15:  return MAXLIST(threadsTime) + stageCompletionOverhead
16: end function
```

A. Code Snippets

B. Experimental Results

This appendix chapter contains the calculated POP metrics for all experiments that were conducted as part of this thesis. Please keep in mind that different stages within an application do different work and thus should not be directly compared with each other, even if the column-based (re-)presentation in this appendix (for the purpose of compaction) might suggest so. The last column in each table denoted with the “ \emptyset ” symbol represents the weighted average of each stage as defined in Equation 4.2. The written number of nodes corresponds to the number of *worker* nodes. Aside from POP metrics, each table contains additional information about each stage at the bottom (not colored). All given values are rounded to two decimal points.



Figure B.1.: Colormap used for printing POP scores in the appendix (RdYlGn from matplotlib).

B.1. WordCount

B.1.1. hibench.scale.profile: huge

[E1] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.1.

Metric	Stages		
	0	1	\emptyset
EGlobal	1.57	0.19	1.57
EParallel	0.98	0.23	0.97
ELoadBalance	0.98	0.25	0.97
ECommunication	1.0	0.93	1.0
ESerialization	1.0	0.95	1.0
ETransfer	1.0	0.98	1.0
ECompScaling	1.64	0.96	1.64
ETaskSuccess	1.0	1.0	1.0
ETask	0.98	0.85	0.98
EGC	1.0	0.99	1.0
ETaskDeserialization	0.98	0.87	0.98
EResultSerialization	1.0	1.0	1.0
ETaskShuffle	1.0	1.0	1.0
ETaskShuffleFetch	1.0	1.0	1.0
ETaskShuffleWrite	1.0	1.0	1.0
EInputBalance	0.69	0.95	0.69
ETaskLocality	0.33	0.67	0.33
EMemorySpill	1.0	1.0	1.0
Stage Duration (s)	283.42	0.36	-
Ref. Stage Duration (s)	465.05	0.35	-
$ T_s $	16	4	15.98
$ K_s $	124	4	123.85
Minor GC Count	461	0	460.41
Major GC Count	12	0	11.98

[E2] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
EGlobal	1.64	0.18	1.63
EParallel	0.97	0.22	0.97
ELoadBalance	0.97	0.24	0.97
ECommunication	1.0	0.93	1.0
ESerialization	1.0	0.95	1.0
ETransfer	1.0	0.98	1.0
ECompScaling	1.73	0.95	1.72
ETaskSuccess	1.0	1.0	1.0
ETask	0.98	0.86	0.98
EGC	1.0	1.0	1.0
ETaskDeserialization	0.98	0.88	0.98
EResultSerialization	1.0	1.0	1.0
ETaskShuffle	1.0	1.0	1.0
ETaskShuffleFetch	1.0	1.0	1.0
ETaskShuffleWrite	1.0	1.0	1.0
EInputBalance	0.67	0.95	0.67
ETaskLocality	0.33	0.67	0.33
EMemorySpill	1.0	1.0	1.0
Stage Duration (s)	282.55	0.38	-
Ref. Stage Duration (s)	487.56	0.36	-
$ T_s $	16	4	15.98
$ K_s $	124	4	123.84
Minor GC Count	448	0	447.4
Major GC Count	12	0	11.98

[E3] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.9.

Metric	Stages		
	0	1	\emptyset
EGlobal	1.58	0.17	1.57
EParallel	0.97	0.23	0.97
ELoadBalance	0.97	0.24	0.97
ECommunication	1.0	0.94	1.0
ESerialization	1.0	0.96	1.0
ETransfer	1.0	0.98	1.0
ECompScaling	1.66	0.89	1.66
ETaskSuccess	1.0	1.0	1.0
ETask	0.98	0.87	0.98
EGC	1.0	1.0	1.0
ETaskDeserialization	0.98	0.88	0.98
EResultSerialization	1.0	1.0	1.0
ETaskShuffle	1.0	1.0	1.0
ETaskShuffleFetch	1.0	1.0	1.0
ETaskShuffleWrite	1.0	1.0	1.0
EInputBalance	0.61	0.95	0.61
ETaskLocality	0.33	0.67	0.33
EMemorySpill	1.0	1.0	1.0
Stage Duration (s)	284.01	0.37	-
Ref. Stage Duration (s)	472.11	0.33	-
$ T_s $	16	4	15.98
$ K_s $	124	4	123.84
Minor GC Count	430	0	429.44
Major GC Count	12	0	11.98

B. Experimental Results

[E4] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	1.64	0.13	1.63
E _{Parallel}	0.97	0.33	0.97
E _{LoadBalance}	0.97	0.34	0.97
E _{Communication}	1.0	0.97	1.0
E _{Serialization}	1.0	0.98	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	1.72	0.45	1.72
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.98	0.89	0.98
E _{GC}	1.0	0.99	1.0
E _{TaskDeserialization}	0.98	0.91	0.98
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.75	0.91	0.75
E _{TaskLocality}	0.33	0.67	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	282.95	0.8	-
Ref. Stage Duration (s)	487.56	0.36	-
$ T_s $	16	8	15.98
$ K_s $	128	8	127.66
Minor GC Count	465	0	463.7
Major GC Count	12	0	11.97

[E5] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	1.64	0.25	1.63
E _{Parallel}	0.97	0.65	0.97
E _{LoadBalance}	0.97	0.67	0.97
E _{Communication}	1.0	0.97	1.0
E _{Serialization}	1.0	0.98	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	1.72	0.43	1.72
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.98	0.88	0.98
E _{GC}	1.0	1.0	1.0
E _{TaskDeserialization}	0.98	0.9	0.98
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	0.99	1.0
E _{TaskShuffleFetch}	1.0	0.99	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.69	0.84	0.69
E _{TaskLocality}	0.33	0.67	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	283.5	0.83	-
Ref. Stage Duration (s)	487.56	0.36	-
$ T_s $	16	16	16.0
$ K_s $	128	16	127.67
Minor GC Count	444	95	442.98
Major GC Count	12	3	11.97

[E6] Nodes: 8, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	2.51	0.07	2.5
E _{Parallel}	0.54	0.11	0.54
E _{LoadBalance}	0.56	0.11	0.55
E _{Communication}	0.97	0.94	0.97
E _{Serialization}	0.97	0.95	0.97
E _{Transfer}	1.0	0.98	1.0
E _{CompScaling}	5.19	0.82	5.17
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.9	0.85	0.9
E _{GC}	0.99	1.0	0.99
E _{TaskDeserialization}	0.91	0.87	0.91
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.47	0.91	0.48
E _{TaskLocality}	0.56	0.67	0.56
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	93.97	0.44	-
Ref. Stage Duration (s)	487.56	0.36	-
$ T_s $	64	8	63.74
$ K_s $	128	8	127.44
Minor GC Count	390	0	388.18
Major GC Count	24	0	23.89

[E7] Nodes: 16, executors: 16, cores: 16, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	0.53	0.05	0.53
E _{Parallel}	0.26	0.06	0.26
E _{LoadBalance}	0.27	0.06	0.27
E _{Communication}	0.98	0.93	0.98
E _{Serialization}	0.98	0.95	0.98
E _{Transfer}	1.0	0.98	1.0
E _{CompScaling}	2.2	0.98	2.2
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.92	0.85	0.92
E _{GC}	1.0	1.0	1.0
E _{TaskDeserialization}	0.92	0.88	0.92
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.95	0.84	0.95
E _{TaskLocality}	0.42	1.0	0.42
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	221.59	0.37	-
Ref. Stage Duration (s)	487.56	0.36	-
$ T_s $	128	16	127.81
$ K_s $	128	16	127.81
Minor GC Count	419	0	418.31
Major GC Count	42	0	41.93

B.1.2. hibench.scale.profile: gigantic

[E8] Nodes: 4, executors: 4,
cores: 4, degree of parallelism:
4, memory fraction: 0.1.

Metric	Stages		
	0	1	∅
E _{Global}	7.05	0.3	7.05
E _{Parallel}	0.96	0.23	0.96
E _{LoadBalance}	1.0	0.24	1.0
E _{Communication}	0.97	0.96	0.97
E _{Serialization}	0.97	0.97	0.97
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	7.44	1.43	7.43
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.98	0.91	0.98
E _{GC}	0.99	0.99	0.99
E _{TaskDeserialization}	0.99	0.93	0.99
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.99	0.95	0.99
E _{TaskLocality}	0.67	1.0	0.67
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	659.33	0.6	-
Ref. Stage Duration (s)	4903.01	0.85	-
T _s	16	4	15.99
K _s	1216	4	1214.9
Minor GC Count	4071	0	4067.32
Major GC Count	12	0	11.99

[E9] Nodes: 4, executors: 4,
cores: 4, degree of parallelism:
4, memory fraction: 0.6.

Metric	Stages		
	0	1	∅
E _{Global}	1.73	0.31	1.73
E _{Parallel}	1.0	0.23	1.0
E _{LoadBalance}	1.0	0.24	1.0
E _{Communication}	1.0	0.96	1.0
E _{Serialization}	1.0	0.97	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	1.75	1.49	1.75
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.99	0.9	0.99
E _{GC}	0.99	1.0	0.99
E _{TaskDeserialization}	1.0	0.92	1.0
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.87	0.95	0.87
E _{TaskLocality}	0.33	1.0	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	2781.77	0.58	-
Ref. Stage Duration (s)	4857.12	0.86	-
T _s	16	4	16.0
K _s	1216	4	1215.75
Minor GC Count	12295	0	12292.45
Major GC Count	12	0	12.0

[E10] Nodes: 4, executors: 4,
cores: 4, degree of parallelism:
4, memory fraction: 0.9.

Metric	Stages		
	0	1	∅
E _{Global}	1.73	0.27	1.73
E _{Parallel}	1.0	0.22	1.0
E _{LoadBalance}	1.0	0.23	1.0
E _{Communication}	1.0	0.97	1.0
E _{Serialization}	1.0	0.97	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	1.75	1.35	1.75
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.99	0.91	0.99
E _{GC}	0.99	0.99	0.99
E _{TaskDeserialization}	1.0	0.92	1.0
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.82	0.95	0.82
E _{TaskLocality}	0.33	1.0	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	2781.75	0.64	-
Ref. Stage Duration (s)	4878.0	0.86	-
T _s	16	4	16.0
K _s	1216	4	1215.72
Minor GC Count	11478	2977	11476.05
Major GC Count	12	3	12.0

[E11] Nodes: 4, executors: 4, cores: 4, degree of
parallelism: 8, memory fraction: 0.6.

Metric	Stages		
	0	1	∅
E _{Global}	6.81	0.48	6.8
E _{Parallel}	0.99	0.45	0.99
E _{LoadBalance}	0.99	0.47	0.99
E _{Communication}	1.0	0.97	1.0
E _{Serialization}	1.0	0.98	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	7.32	1.14	7.31
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.94	0.92	0.94
E _{GC}	0.99	0.99	0.99
E _{TaskDeserialization}	0.95	0.93	0.95
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.97	0.91	0.97
E _{TaskLocality}	0.67	1.0	0.67
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	663.52	0.75	-
Ref. Stage Duration (s)	4857.12	0.86	-
T _s	16	8	15.99
K _s	1216	8	1214.63
Minor GC Count	3945	0	3940.53
Major GC Count	12	0	11.99

[E12] Nodes: 4, executors: 4, cores: 4, degree of
parallelism: 16, memory fraction: 0.6.

