Grade10: A Framework for Performance Characterization of Distributed Graph Processing

Tim Hegeman
VU Amsterdam
t.m.hegeman@vu.nl

Animesh Trivedi
VU Amsterdam
a.trivedi@vu.nl

Alexandru Iosup
VU Amsterdam
a.iosup@vu.nl

***Abstract*—Graph processing is one of the most important and ubiquitous classes of analytical workloads. To process large graph datasets with diverse algorithms, tens of distributed graph processing frameworks emerged. Their users are increasingly expecting high performance for diversifying workloads. Meeting this expectation depends on understanding the performance of each framework. However, performance analysis and characterization of a distributed graph processing framework is challenging. Contributing factors are the irregular nature of graph computation across datasets and algorithms, the semantic gap between workload-level and system-level monitoring, and the lack of lightweight mechanisms for collecting fine-grained performance data. Addressing the challenge, in this work we present Grade10, an