Metric	Stages		
	0	1	∅
E _{Global}	1.71	0.48	1.71
E _{Parallel}	1.0	0.73	1.0
E _{LoadBalance}	1.0	0.74	1.0
E _{Communication}	1.0	0.98	1.0
E _{Serialization}	1.0	0.99	1.0
E _{Transfer}	1.0	1.0	1.0
E _{CompScaling}	1.74	0.73	1.74
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.99	0.91	0.99
E _{GC}	0.99	0.99	0.99
E _{TaskDeserialization}	1.0	0.93	1.0
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.86	0.84	0.86
E _{TaskLocality}	0.33	1.0	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	2796.95	1.18	-
Ref. Stage Duration (s)	4857.12	0.86	-
T _s	16	16	16.0
K _s	1216	16	1215.49
Minor GC Count	13261	6716	13258.24
Major GC Count	12	6	12.0

B. Experimental Results

[E13] Nodes: 8, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	1.71	0.17	1.71
E _{Parallel}	0.99	0.11	0.99
E _{LoadBalance}	0.99	0.12	0.99
E _{Communication}	1.0	0.95	1.0
E _{Serialization}	1.0	0.97	1.0
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	1.75	1.7	1.75
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.99	0.89	0.99
E _{GC}	1.0	1.0	1.0
E _{TaskDeserialization}	1.0	0.91	1.0
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.56	0.91	0.56
E _{TaskLocality}	0.33	1.0	0.33
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	2782.15	0.51	-
Ref. Stage Duration (s)	4857.12	0.86	-
$ T_s $	64	8	63.99
$ K_s $	1216	8	1215.78
Minor GC Count	18275	2590	18272.14
Major GC Count	24	3	24.0

[E14] Nodes: 16, executors: 16, cores: 16, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages		
	0	1	\emptyset
E _{Global}	5.31	0.03	5.3
E _{Parallel}	0.77	0.03	0.77
E _{LoadBalance}	0.78	0.03	0.78
E _{Communication}	0.99	0.97	0.99
E _{Serialization}	0.99	0.98	0.99
E _{Transfer}	1.0	0.99	1.0
E _{CompScaling}	7.01	1.04	7.0
E _{TaskSuccess}	1.0	1.0	1.0
E _{Task}	0.98	0.88	0.98
E _{GC}	1.0	0.99	1.0
E _{TaskDeserialization}	0.98	0.91	0.98
E _{ResultSerialization}	1.0	1.0	1.0
E _{TaskShuffle}	1.0	0.99	1.0
E _{TaskShuffleFetch}	1.0	0.99	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0
E _{InputBalance}	0.16	0.84	0.16
E _{TaskLocality}	0.57	1.0	0.57
E _{MemorySpill}	1.0	1.0	1.0
Stage Duration (s)	693.11	0.83	-
Ref. Stage Duration (s)	4857.12	0.86	-
$ T_s $	256	16	255.71
$ K_s $	1216	16	1214.57
Minor GC Count	4112	0	4107.1
Major GC Count	48	0	47.94

B.2. PageRank (hibench.scale.profile: huge)

[E15] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.1.

Metric	Stages						
	0	1	2	3	4	5	\emptyset
E _{Global}	4.73	6.01	1.37	1.67	1.48	0.92	2.43
E _{Parallel}	0.67	0.69	0.19	0.22	0.21	0.23	0.32
E _{LoadBalance}	0.67	0.69	0.19	0.22	0.21	0.23	0.32
E _{Communication}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Serialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Transfer}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{CompScaling}	8.65	9.38	8.3	8.73	8.16	4.35	8.53
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.81	0.93	0.88	0.87	0.85	0.92	0.87
E _{GC}	0.85	0.93	0.88	0.87	0.85	0.93	0.88
E _{TaskDeserialization}	0.96	1.0	1.0	1.0	1.0	0.99	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.93	1.0	1.0	1.0	1.0	1.0	0.99
E _{TaskLocality}	0.33	0.67	0.67	0.67	0.67	0.67	0.64
E _{MemorySpill}	0.0	0.0	0.0	0.0	0.0	1.0	0.0
Stage Duration (s)	136.26	206.77	517.5	279.31	298.29	4.69	-
Ref. Stage Duration (s)	1178.17	1939.54	4296.31	2437.57	2434.22	20.39	-
$ T_s $	12	12	4	4	4	4	5.9
$ K_s $	12	12	4	4	4	4	5.9
Minor GC Count	420	789	1266	1562	1869	973	1298.76
Major GC Count	24	35	50	63	79	36	53.86

B.2. PageRank (*hibench.scale.profile: huge*)

[E16] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.6.

Metric	Stages						∅
	0	1	2	3	4	5	
E _{Global}	2.79	2.54	0.79	0.7	0.71	0.2	1.47
E _{Parallel}	0.69	0.67	0.23	0.23	0.24	0.21	0.4
E _{LoadBalance}	0.69	0.67	0.23	0.23	0.24	0.21	0.4
E _{Communication}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Serialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Transfer}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{CompScaling}	7.31	6.31	5.12	4.72	4.65	1.47	5.56
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.56	0.6	0.67	0.64	0.64	0.66	0.63
E _{GC}	0.57	0.6	0.67	0.64	0.64	0.67	0.63
E _{TaskDeserialization}	0.98	1.0	1.0	1.0	1.0	1.0	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.93	1.0	1.0	1.0	1.0	1.0	0.99
E _{TaskLocality}	0.33	0.67	0.67	0.67	0.67	0.67	0.61
E _{MemorySpill}	0.0	0.0	0.0	0.0	0.0	1.0	0.01
Stage Duration (s)	222.4	288.21	338.09	239.96	238.99	8.45	-
Ref. Stage Duration (s)	1625.53	1818.97	1729.51	1132.87	1111.92	12.44	-
T _s	12	12	4	4	4	4	7.06
K _s	12	12	4	4	4	4	7.06
Minor GC Count	441	801	1163	1410	1660	1253	1098.56
Major GC Count	41	77	133	166	197	148	123.08

[E17] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.9.

The application crashed while computing the first stage due to running out of memory, below results come from the first of two tries. Contrary to the configuration, Spark executed this application with 9 executors.

Metric	Stages	
	0	∅
E _{Global}	0.01	0.01
E _{Parallel}	0.06	0.06
E _{LoadBalance}	0.21	0.21
E _{Communication}	0.29	0.29
E _{Serialization}	0.5	0.5
E _{Transfer}	0.59	0.59
E _{CompScaling}	2.0	2.0
E _{TaskSuccess}	0.26	0.26
E _{Task}	0.18	0.18
E _{GC}	0.83	0.83
E _{TaskDeserialization}	1.0	1.0
E _{ResultSerialization}	1.0	1.0
E _{TaskShuffle}	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0
E _{InputBalance}	0.56	0.56
E _{TaskLocality}	0.51	0.51
E _{MemorySpill}	0.0	0.0
Stage Duration (s)	1780.36	-
Ref. Stage Duration (s)	3566.74	-
T _s	13	13.0
K _s	31	31.0
Minor GC Count	175	175.0
Major GC Count	77	77.0

B. Experimental Results

[E18] Nodes: 8, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages						\emptyset
	0	1	2	3	4	5	
E _{Global}	0.71	1.26	1.09	0.84	0.88	0.13	0.95
E _{Parallel}	0.16	0.21	0.11	0.11	0.12	0.11	0.15
E _{LoadBalance}	0.16	0.21	0.11	0.11	0.12	0.11	0.15
E _{Communication}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Serialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Transfer}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{CompScaling}	7.63	10.77	13.29	11.11	11.41	1.86	10.34
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.58	0.55	0.76	0.68	0.67	0.64	0.63
E _{GC}	0.61	0.55	0.76	0.68	0.67	0.65	0.64
E _{TaskDeserialization}	0.95	1.0	1.0	1.0	1.0	0.99	0.98
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	0.98	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	0.98	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.7	1.0	1.0	1.0	1.0	1.0	0.91
E _{TaskLocality}	0.33	1.0	1.0	1.0	1.0	1.0	0.8
E _{MemorySpill}	0.04	0.0	1.0	1.0	1.0	1.0	0.48
Stage Duration (s)	213.18	168.89	130.18	101.99	97.47	6.68	-
Ref. Stage Duration (s)	1625.53	1818.97	1729.51	1132.87	1111.92	12.44	-
$ T_s $	16	16	8	8	8	8	12.25
$ K_s $	16	16	8	8	8	8	12.25
Minor GC Count	471	867	1268	1497	1727	883	1028.43
Major GC Count	51	106	129	145	161	80	106.6

[E19] Nodes: 16, executors: 16, cores: 16, degree of parallelism: 16, memory fraction: 0.6.

Contrary to the configuration, Spark executed this application with 10 executors.

Metric	Stages						\emptyset
	0	1	2	3	4	5	
E _{Global}	0.4	0.4	0.23	0.18	0.19	0.23	0.29
E _{Parallel}	0.1	0.1	0.1	0.1	0.1	0.09	0.1
E _{LoadBalance}	0.1	0.1	0.1	0.1	0.1	0.09	0.1
E _{Communication}	1.0	1.0	1.0	1.0	1.0	0.99	1.0
E _{Serialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Transfer}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{CompScaling}	9.12	5.93	7.52	6.19	6.32	2.86	6.89
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.44	0.69	0.32	0.3	0.3	0.85	0.44
E _{GC}	0.45	0.69	0.32	0.3	0.3	0.87	0.44
E _{TaskDeserialization}	0.98	1.0	1.0	1.0	1.0	0.99	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.99	1.0	0.98	0.98	0.98	0.98	0.99
E _{TaskLocality}	0.67	0.67	0.67	0.67	0.67	0.67	0.67
E _{MemorySpill}	0.0	0.0	0.0	0.0	0.0	1.0	0.0
Stage Duration (s)	178.31	306.75	229.99	182.93	175.9	4.35	-
Ref. Stage Duration (s)	1625.53	1818.97	1729.51	1132.87	1111.92	12.44	-
$ T_s $	16	16	16	16	16	16	16.0
$ K_s $	16	16	16	16	16	16	16.0
Minor GC Count	452	883	1314	1661	2032	1019	1223.64
Major GC Count	46	82	136	175	215	106	125.14

B.3. K-means

B.3.1. hibench.scale.profile: huge

[E20] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.1.

The simulated thread scheduler had to perform additional waiting of max. 2ms to schedule all tasks.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.53	0.03	2.33	8.48	10.33	0.07	10.1	0.1	9.98	0.09	10.11	0.06	10.13	0.08	1.86	1.69	2.57
E _{Parallel}	0.96	0.06	0.94	0.88	0.88	0.18	0.84	0.17	0.82	0.16	0.83	0.15	0.83	0.15	0.97	0.7	0.94
E _{LoadBalance}	0.96	0.06	0.95	0.89	0.88	0.2	0.84	0.22	0.82	0.23	0.83	0.19	0.84	0.22	0.97	0.9	0.95
E _{Communication}	1.0	0.91	0.99	0.99	1.0	0.91	1.0	0.77	1.0	0.7	1.0	0.77	1.0	0.7	1.0	0.78	0.99
E _{Serialization}	1.0	0.96	0.99	1.0	1.0	0.96	1.0	0.85	1.0	0.84	1.0	0.85	1.0	0.8	1.0	0.95	1.0
E _{Transfer}	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.9	1.0	0.83	1.0	0.9	1.0	0.87	1.0	0.82	1.0
E _{CompScaling}	1.66	0.85	2.52	10.81	12.12	0.48	12.26	1.03	12.39	1.03	12.34	0.83	12.33	1.04	1.95	5.03	2.87
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.96	0.71	0.98	0.89	0.97	0.8	0.98	0.58	0.98	0.54	0.99	0.52	0.99	0.53	0.98	0.48	0.98
E _{GC}	0.99	1.0	0.98	0.9	0.98	1.0	0.99	1.0	0.99	1.0	0.99	1.0	0.99	1.0	0.99	1.0	0.98
E _{TaskDeserialization}	0.97	0.75	1.0	0.99	0.99	0.85	1.0	0.68	1.0	0.65	1.0	0.62	1.0	0.68	1.0	0.67	0.99
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.66	1.0	0.67	0.77	0.93	0.71	0.93	0.79	0.93	0.79	0.93	0.78	0.93	0.78	0.66	0.62	0.68
E _{TaskLocality}	0.33	0.67	0.34	1.0	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	0.33	0.98	0.39
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	176.09	0.16	340.42	7.52	9.34	0.1	9.12	0.03	8.99	0.03	8.89	0.03	8.87	0.02	169.74	0.16	-
Ref. Stage Duration (s)	291.9	0.13	858.78	81.33	113.17	0.05	111.88	0.03	111.45	0.03	109.76	0.03	109.33	0.02	331.57	0.81	-
[T _s]	16	1	16	16	16	4	16	4	16	4	16	4	16	4	16	16	15.99
[K _s]	75	1	75	75	75	4	75	4	75	4	75	4	75	4	75	200	74.99
Minor GC Count	549	0	1853	1435	1968	0	1459	0	960	0	1541	0	1646	0	2694	0	1709.44
Major GC Count	12	0	18	14	18	0	13	0	8	0	13	0	14	0	21	0	16.91

[E21] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	6.56	0.03	3.74	3.21	10.21	0.06	8.37	0.06	9.58	0.06	7.89	0.09	7.97	0.06	7.97	1.74	5.87
E _{Parallel}	0.95	0.06	0.91	0.86	0.83	0.18	0.8	0.17	0.81	0.14	0.79	0.16	0.81	0.15	0.93	0.68	0.89
E _{LoadBalance}	0.95	0.06	0.94	0.86	0.84	0.21	0.8	0.24	0.82	0.2	0.79	0.23	0.82	0.21	0.93	0.85	0.91
E _{Communication}	1.0	0.92	0.97	1.0	0.99	0.86	1.0	0.73	0.99	0.71	1.0	0.71	1.0	0.7	1.0	0.81	0.98
E _{Serialization}	1.0	0.96	1.0	1.0	1.0	0.9	1.0	0.81	1.0	0.83	1.0	0.85	1.0	0.86	1.0	0.95	1.0
E _{Transfer}	1.0	0.96	0.97	1.0	0.99	0.96	1.0	0.9	0.99	0.86	1.0	0.83	1.0	0.81	1.0	0.85	0.99
E _{CompScaling}	8.56	0.82	7.16	6.99	13.49	0.45	12.26	0.72	13.09	0.77	11.57	1.04	11.64	0.81	9.83	5.21	8.88
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.81	0.67	0.58	0.54	0.91	0.77	0.86	0.5	0.9	0.52	0.86	0.55	0.84	0.53	0.87	0.49	0.72
E _{GC}	0.95	0.92	0.58	0.54	0.92	1.0	0.86	0.91	0.91	1.0	0.87	1.0	0.85	1.0	0.87	1.0	0.75
E _{TaskDeserialization}	0.85	0.76	1.0	1.0	0.99	0.82	1.0	0.75	1.0	0.63	1.0	0.71	1.0	0.67	1.0	0.68	0.97
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.84	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.84	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.93	1.0	0.94	0.66	0.91	0.69	0.91	0.79	0.91	0.79	0.91	0.79	0.91	0.79	0.93	0.62	0.91
E _{TaskLocality}	0.67	0.67	0.78	1.0	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	0.67	0.98	0.8
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	32.66	0.17	86.68	12.25	8.36	0.1	9.08	0.04	8.38	0.04	9.34	0.02	9.36	0.03	33.09	0.15	-
Ref. Stage Duration (s)	279.46	0.14	620.9	85.68	112.83	0.05	111.35	0.03	109.76	0.03	108.09	0.03	108.92	0.02	325.13	0.8	-
[T _s]	16	1	16	16	16	4	16	4	16	4	16	4	16	4	16	16	15.98
[K _s]	75	1	75	75	75	4	75	4	75	4	75	4	75	4	75	200	74.96
Minor GC Count	338	0	1028	1078	864	0	1179	0	625	0	653	0	677	0	1964	0	1019.98
Major GC Count	20	0	66	73	56	0	76	0	39	0	40	0	41	0	88	0	59.23

B. Experimental Results

[E22] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.9.

Contrary to the configuration, Spark executed this application with 6 executors. Only 5 executors were actively used in stage 10, one of which had access to double the amount of configured cores.

Metric	Stages																∅
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
E _{Global}	1.01	0.03	0.14	0.06	0.06	0.0	0.27	0.07	0.27	0.0	0.09	0.04	0.08	0.04	1.57	0.11	0.32
E _{Parallel}	0.63	0.04	0.61	0.31	0.39	0.05	0.4	0.11	0.43	0.16	0.43	0.1	0.29	0.1	0.63	0.15	0.51
E _{LoadBalance}	0.63	0.04	0.61	0.32	0.41	0.05	0.41	0.15	0.43	0.17	0.44	0.13	0.31	0.14	0.63	0.52	0.52
E _{Communication}	1.0	0.91	1.0	0.98	0.96	0.99	1.0	0.71	1.0	0.98	0.98	0.76	0.95	0.67	1.0	0.28	0.99
E _{Serialization}	1.0	0.95	1.0	1.0	1.0	1.0	1.0	0.8	1.0	0.98	1.0	0.85	1.0	0.79	1.0	0.94	1.0
E _{Transfer}	1.0	0.95	1.0	0.98	0.96	1.0	1.0	0.89	1.0	1.0	0.98	0.89	0.95	0.85	1.0	0.3	0.99
E _{CompScaling}	1.87	0.91	1.95	0.64	0.81	0.02	2.95	1.0	2.88	0.02	0.67	0.66	1.05	0.73	2.85	2.09	1.62
E _{TaskSuccess}	1.0	1.0	1.0	0.95	1.0	1.0	1.0	1.0	1.0	1.0	0.95	1.0	1.0	1.0	1.0	1.0	0.99
E _{Task}	0.86	0.74	0.12	0.32	0.19	0.14	0.23	0.61	0.22	0.19	0.32	0.54	0.24	0.56	0.87	0.37	0.32
E _{GC}	0.88	1.0	0.12	0.37	0.19	0.14	0.23	1.0	0.22	0.75	0.36	1.0	0.24	1.0	0.88	0.98	0.33
E _{TaskDeserialization}	0.97	0.78	1.0	1.0	0.99	0.98	1.0	0.72	1.0	1.0	0.99	0.76	1.0	0.71	1.0	0.82	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.26	1.0	0.84	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.26	1.0	0.84	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.58	1.0	0.73	0.54	0.58	0.71	0.66	0.79	0.67	0.79	0.39	0.79	0.53	0.79	0.66	0.62	0.62
E _{TaskLocality}	0.33	0.67	0.68	0.97	0.88	0.67	1.0	0.67	1.0	0.67	0.85	0.67	0.91	0.67	0.4	0.98	0.72
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	178.19	0.14	759.43	200.92	175.79	1.91	46.82	0.03	46.32	1.92	205.62	0.04	128.85	0.03	159.19	0.62	-
Ref. Stage Duration (s)	332.33	0.12	1477.88	128.1	143.07	0.04	138.3	0.03	133.42	0.03	137.46	0.03	135.75	0.02	454.19	1.3	-
T _s	16	1	16	16	16	4	16	4	16	4	24	4	16	4	16	16	16.84
K _s	75	1	75	79	75	4	75	4	75	4	87	4	75	4	75	200	76.61
Minor GC Count	395	0	659	692	608	0	569	0	578	377	716	0	599	0	1058	0	663.09
Major GC Count	622	0	2292	2540	2176	0	2242	0	2321	1625	2640	0	1971	0	2531	0	2182.59

[E23] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages																∅
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
E _{Global}	1.46	0.04	2.85	2.98	7.5	0.23	9.54	0.24	10.76	0.22	11.02	0.19	10.27	0.17	1.79	1.02	2.61
E _{Parallel}	0.96	0.06	0.92	0.82	0.84	0.67	0.83	0.51	0.84	0.5	0.86	0.55	0.84	0.45	0.96	0.58	0.93
E _{LoadBalance}	0.96	0.06	0.94	0.83	0.84	0.74	0.84	0.73	0.85	0.76	0.86	0.82	0.84	0.67	0.96	0.87	0.94
E _{Communication}	1.0	0.9	0.98	1.0	1.0	0.9	1.0	0.69	1.0	0.66	1.0	0.67	1.0	0.67	1.0	0.67	0.99
E _{Serialization}	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.83	1.0	0.76	1.0	0.76	1.0	0.82	1.0	0.94	1.0
E _{Transfer}	1.0	0.95	0.98	1.0	1.0	0.95	1.0	0.83	1.0	0.86	1.0	0.88	1.0	0.81	1.0	0.71	0.99
E _{CompScaling}	1.59	0.89	3.95	6.65	11.22	0.45	12.9	1.0	13.76	0.93	13.83	0.76	13.47	0.81	1.91	4.0	3.33
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.96	0.71	0.79	0.54	0.8	0.75	0.89	0.48	0.93	0.46	0.93	0.46	0.91	0.46	0.98	0.44	0.9
E _{GC}	0.99	1.0	0.79	0.55	0.82	1.0	0.89	1.0	0.93	1.0	0.94	1.0	0.92	1.0	0.98	1.0	0.91
E _{TaskDeserialization}	0.97	0.75	1.0	1.0	0.98	0.8	1.0	0.64	1.0	0.65	1.0	0.59	1.0	0.63	1.0	0.72	0.99
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.97	1.0	0.93	1.0	0.97	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.97	1.0	0.93	1.0	0.97	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.68	1.0	0.95	0.65	0.77	0.43	0.78	0.62	0.77	0.62	0.77	0.62	0.77	0.62	0.68	0.62	0.76
E _{TaskLocality}	0.33	0.67	0.56	1.0	1.0	0.79	1.0	0.79	1.0	0.79	1.0	0.79	1.0	0.79	0.33	0.98	0.46
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	175.87	0.15	157.04	12.89	10.06	0.1	8.63	0.03	7.98	0.03	7.82	0.03	8.09	0.03	170.15	0.2	-
Ref. Stage Duration (s)	279.46	0.14	620.9	85.68	112.83	0.05	111.35	0.03	109.76	0.03	108.09	0.03	108.92	0.02	325.13	0.8	-
T _s	16	1	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16.0
K _s	75	1	75	75	75	16	75	16	75	16	75	16	75	16	75	200	75.0
Minor GC Count	605	0	1488	1538	1600	0	1173	0	824	0	933	0	1788	0	2542	0	1514.89
Major GC Count	16	0	56	63	68	0	51	0	34	0	35	0	69	0	77	0	49.63

[E24] Nodes: 8, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.25	0.01	4.66	4.28	9.61	0.03	16.33	0.01	14.85	0.03	8.95	0.03	18.28	0.02	1.65	2.32	2.75
E _{Parallel}	0.51	0.01	0.46	0.39	0.53	0.09	0.5	0.04	0.46	0.07	0.36	0.07	0.51	0.06	0.5	0.72	0.49
E _{LoadBalance}	0.53	0.02	0.46	0.4	0.54	0.1	0.5	0.04	0.46	0.1	0.37	0.1	0.51	0.09	0.52	0.85	0.51
E _{Communication}	0.96	0.9	1.0	1.0	0.99	0.88	0.99	0.88	0.99	0.74	1.0	0.69	0.99	0.7	0.96	0.85	0.97
E _{Serialization}	0.97	0.95	1.0	1.0	1.0	0.93	1.0	0.95	1.0	0.87	1.0	0.8	1.0	0.8	0.96	0.94	0.97
E _{Transfer}	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.93	1.0	0.86	1.0	0.86	1.0	0.88	1.0	0.9	1.0
E _{CompScaling}	2.77	0.76	14.6	22.05	29.16	0.4	37.74	0.34	36.19	0.77	28.02	0.86	39.05	0.55	3.43	6.69	7.19
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.88	0.77	0.7	0.49	0.62	0.78	0.87	0.63	0.9	0.56	0.88	0.53	0.92	0.51	0.97	0.48	0.87
E _{GC}	0.98	1.0	0.71	0.51	0.72	1.0	0.88	1.0	0.91	1.0	0.89	1.0	0.93	1.0	0.97	0.99	0.92
E _{TaskDeserialization}	0.9	0.8	0.99	0.98	0.87	0.83	0.99	0.76	0.99	0.69	0.99	0.68	0.99	0.65	1.0	0.59	0.96
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.58	1.0	0.67	0.55	0.55	0.5	0.55	0.62	0.55	0.62	0.55	0.62	0.59	0.62	0.58	0.16	0.59
E _{TaskLocality}	0.49	1.0	0.83	1.0	1.0	0.71	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	0.49	0.98	0.59
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	100.86	0.18	42.53	3.88	3.87	0.12	2.95	0.09	3.03	0.04	3.86	0.03	2.79	0.04	94.77	0.12	-
Ref. Stage Duration (s)	279.46	0.14	620.9	85.68	112.83	0.05	111.35	0.03	109.76	0.03	108.09	0.03	108.92	0.02	325.13	0.8	-
T _s	64	1	64	64	64	8	64	8	64	8	64	8	64	8	64	64	63.89
K _s	75	1	75	75	75	8	75	8	75	8	75	8	75	8	75	200	74.93
Minor GC Count	508	0	1128	422	170	0	325	158	297	0	376	0	351	0	1970	0	1128.72
Major GC Count	28	0	56	24	8	0	17	7	18	0	18	0	17	0	75	0	48.85

[E25] Nodes: 16, executors: 16, cores: 16, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	0.27	0.0	2.35	1.29	5.93	0.0	5.98	0.0	6.22	0.0	5.5	0.0	5.77	0.01	0.13	0.63	0.61
E _{Parallel}	0.13	0.0	0.16	0.07	0.16	0.01	0.14	0.01	0.15	0.0	0.13	0.02	0.12	0.03	0.06	0.34	0.09
E _{LoadBalance}	0.13	0.0	0.16	0.07	0.16	0.01	0.14	0.02	0.15	0.0	0.13	0.02	0.12	0.04	0.06	0.41	0.09
E _{Communication}	0.96	0.91	1.0	0.99	0.99	1.0	0.99	0.87	0.99	1.0	0.99	0.9	0.99	0.77	0.97	0.83	0.97
E _{Serialization}	0.96	0.97	1.0	1.0	1.0	1.0	1.0	0.94	1.0	1.0	1.0	0.96	1.0	0.89	0.97	0.96	0.97
E _{Transfer}	1.0	0.95	1.0	1.0	0.99	1.0	1.0	0.93	0.99	1.0	1.0	0.94	1.0	0.87	1.0	0.87	1.0
E _{CompScaling}	2.51	0.73	22.31	29.57	47.95	0.0	50.77	0.26	51.89	0.0	48.58	0.24	50.85	0.71	2.89	3.7	6.31
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.81	1.0	0.93
E _{Task}	0.85	0.78	0.65	0.6	0.78	0.03	0.82	0.59	0.81	0.04	0.89	0.61	0.93	0.44	0.96	0.5	0.79
E _{GC}	0.98	1.0	0.66	0.64	0.85	1.0	0.85	1.0	0.83	1.0	0.95	1.0	0.94	1.0	0.96	1.0	0.94
E _{TaskDeserialization}	0.87	0.83	0.99	0.96	0.93	0.03	0.99	0.72	0.99	0.04	0.95	0.73	0.99	0.58	1.0	0.61	0.85
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.98	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.99	1.0	0.96	0.99	0.99	0.44	0.99	0.56	0.99	0.56	0.99	0.56	0.99	0.56	0.69	0.04	0.82
E _{TaskLocality}	0.48	0.67	0.91	1.0	1.0	0.92	1.0	0.92	1.0	0.92	1.0	0.92	1.0	0.92	0.63	0.99	0.65
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	111.17	0.18	27.83	2.9	2.35	19.38	2.19	0.11	2.12	10.06	2.23	0.1	2.14	0.03	112.35	0.22	-
Ref. Stage Duration (s)	279.46	0.14	620.9	85.68	112.83	0.05	111.35	0.03	109.76	0.03	108.09	0.03	108.92	0.02	325.13	0.8	-
T _s	75	1	75	75	75	16	75	16	75	16	75	16	75	16	75	200	62.65
K _s	75	1	75	75	75	16	75	16	75	16	75	16	75	16	93	200	75.96
Minor GC Count	436	0	884	0	622	0	195	0	0	0	64	0	273	0	1705	10	904.82
Major GC Count	38	0	50	0	31	0	15	0	0	0	5	0	10	0	90	3	53.72

B. Experimental Results

B.3.2. hibench.scale.profile: gigantic

[E26] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.1.
The simulated thread scheduler had to perform additional waiting of max. 2ms to schedule all tasks.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.63	0.04	2.7	11.41	11.34	0.07	11.47	0.11	11.76	0.08	11.74	0.11	11.28	0.09	1.99	1.65	2.86
E _{Parallel}	0.98	0.06	0.98	0.9	0.9	0.19	0.9	0.17	0.9	0.15	0.9	0.16	0.89	0.16	0.98	0.73	0.97
E _{LoadBalance}	0.98	0.06	0.98	0.91	0.91	0.2	0.9	0.21	0.9	0.21	0.91	0.23	0.89	0.23	0.98	0.9	0.98
E _{Communication}	1.0	0.9	1.0	1.0	1.0	0.93	1.0	0.79	1.0	0.7	1.0	0.71	1.0	0.71	1.0	0.81	1.0
E _{Serialization}	1.0	0.94	1.0	1.0	1.0	0.97	1.0	0.87	1.0	0.79	1.0	0.8	1.0	0.85	1.0	0.94	1.0
E _{Transfer}	1.0	0.96	1.0	1.0	1.0	0.95	1.0	0.91	1.0	0.88	1.0	0.89	1.0	0.83	1.0	0.87	1.0
E _{CompScaling}	1.7	0.92	2.8	12.95	12.79	0.48	12.92	1.0	13.26	0.82	13.12	1.11	12.85	1.0	2.05	4.53	3.04
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.98	0.73	0.98	0.98	0.98	0.81	0.99	0.64	0.99	0.63	0.99	0.62	0.99	0.55	0.99	0.5	0.98
E _{GC}	0.99	1.0	0.98	0.98	0.99	1.0	0.99	1.0	0.99	1.0	0.99	1.0	0.99	1.0	0.99	0.97	0.99
E _{TaskDeserialization}	0.99	0.76	1.0	1.0	0.99	0.85	1.0	0.74	1.0	0.74	1.0	0.73	1.0	0.68	1.0	0.66	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.77	1.0	0.84	0.93	0.93	0.69	0.93	0.77	0.93	0.78	0.93	0.78	0.93	0.78	0.77	0.62	0.82
E _{TaskLocality}	0.33	0.67	0.34	1.0	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	0.33	0.98	0.38
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	344.69	0.17	678.7	12.88	17.57	0.11	17.12	0.03	16.45	0.03	16.2	0.03	16.73	0.02	339.8	0.18	-
Ref. Stage Duration (s)	584.79	0.15	1903.75	166.92	224.73	0.05	221.22	0.03	218.16	0.03	212.5	0.03	214.89	0.02	696.19	0.8	-
$ T_s $	16	1	16	16	16	4	16	4	16	4	16	4	16	4	16	16	16.0
$ K_s $	150	1	150	150	150	4	150	4	150	4	150	4	150	4	150	200	149.97
Minor GC Count	1048	0	3500	3571	3650	0	3712	0	3766	0	3832	0	3899	0	4916	0	3265.99
Major GC Count	15	0	25	25	25	0	25	0	25	0	25	0	25	0	32	0	24.26

[E27] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.7	0.04	2.71	4.07	7.63	0.07	11.49	0.1	9.76	0.08	11.29	0.1	11.81	0.08	2.02	1.63	2.8
E _{Parallel}	0.98	0.06	0.97	0.83	0.84	0.18	0.91	0.16	0.88	0.15	0.91	0.17	0.93	0.16	0.98	0.7	0.97
E _{LoadBalance}	0.98	0.06	0.97	0.86	0.84	0.2	0.91	0.21	0.88	0.2	0.91	0.23	0.93	0.22	0.98	0.91	0.97
E _{Communication}	1.0	0.89	1.0	0.97	1.0	0.91	1.0	0.78	1.0	0.76	1.0	0.77	1.0	0.7	1.0	0.77	1.0
E _{Serialization}	1.0	0.94	1.0	1.0	1.0	0.96	1.0	0.88	1.0	0.86	1.0	0.88	1.0	0.83	1.0	0.94	1.0
E _{Transfer}	1.0	0.95	1.0	0.97	1.0	0.95	1.0	0.89	1.0	0.88	1.0	0.87	1.0	0.85	1.0	0.82	1.0
E _{CompScaling}	1.78	0.95	3.5	7.36	10.42	0.47	13.12	1.0	12.02	0.88	12.97	1.0	13.23	0.96	2.09	4.68	3.3
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.97	0.77	0.8	0.67	0.87	0.8	0.96	0.59	0.93	0.56	0.96	0.59	0.97	0.55	0.98	0.5	0.91
E _{GC}	0.99	1.0	0.8	0.68	0.88	1.0	0.96	1.0	0.93	1.0	0.96	1.0	0.97	1.0	0.98	1.0	0.92
E _{TaskDeserialization}	0.99	0.82	1.0	1.0	0.99	0.85	1.0	0.71	1.0	0.69	1.0	0.69	1.0	0.67	1.0	0.66	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.62	1.0	0.8	0.79	0.84	0.64	0.84	0.74	0.84	0.74	0.79	0.74	0.84	0.74	0.73	0.62	0.73
E _{TaskLocality}	0.33	0.67	0.44	1.0	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	1.0	0.67	0.33	0.98	0.43
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	344.78	0.15	360.93	21.37	20.23	0.12	16.27	0.04	17.73	0.03	16.38	0.03	15.99	0.03	339.64	0.17	-
Ref. Stage Duration (s)	614.12	0.14	1264.7	157.22	210.88	0.06	213.44	0.04	213.03	0.03	212.45	0.03	211.44	0.03	711.11	0.79	-
$ T_s $	16	1	16	16	16	4	16	4	16	4	16	4	16	4	16	16	16.0
$ K_s $	150	1	150	150	150	4	150	4	150	4	150	4	150	4	150	200	149.96
Minor GC Count	1035	0	3023	3132	3218	0	3309	0	3397	0	3484	0	3572	0	4937	0	3020.25
Major GC Count	19	0	110	122	122	0	122	0	123	0	123	0	123	0	137	0	91.87

[E28] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 4, memory fraction: 0.9.

Contrary to the configuration, Spark executed this application with 8 executors. Only 5 executors were actively used in stages 6 and 8, one of which had access to double the amount of configured cores.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	0.8	0.02	0.17	0.06	0.27	0.04	0.06	0.02	0.04	0.01	0.21	0.01	0.18	0.04	2.0	1.41	0.27
E _{Parallel}	0.49	0.03	0.44	0.42	0.35	0.1	0.39	0.07	0.43	0.06	0.31	0.05	0.31	0.08	0.42	0.32	0.43
E _{LoadBalance}	0.49	0.03	0.45	0.42	0.35	0.11	0.39	0.08	0.46	0.07	0.31	0.05	0.31	0.12	0.43	0.41	0.43
E _{Communication}	1.0	0.9	1.0	0.99	1.0	0.9	1.0	0.9	0.94	0.92	1.0	0.91	1.0	0.69	0.97	0.79	0.99
E _{Serialization}	1.0	0.95	1.0	1.0	1.0	0.93	1.0	0.95	0.99	0.97	1.0	0.95	1.0	0.79	1.0	0.91	1.0
E _{Transfer}	1.0	0.94	1.0	0.99	1.0	0.97	1.0	0.95	0.95	0.95	1.0	0.96	1.0	0.88	0.97	0.87	0.99
E _{CompScaling}	1.95	0.87	2.02	0.82	3.27	0.51	0.81	0.29	0.69	0.27	2.94	0.27	2.95	0.81	7.31	8.99	1.88
E _{TaskSuccess}	1.0	1.0	0.95	1.0	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98
E _{Task}	0.84	0.75	0.19	0.18	0.24	0.81	0.21	0.75	0.14	0.63	0.23	0.66	0.21	0.58	0.65	0.48	0.27
E _{GC}	0.85	1.0	0.21	0.18	0.24	1.0	0.21	1.0	0.14	1.0	0.23	1.0	0.21	1.0	0.65	1.0	0.28
E _{TaskDeserialization}	0.99	0.79	1.0	1.0	0.99	0.86	1.0	0.85	1.0	0.76	1.0	0.74	1.0	0.7	1.0	0.68	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.95	1.0	0.91	1.0	0.99	1.0	0.95	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.95	1.0	0.91	1.0	0.99	1.0	0.95	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.73	1.0	0.56	0.55	0.79	0.78	0.64	0.74	0.6	0.74	0.52	0.74	0.51	0.74	0.58	0.62	0.6
E _{TaskLocality}	0.33	0.67	0.5	0.83	1.0	0.67	0.9	0.67	0.88	0.67	1.0	0.67	0.99	0.67	0.57	0.98	0.65
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	346.69	0.16	1520.6	326.02	94.33	0.1	377.85	0.1	430.19	0.1	101.39	0.1	101.8	0.03	126.63	0.14	-
Ref. Stage Duration (s)	675.78	0.14	3067.24	267.44	308.91	0.05	306.6	0.03	295.55	0.03	298.14	0.03	300.59	0.03	926.15	1.22	-
T _s	16	1	24	16	16	4	24	4	24	4	16	4	16	4	16	16	21.44
K _s	150	1	158	150	150	4	231	4	246	4	150	4	150	4	150	200	174.51
Minor GC Count	398	0	998	900	948	0	1116	0	1047	0	769	0	778	0	1811	0	962.28
Major GC Count	2035	0	4913	4483	4644	0	5467	0	5070	0	3475	0	3720	0	4907	0	4575.02

[E29] Nodes: 4, executors: 4, cores: 4, degree of parallelism: 16, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.7	0.04	2.72	4.23	3.02	0.25	8.12	0.04	7.47	0.25	11.1	0.17	11.78	0.25	2.02	1.16	2.66
E _{Parallel}	0.98	0.06	0.97	0.88	0.61	0.64	0.83	0.62	0.81	0.54	0.89	0.5	0.93	0.51	0.98	0.61	0.96
E _{LoadBalance}	0.98	0.06	0.97	0.88	0.72	0.7	0.83	0.65	0.81	0.71	0.89	0.61	0.94	0.72	0.99	0.86	0.96
E _{Communication}	1.0	0.92	1.0	1.0	0.85	0.91	1.0	0.95	1.0	0.75	1.0	0.82	1.0	0.7	1.0	0.71	0.99
E _{Serialization}	1.0	0.95	1.0	1.0	1.0	0.94	1.0	0.97	1.0	0.86	1.0	0.92	1.0	0.83	1.0	0.94	1.0
E _{Transfer}	1.0	0.96	1.0	1.0	0.85	0.96	1.0	0.98	1.0	0.88	1.0	0.9	1.0	0.85	1.0	0.75	0.99
E _{CompScaling}	1.78	0.93	3.47	7.43	6.24	0.51	11.05	0.22	10.63	0.91	13.08	0.75	13.44	0.96	2.09	4.21	3.24
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.97	0.75	0.81	0.65	0.79	0.75	0.88	0.32	0.87	0.51	0.95	0.45	0.94	0.51	0.98	0.45	0.91
E _{GC}	0.99	1.0	0.81	0.65	0.8	1.0	0.89	0.76	0.87	1.0	0.96	1.0	0.94	1.0	0.98	1.0	0.91
E _{TaskDeserialization}	0.99	0.78	1.0	1.0	1.0	0.81	1.0	0.93	1.0	0.67	1.0	0.54	1.0	0.68	1.0	0.71	1.0
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.46	1.0	0.94	1.0	0.98	1.0	0.96	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.46	1.0	0.94	1.0	0.98	1.0	0.96	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.56	1.0	0.87	0.68	0.83	0.44	0.75	0.62	0.77	0.62	0.77	0.62	0.82	0.62	0.77	0.62	0.74
E _{TaskLocality}	0.33	0.67	0.45	1.0	0.98	0.79	1.0	0.79	1.0	0.79	1.0	0.79	1.0	0.79	0.33	0.98	0.44
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	344.92	0.15	364.35	21.16	33.77	0.11	19.31	0.17	20.05	0.03	16.24	0.04	15.73	0.03	339.72	0.19	-
Ref. Stage Duration (s)	614.12	0.14	1264.7	157.22	210.88	0.06	213.44	0.04	213.03	0.03	212.45	0.03	211.44	0.03	711.11	0.79	-
T _s	16	1	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16.0
K _s	150	1	150	150	150	16	150	16	150	16	150	16	150	16	150	200	149.95
Minor GC Count	1085	0	3096	3196	3309	0	3419	0	3521	0	3597	0	3691	0	5083	0	3113.64
Major GC Count	21	0	113	124	125	0	128	0	132	0	132	0	133	0	151	0	98.57

B. Experimental Results

[E30] Nodes: 4, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.
The simulated thread scheduler had to perform additional waiting of max. 2ms to schedule all tasks.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	3.06	0.01	5.24	4.7	10.64	0.04	19.71	0.01	18.92	0.01	7.0	0.04	20.78	0.02	3.99	2.5	5.19
E _{Parallel}	0.57	0.01	0.67	0.61	0.6	0.09	0.66	0.04	0.63	0.04	0.41	0.07	0.64	0.06	0.56	0.68	0.6
E _{LoadBalance}	0.59	0.02	0.71	0.61	0.6	0.1	0.66	0.05	0.63	0.04	0.41	0.1	0.64	0.08	0.58	0.78	0.62
E _{Communication}	0.96	0.92	0.95	1.0	1.0	0.91	1.0	0.92	1.0	0.92	1.0	0.74	1.0	0.74	0.96	0.87	0.96
E _{Serialization}	0.96	0.96	0.99	1.0	1.0	0.95	1.0	0.96	1.0	0.96	1.0	0.88	1.0	0.86	0.96	0.93	0.98
E _{Transfer}	1.0	0.96	0.95	1.0	1.0	0.95	1.0	0.95	1.0	0.96	1.0	0.84	1.0	0.85	1.0	0.94	0.98
E _{CompScaling}	6.09	0.83	10.36	16.61	24.41	0.51	35.13	0.33	34.74	0.28	21.1	0.97	35.97	0.76	7.59	7.09	10.48
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.89	0.77	0.75	0.46	0.73	0.79	0.85	0.72	0.87	0.64	0.81	0.54	0.91	0.52	0.95	0.52	0.84
E _{GC}	0.98	1.0	0.75	0.47	0.74	1.0	0.86	1.0	0.88	1.0	0.81	1.0	0.91	1.0	0.95	1.0	0.87
E _{TaskDeserialization}	0.91	0.8	1.0	0.99	0.98	0.84	1.0	0.8	1.0	0.73	1.0	0.66	1.0	0.67	1.0	0.62	0.97
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.97	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.97	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.46	1.0	0.59	0.45	0.6	0.53	0.54	0.53	0.52	0.53	0.52	0.53	0.6	0.53	0.46	0.14	0.52
E _{TaskLocality}	0.58	0.67	0.67	1.0	1.0	0.83	1.0	0.75	1.0	0.75	1.0	0.75	1.0	0.75	0.58	0.98	0.66
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	100.89	0.17	122.07	9.47	8.64	0.11	6.08	0.11	6.13	0.1	10.07	0.03	5.88	0.03	93.7	0.11	-
Ref. Stage Duration (s)	614.12	0.14	1264.7	157.22	210.88	0.06	213.44	0.04	213.03	0.03	212.45	0.03	211.44	0.03	711.11	0.79	-
T _s	64	1	64	64	64	8	64	8	64	8	64	8	64	8	64	64	63.91
K _s	150	1	150	150	150	8	150	8	150	8	150	8	150	8	150	200	149.79
Minor GC Count	804	0	2553	2199	1516	0	780	0	1527	0	2220	0	1991	0	4220	0	2393.48
Major GC Count	34	0	113	105	74	0	35	0	72	0	98	0	88	0	160	0	99.03

[E31] Nodes: 8, executors: 8, cores: 8, degree of parallelism: 8, memory fraction: 0.6.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	1.51	0.01	3.73	5.33	18.15	0.03	18.07	0.01	24.44	0.06	20.37	0.05	22.39	0.03	1.88	2.59	2.83
E _{Parallel}	0.88	0.01	0.84	0.61	0.75	0.09	0.61	0.04	0.7	0.09	0.68	0.08	0.68	0.07	0.91	0.73	0.87
E _{LoadBalance}	0.89	0.02	0.84	0.62	0.75	0.1	0.62	0.04	0.71	0.11	0.69	0.1	0.68	0.1	0.91	0.88	0.87
E _{Communication}	1.0	0.93	1.0	1.0	1.0	0.91	1.0	0.89	0.99	0.78	1.0	0.75	1.0	0.7	1.0	0.83	1.0
E _{Serialization}	1.0	0.96	1.0	1.0	1.0	0.96	1.0	0.94	1.0	0.88	1.0	0.84	1.0	0.81	1.0	0.92	1.0
E _{Transfer}	1.0	0.96	1.0	1.0	1.0	0.95	1.0	0.95	0.99	0.89	1.0	0.89	1.0	0.87	1.0	0.9	1.0
E _{CompScaling}	1.77	0.87	4.95	17.7	31.7	0.48	33.83	0.35	39.63	1.07	35.75	1.07	37.66	0.87	2.1	7.09	3.88
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.97	0.74	0.9	0.49	0.76	0.8	0.87	0.65	0.88	0.62	0.83	0.56	0.87	0.56	0.99	0.5	0.95
E _{GC}	0.99	1.0	0.9	0.49	0.78	1.0	0.88	1.0	0.89	1.0	0.84	1.0	0.88	1.0	0.99	0.99	0.96
E _{TaskDeserialization}	0.97	0.77	1.0	1.0	0.98	0.86	1.0	0.74	1.0	0.74	1.0	0.69	1.0	0.69	1.0	0.6	0.99
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.97	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99	1.0	0.97	1.0	0.99	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.29	1.0	0.65	0.45	0.6	0.6	0.48	0.59	0.48	0.59	0.48	0.59	0.6	0.59	0.46	0.16	0.45
E _{TaskLocality}	0.33	1.0	0.56	1.0	1.0	0.71	1.0	0.75	1.0	0.75	1.0	0.75	1.0	0.75	0.33	0.99	0.42
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	347.0	0.17	255.48	8.88	6.65	0.12	6.31	0.1	5.38	0.03	5.94	0.03	5.62	0.03	339.18	0.11	-
Ref. Stage Duration (s)	614.12	0.14	1264.7	157.22	210.88	0.06	213.44	0.04	213.03	0.03	212.45	0.03	211.44	0.03	711.11	0.79	-
T _s	64	1	64	64	64	8	64	8	64	8	64	8	64	8	64	64	63.97
K _s	150	1	150	150	150	8	150	8	150	8	150	8	150	8	150	200	149.94
Minor GC Count	1302	0	3126	2392	2005	0	861	0	1700	0	2216	0	913	0	5144	0	3121.88
Major GC Count	35	0	112	88	75	0	34	0	64	0	81	0	34	0	149	0	95.62

B.3. K-means

[E32] Nodes: 16, executors: 16, cores: 16, degree of parallelism: 16, memory fraction: 0.6.

Contrary to the configuration, Spark executed this application with 14 executors.

Metric	Stages															∅	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
E _{Global}	0.76	0.0	1.85	4.64	12.97	0.02	12.33	0.01	15.35	0.01	15.33	0.01	28.0	0.01	0.98	1.46	1.44
E _{Parallel}	0.37	0.0	0.2	0.24	0.34	0.05	0.25	0.02	0.29	0.03	0.28	0.03	0.36	0.03	0.38	0.52	0.34
E _{LoadBalance}	0.38	0.0	0.21	0.25	0.34	0.05	0.25	0.02	0.29	0.03	0.28	0.04	0.36	0.05	0.38	0.64	0.35
E _{Communication}	0.99	0.92	0.97	0.99	0.99	0.91	0.99	0.91	0.99	0.88	0.99	0.77	0.99	0.67	0.99	0.81	0.98
E _{Serialization}	0.99	0.96	0.97	1.0	1.0	0.95	1.0	0.96	0.99	0.95	1.0	0.9	1.0	0.79	0.99	0.92	0.98
E _{Transfer}	1.0	0.96	1.0	1.0	1.0	0.96	1.0	0.95	1.0	0.93	1.0	0.86	1.0	0.85	1.0	0.89	1.0
E _{CompScaling}	2.19	0.74	10.98	37.78	51.61	0.43	55.08	0.35	60.95	0.43	59.51	0.86	82.82	0.79	2.62	6.4	5.53
E _{TaskSuccess}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{Task}	0.93	0.78	0.83	0.5	0.74	0.8	0.9	0.72	0.88	0.59	0.91	0.49	0.94	0.51	0.99	0.44	0.93
E _{GC}	0.99	1.0	0.83	0.52	0.79	1.0	0.91	1.0	0.9	1.0	0.93	1.0	0.95	1.0	0.99	0.98	0.96
E _{TaskDeserialization}	0.94	0.82	1.0	0.98	0.95	0.86	0.99	0.83	0.99	0.7	0.99	0.63	0.99	0.67	1.0	0.56	0.97
E _{ResultSerialization}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffle}	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleFetch}	1.0	1.0	1.0	1.0	1.0	0.98	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{TaskShuffleWrite}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
E _{InputBalance}	0.99	1.0	0.82	0.88	0.88	0.52	0.88	0.62	0.88	0.62	0.88	0.62	0.88	0.62	0.99	0.05	0.96
E _{TaskLocality}	0.4	1.0	0.76	1.0	1.0	0.88	1.0	0.98	1.0	0.98	1.0	0.98	1.0	0.98	0.4	1.0	0.48
E _{MemorySpill}	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Stage Duration (s)	280.04	0.19	115.17	4.16	4.09	0.13	3.88	0.1	3.5	0.07	3.57	0.04	2.55	0.03	270.98	0.12	-
Ref. Stage Duration (s)	614.12	0.14	1264.7	157.22	210.88	0.06	213.44	0.04	213.03	0.03	212.45	0.03	211.44	0.03	711.11	0.79	-
T _s	150	1	150	150	150	16	150	16	150	16	150	16	150	16	150	200	149.9
K _s	150	1	150	150	150	16	150	16	150	16	150	16	150	16	150	200	149.9
Minor GC Count	1293	0	2624	947	610	215	389	0	417	198	1453	0	823	0	4541	0	2775.93
Major GC Count	57	0	109	41	25	10	17	0	18	8	59	0	37	0	145	0	99.5

Bibliography

- [1] S. Batistič and P. van der Laken, “History, Evolution and Future of Big Data and Analytics: A Bibliometric Analysis of Its Relationship to Performance in Organizations”, *British Journal of Management*, vol. 30, no. 2, pp. 229–251, 2019, ISSN: 1467-8551. DOI: 10.1111/1467-8551.12340. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8551.12340> (visited on 03/22/2023).
- [2] S. Bonner, I. Kureshi, J. Brennan, and G. Theodoropoulos, “Chapter 14 - Exploring the Evolution of Big Data Technologies”, in *Software Architecture for Big Data and the Cloud*, I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, and B. Maxim, Eds., Boston: Morgan Kaufmann, 2017, pp. 253–283. DOI: 10.1016/B978-0-12-805467-3.00014-4. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780128054673000144> (visited on 03/24/2023).
- [3] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, “Dynamic Configuration of Partitioning in Spark Applications”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1891–1904, 2017, ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2647939. [Online]. Available: <http://ieeexplore.ieee.org/document/7807262/> (visited on 02/13/2023).
- [4] P. Petridis, A. Gounaris, and J. Torres, “Spark Parameter Tuning via Trial-and-Error”, arXiv:1607.07348 [cs], arXiv, 2016. [Online]. Available: <http://arxiv.org/abs/1607.07348> (visited on 03/05/2023).
- [5] N. Ahmed, A. L. C. Barczak, T. Susnjak, and M. A. Rashid, “A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench”, *Journal of Big Data*, vol. 7, no. 1, pp. 1–18, 2020, ISSN: 2196-1115. DOI: 10.1186/s40537-020-00388-5. [Online]. Available: <https://journalofbigdata.springeropen.com/articles/10.1186/s40537-020-00388-5> (visited on 01/25/2023).
- [6] K. Wang and M. M. H. 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*, New York, New York: IEEE, 2015, pp. 166–173. DOI: 10.1109/HPCC-CSS-ICSS.2015.246.

- [7] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making Sense of Performance in Data Analytics Frameworks”, in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, USA: USENIX Association, 2015, pp. 293–307, ISBN: 978-1-931971-21-8. [Online]. Available: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout> (visited on 03/02/2023).
- [8] POP Centre of Excellence, *Pop standard metrics for parallel performance analysis*. [Online]. Available: <https://pop-coe.eu/node/69> (visited on 11/26/2022).
- [9] O. Yildiz and S. Ibrahim, “On the Performance of Spark on HPC Systems: Towards a Complete Picture”, in *Supercomputing Frontiers*, ser. Lecture Notes in Computer Science, R. Yokota and W. Wu, Eds., vol. 10776, Cham: Springer International Publishing, 2018, pp. 70–89. DOI: 10.1007/978-3-319-69953-0_5.
- [10] A. Rowstron, D. Narayanan, A. Donnelly, G. O’Shea, and A. Douglas, “Nobody ever got fired for using Hadoop on a cluster”, in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, ser. HotCDP ’12, Bern, Switzerland: Association for Computing Machinery, New York, 2012. DOI: 10.1145/2169090.2169092. [Online]. Available: <https://dl.acm.org/doi/10.1145/2169090.2169092> (visited on 03/24/2023).
- [11] M. Asch, T. Moore, R. Badia, *et al.*, “Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry”, *The International Journal of High Performance Computing Applications*, vol. 32, no. 4, pp. 435–479, 2018. DOI: 10.1177/1094342018778123. [Online]. Available: <http://journals.sagepub.com/doi/10.1177/1094342018778123> (visited on 03/26/2023).
- [12] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron, “Scale-up vs scale-out for Hadoop: Time to rethink?”, in *Proceedings of the 4th annual Symposium on Cloud Computing*, Santa Clara California: ACM, 2013, pp. 1–13, ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523629. [Online]. Available: <https://dl.acm.org/doi/10.1145/2523616.2523629> (visited on 03/26/2023).
- [13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Disk-Locality in Datacenter Computing Considered Irrelevant”, in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, vol. HotOS’13, Napa, California: USENIX Association, 2011, pp. 1–5. [Online]. Available: https://www.usenix.org/legacy/events/hotos11/tech/final_files/Ananthanarayanan.pdf (visited on 02/15/2023).
- [14] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, “Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks”, in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*,

- Taipei, Taiwan: IEEE, 2016, pp. 433–442, ISBN: 978-1-5090-3653-0. DOI: 10.1109/CLUSTER.2016.22. [Online]. Available: <http://ieeexplore.ieee.org/document/7776539/> (visited on 09/14/2022).
- [15] L. Liu, “Performance comparison by running benchmarks on Hadoop, Spark, and HAMR”, M.S. thesis, University of Delaware, Delaware, USA, 2015. [Online]. Available: <https://udspace.udel.edu/server/api/core/bitstreams/26f5ea47-9fee-41cd-a000-53bf23860379/content> (visited on 02/13/2023).
- [16] L. Gu and H. Li, “Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark”, in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Zhangjiajie, China, 2013, pp. 721–727. DOI: 10.1109/HPCC.and.EUC.2013.106.
- [17] S. Gopalani and R. Arora, “Comparing Apache Spark and Map Reduce with Performance Analysis using K-Means”, *International Journal of Computer Applications*, vol. 113, no. 1, pp. 8–11, 2015, ISSN: 09758887. DOI: 10.5120/19788-0531. [Online]. Available: <http://research.ijcaonline.org/volume113/number1/pxc3900531.pdf> (visited on 02/13/2023).
- [18] F. Liang, C. Feng, X. Lu, and Z. Xu, “Performance Benefits of DataMPI: A Case Study with BigDataBench”, in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, J. Zhan, R. Han, and C. Weng, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 111–123, ISBN: 978-3-319-13021-7. DOI: 10.1007/978-3-319-13021-7_9.
- [19] A. Döschl, M.-E. Keller, and P. Mandl, “Performance evaluation of Apache Hadoop and Apache Spark for parallelization of compute-intensive tasks”, in *Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services*, Chiang Mai Thailand: ACM, 2020, pp. 313–321, ISBN: 978-1-4503-8922-8. DOI: 10.1145/3428757.3429121. [Online]. Available: <https://dl.acm.org/doi/10.1145/3428757.3429121> (visited on 12/06/2022).
- [20] P. Zhou, Z. Ruan, Z. Fang, M. Shand, D. Roazen, and J. Cong, “Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-memory Computing Framework”, in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Belfast: IEEE, 2018, pp. 22–32, ISBN: 978-1-5386-5010-3. DOI: 10.1109/ISPASS.2018.00011. [Online]. Available: <https://ieeexplore.ieee.org/document/8366932/> (visited on 03/05/2023).
- [21] L. Canali, *SparkMeasure*, 2017. [Online]. Available: <https://github.com/LucaCanali/sparkMeasure> (visited on 01/31/2023).

- [22] L. Canali, *On Measuring Apache Spark Workload Metrics for Performance Troubleshooting*, 2017. [Online]. Available: <https://db-blog.web.cern.ch/blog/luca-canali/2017-03-measuring-apache-spark-workload-metrics-performance-troubleshooting> (visited on 01/31/2023).
- [23] L. Canali, *SparkMeasure, a tool for performance troubleshooting of Apache Spark workloads*, 2018. [Online]. Available: <https://db-blog.web.cern.ch/blog/luca-canali/2018-08-sparkmeasure-tool-performance-troubleshooting-apache-spark-workloads> (visited on 01/31/2023).
- [24] M. L. Massie, B. N. Chun, and D. E. Culler, “The ganglia distributed monitoring system: Design, implementation, and experience”, *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004, ISSN: 0167-8191. DOI: <https://doi.org/10.1016/j.parco.2004.04.001>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819104000535>.
- [25] J. Shi, Y. Qiu, U. F. Minhas, *et al.*, “Clash of the titans: MapReduce vs. Spark for large scale data analytics”, *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015, ISSN: 2150-8097. DOI: [10.14778/2831360.2831365](https://doi.org/10.14778/2831360.2831365). [Online]. Available: <https://dl.acm.org/doi/10.14778/2831360.2831365> (visited on 09/14/2022).
- [26] L. Canali, *Apache Spark 2.0 Performance Improvements Investigated With Flame Graphs*, 2016. [Online]. Available: <https://db-blog.web.cern.ch/blog/luca-canali/2016-09-spark-20-performance-improvements-investigated-flame-graphs> (visited on 12/06/2022).
- [27] P. Li, X. Huang, T. Zhao, Y. Luo, and Y. Cao, *Sparkling: Identification of Task Skew and Speculative Partition of Data for Spark Applications*, 2014. [Online]. Available: <https://www.databricks.com/session/sparkling-identification-of-task-skew-and-speculative-partition-of-data-for-spark-applications>.
- [28] M. U. Javaid, A. A. Kanoun, F. Demesmaeker, A. Ghrab, and S. Skhiri, “A Performance Prediction Model for Spark Applications”, in *Big Data – BigData 2020*, S. Nepal, W. Cao, A. Nasridinov, M. Z. A. Bhuiyan, X. Guo, and L.-J. Zhang, Eds., vol. 12402, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 13–22, ISBN: 978-3-030-59611-8 978-3-030-59612-5. DOI: [10.1007/978-3-030-59612-5_2](https://doi.org/10.1007/978-3-030-59612-5_2). [Online]. Available: http://link.springer.com/10.1007/978-3-030-59612-5_2 (visited on 01/25/2023).
- [29] Z. Gao, T. Wang, Q. Wang, and Y. Yang, “Execution Time Prediction for Apache Spark”, in *Proceedings of the 2018 International Conference on Computing and Big Data - ICCBD '18*, Charleston, SC, USA: ACM Press, 2018, pp. 47–51, ISBN: 978-1-4503-6540-6. DOI: [10.1145/3277104.3277109](https://doi.org/10.1145/3277104.3277109). [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3277104.3277109> (visited on 12/06/2022).

- [30] H. Herodotou, H. Lim, G. Luo, *et al.*, “Starfish: A Self-tuning System for Big Data Analytics”, in *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011*, 2011, pp. 261–272. [Online]. Available: https://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper36.pdf (visited on 01/05/2023).
- [31] L. Canali, *Apache Spark Performance Troubleshooting at Scale: Challenges, Tools and Methods*, Dublin, Ireland, 2017. [Online]. Available: https://canali.web.cern.ch/docs/Spark_Summit_2017EU_Performance_Luca_Canali_CERN.pdf.
- [32] M. Kang and J.-G. Lee, “An experimental analysis of limitations of MapReduce for iterative algorithms on Spark”, *Cluster Computing*, vol. 20, no. 4, pp. 3593–3604, 2017, ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-017-1167-y. [Online]. Available: <http://link.springer.com/10.1007/s10586-017-1167-y> (visited on 02/14/2023).
- [33] R. East, *Digging into Spark Scheduler Delay*, 2015. [Online]. Available: <https://mlspeed.wordpress.com/2015/09/10/digging-into-spark-scheduler-delay/> (visited on 02/07/2023).
- [34] M. Sinton-Hewitt, *Apache Spark - question everything*, 2018. [Online]. Available: <https://blog.scottlogic.com/2018/03/14/apache-spark-question-everything.html> (visited on 02/07/2023).
- [35] M. Kang and J.-G. Lee, “Effect of garbage collection in iterative algorithms on Spark: An experimental analysis”, *The Journal of Supercomputing*, vol. 76, no. 9, pp. 7204–7218, 2020, ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-020-03150-z. [Online]. Available: <http://link.springer.com/10.1007/s11227-020-03150-z> (visited on 02/14/2023).
- [36] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis”, in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, 2010, pp. 41–51. DOI: 10.1109/ICDEW.2010.5452747.
- [37] W. Gao, J. Zhan, L. Wang, *et al.*, *BigDataBench: A Scalable and Unified Big Data and AI Benchmark Suite*, Technical Report, arXiv:1802.08254 [cs], 2018. [Online]. Available: <http://arxiv.org/abs/1802.08254> (visited on 02/13/2023).
- [38] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, “SparkBench: A comprehensive benchmarking suite for in memory data analytic platform Spark”, in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, Ischia Italy: ACM, 2015, pp. 1–8, ISBN: 978-1-4503-3358-0. DOI: 10.1145/2742854.2747283. [Online]. Available: <https://dl.acm.org/doi/10.1145/2742854.2747283> (visited on 12/06/2022).

- [39] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, “PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications”, in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA: IEEE, 2010, pp. 1–11, ISBN: 978-1-4244-7557-5. DOI: 10.1109/SC.2010.41. [Online]. Available: <http://ieeexplore.ieee.org/document/5644905/> (visited on 03/13/2023).
- [40] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang, “Understanding the Influence of Configuration Settings: An Execution Model-Driven Framework for Apache Spark Platform”, in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, Honolulu, CA, USA: IEEE, 2017, pp. 802–807, ISBN: 978-1-5386-1993-3. DOI: 10.1109/CLOUD.2017.119. [Online]. Available: <http://ieeexplore.ieee.org/document/8030677/> (visited on 03/16/2023).
- [41] The Apache Software Foundation, *Apache Spark*, n.d. [Online]. Available: <https://spark.apache.org/> (visited on 11/16/2022).
- [42] The Apache Software Foundation, *Apache hadoop*, version 3.2.3, Mar. 28, 2022. [Online]. Available: <https://hadoop.apache.org>.
- [43] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, in *6th Symposium on Operating Systems Design and Implementation (OSDI ’04)*, San Francisco, CA: USENIX Association, 2004, pp. 137–149. [Online]. Available: <https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters> (visited on 11/22/2022).
- [44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets”, in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’10, Boston, Massachusetts, USA: USENIX Association, 2010. [Online]. Available: http://www.icsi.berkeley.edu/pubs/networking/ICSI_sparkclustercomputing10.pdf (visited on 11/16/2022).
- [45] X. Lin, P. Wang, and B. Wu, “Log analysis in cloud computing environment with Hadoop and Spark”, in *2013 5th IEEE International Conference on Broadband Network & Multimedia Technology*, 2013, pp. 273–276. DOI: 10.1109/ICBNMT.2013.6823956.
- [46] M. Zaharia, M. Chowdhury, T. Das, *et al.*, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”, in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’12, San Jose, CA, USA: USENIX Association, 2012.

- [47] T. da Silva Morais, “Survey on Frameworks for Distributed Computing: Hadoop, Spark and Storm”, vol. DSIE’15, 2015, pp. 95–105, ISBN: 978-972-752-173-9. [Online]. Available: https://paginas.fe.up.pt/~prodei/dsie15/web/papers/dsie15_submission_7.pdf (visited on 11/22/2022).
- [48] J. Laskowski, *The Internals of Apache Spark 3.3.1*. [Online]. Available: <https://books.japila.pl/apache-spark-internals/> (visited on 01/31/2023).
- [49] The Apache Software Foundation, *Spark 3.0.0 Documentation*, n.d. [Online]. Available: <https://spark.apache.org/docs/3.0.0/> (visited on 12/13/2022).
- [50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, *et al.*, “Apache Hadoop YARN: Yet another resource negotiator”, in *Proceedings of the 4th annual Symposium on Cloud Computing*, Santa Clara, California, USA: ACM, 2013, pp. 1–16, ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. [Online]. Available: <https://dl.acm.org/doi/10.1145/2523616.2523633> (visited on 02/13/2023).
- [51] The Apache Software Foundation, *Apache Hadoop 3.2.3 – HDFS Architecture*, 2022. [Online]. Available: <https://hadoop.apache.org/docs/r3.2.3/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> (visited on 03/11/2023).
- [52] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, 2007. [Online]. Available: https://svn.apache.org/repos/asf/hadoop/common/tags/release-0.16.3/docs/hdfs_design.pdf (visited on 03/11/2023).
- [53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System”, in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, USA, 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.
- [54] POP Centre of Excellence, *Performance Optimisation and Productivity | A Centre of Excellence in HPC*. [Online]. Available: <https://pop-coe.eu/> (visited on 11/26/2022).
- [55] POP Centre of Excellence, *POP Standard Metrics for Performance Analysis of Hybrid Parallel Applications | Performance Optimisation and Productivity*. [Online]. Available: <https://pop-coe.eu/further-information/learning-material/pop-standard-hybrid-metrics-for-parallel-performance-analysis> (visited on 11/16/2022).
- [56] POP Centre of Excellence, *In-depth explanation of the Additive Hybrid Metrics*. [Online]. Available: https://pop-coe.eu/sites/default/files/pop_files/pop_hybrid_metrics_additive_explained.pdf (visited on 11/16/2022).
- [57] Singularity Developers, *Singularity*, 2022. DOI: 10.5281/zenodo.4667718. [Online]. Available: <https://doi.org/10.5281/zenodo.1310023> (visited on 11/15/2022).

Bibliography

- [58] G. M. Kurtzer, V. Sochat, and M. W. Bauer, “Singularity: Scientific containers for mobility of compute”, *PLOS ONE*, vol. 12, no. 5, A. Gursoy, Ed., 2017, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459. [Online]. Available: <https://dx.plos.org/10.1371/journal.pone.0177459> (visited on 11/15/2022).
- [59] V. Sochat, “The Scientific Filesystem (SCIF)”, *GigaScience*, vol. 7, no. 5, giy023, 2018, ISSN: 2047-217X. DOI: 10.1093/gigascience/giy023. [Online]. Available: <https://doi.org/10.1093/gigascience/giy023> (visited on 12/13/2022).
- [60] SchedMD, *Slurm Workload Manager*, n.d. [Online]. Available: <https://slurm.schedmd.com/> (visited on 11/15/2022).
- [61] L. Yi and J. Dai, “Experience from Hadoop Benchmarking with HiBench: From Micro-Benchmarks Toward End-to-End Pipelines”, in *Advancing Big Data Benchmarks*, T. Rabl, N. Raghunath, M. Poess, M. Bhandarkar, H.-A. Jacobsen, and C. Baru, Eds., vol. 8585, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 43–48, ISBN: 978-3-319-10595-6 978-3-319-10596-3. DOI: 10.1007/978-3-319-10596-3_4. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10596-3_4 (visited on 12/06/2022).
- [62] R. Sinn and G. M. Ziegler, “Landau on Chess Tournaments and Google’s PageRank”, 2022. arXiv: 2210.17300 [math.HO]. [Online]. Available: <http://arxiv.org/abs/2210.17300> (visited on 03/23/2023).
- [63] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web”, Stanford InfoLab, Stanford, California, Technical Report 1999-66, 1999. [Online]. Available: <http://ilpubs.stanford.edu:8090/422/> (visited on 12/06/2022).
- [64] J. B. MacQueen, “Some methods for classification and analysis of multivariate observations”, in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, held at the Statistical Laboratory, University of California, June 21–July 18, 1965 and December 27, 1965–January 7, 1966*, J. Neyman and L. M. Le Cam, Eds., vol. 1, Berkeley and Los Angeles; London: Univ. California Press; Cambridge Univ. Press, 1967, pp. 281–297.
- [65] The Apache Software Foundation, *Apache Hadoop 3.2.4 - Documentation*. [Online]. Available: <https://hadoop.apache.org/docs/r3.2.4/> (visited on 12/13/2022).
- [66] S. Ryza, *How-to: Tune Your Apache Spark Jobs (Part 2)*, 2015. [Online]. Available: <https://blog.cloudera.com/how-to-tune-your-apache-spark-job-part-2/> (visited on 02/05/2023).

- [67] X. Hua, H. Wu, and S. Ren, “Enhancing Throughput of Hadoop Distributed File System for Interaction-Intensive Tasks”, in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Turin, Italy, 2014, pp. 508–511. DOI: 10.1109/PDP.2014.110.
- [68] S. Kambhampati, *Explore best practices for Spark performance optimization*, 2020. [Online]. Available: <https://developer.ibm.com/blogs/spark-performance-optimization-guidelines/developer.ibm.com/blogs/spark-performance-optimization-guidelines> (visited on 02/04/2023).
- [69] R. Manivannan, *Apache Spark Performance Tuning – Degree of Parallelism*, 2017. [Online]. Available: <https://dzone.com/articles/apache-spark-performance-tuning-degree-of-parallel> (visited on 02/04/2023).
- [70] IT Center RWTH Aachen, *Queue Partitions (RWTH High Performance Computing (Linux)) - IT Center Help*. [Online]. Available: <https://help.itc.rwth-aachen.de/service/rhr4fjjuttftf/article/e018f684c5624ae6b9bf7f0994d399f2/> (visited on 12/03/2022).
- [71] OpenSFS and EOFS, *Lustre*, 2023. [Online]. Available: <https://www.lustre.org/> (visited on 03/28/2023).
- [72] *Neuer Hochleistungsrechner CLAIX-2018 an der RWTH Aachen University*, 2021. [Online]. Available: <https://www.itc.rwth-aachen.de/cms/it-center/IT-Center/Aktuelles-test/Aktuelle-Meldungen/~rxnl/CLAIX-2018> (visited on 03/28/2023).
- [73] Software in the Public Interest & others, *Debian – Debian “stretch” Release Information*, n.d. [Online]. Available: <https://www.debian.org/releases/stretch/> (visited on 11/15/2022).
- [74] I. Rashid, *Spark Accumulators, What Are They Good For?*, 2015. [Online]. Available: <http://imranrashid.com/posts/Spark-Accumulators/> (visited on 02/07/2023).
- [75] *Vis.js*. [Online]. Available: <https://visjs.org/> (visited on 03/25/2023).
- [76] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”, in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS ’67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, New York, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560. [Online]. Available: <https://dl.acm.org/doi/10.1145/1465482.1465560> (visited on 03/25/2023).
- [77] V. K. Sowrirajan and R. Hu, *Best Practices for Enabling Speculative Execution on Large Scale Platforms*, Virtual Event, 2021. [Online]. Available: https://www.databricks.com/session_na21/best-practices-for-enabling-speculative-execution-on-large-scale-platforms (visited on 03/16/2023).

- [78] D. Jia, “ATMM: Auto Tuning Memory Manager in Apache Spark”, M.S. thesis, Northeastern University, Boston, Massachusetts, USA, 2018. [Online]. Available: <https://repository.library.northeastern.edu/files/neu:m044c5439/fulltext.pdf> (visited on 02/14/2023).
- [79] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade, “How Data Volume Affects Spark Based Data Analytics on a Scale-up Server”, in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, ser. Lecture Notes in Computer Science, J. Zhan, R. Han, and R. V. Zicari, Eds., vol. 9495, Cham: Springer International Publishing, 2016, pp. 81–92, ISBN: 978-3-319-29005-8 978-3-319-29006-5. [Online]. Available: http://link.springer.com/10.1007/978-3-319-29006-5_7 (visited on 02/14/2023).
- [80] Oracle Corporation, *Java platform, standard edition hotspot virtual machine garbage collection tuning guide: Generations*, 2015. [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html> (visited on 03/25/2023).
- [81] K. J. Matteussi, J. C. S. dos Anjos, V. R. Q. Leithardt, and C. F. R. Geyer, “Performance Evaluation Analysis of Spark Streaming Backpressure for Data-Intensive Pipelines”, *Sensors*, vol. 22, no. 13, p. 4756, 2022, ISSN: 1424-8220. DOI: 10.3390/s22134756. [Online]. Available: <https://www.mdpi.com/1424-8220/22/13/4756> (visited on 02/14/2023).
- [82] D. Wang and J. Huang, *Tuning Java Garbage Collection for Apache Spark Applications*, 2015. [Online]. Available: <https://www.databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html> (visited on 02/14/2023).
- [83] J. Wüthrich, *GCViewer 1.36*, 2019. [Online]. Available: <https://github.com/chewiebug/GCViewer/tree/1.36> (visited on 02/06/2023).
- [84] AbhijithLp and leifbro, *Task deserialization time is high - Azure Databricks*, 2022. [Online]. Available: <https://learn.microsoft.com/en-us/azure/databricks/kb/jobs/task-deserialization-time-high> (visited on 03/08/2023).
- [85] M. Liroz-Gistau, R. Akbarinia, and P. Valduriez, “FP-Hadoop: Efficient execution of parallel jobs over skewed data”, *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1856–1859, 2015, ISSN: 2150-8097. DOI: 10.14778/2824032.2824085. [Online]. Available: <https://dl.acm.org/doi/10.14778/2824032.2824085> (visited on 02/14/2023).
- [86] Z. He, Q. Huang, Z. Li, and C. Weng, “Handling Data Skew for Aggregation in Spark SQL Using Task Stealing”, *International Journal of Parallel Programming*, vol. 48, no. 6, pp. 941–956, 2020, ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-020-00657-z. [Online]. Available: <http://link.springer.com/10.1007/s10766-020-00657-z> (visited on 02/14/2023).

- [87] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling”, in *Proceedings of the 5th European conference on Computer systems*, Paris France: ACM, 2010, pp. 265–278, ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. [Online]. Available: <https://dl.acm.org/doi/10.1145/1755913.1755940> (visited on 02/07/2023).
- [88] X. Jiang, *Deep Dive into the Apache Spark Scheduler*, San Francisco, California, 2018. [Online]. Available: <https://www.databricks.com/session/deep-dive-into-the-apache-spark-scheduler>.
- [89] M. Heil, *Understanding common Performance Issues in Apache Spark - Deep Dive: Data Spill*, 2021. [Online]. Available: <https://michaelheil.medium.com/understanding-common-performance-issues-in-apache-spark-deep-dive-data-spill-7cdba81e697e> (visited on 03/17/2023).
- [90] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, *RRZE-HPC/likwid: Likwid-5.1.1*, 2021. DOI: 10.5281/zenodo.4665995. [Online]. Available: <https://zenodo.org/record/4665995> (visited on 01/24/2023).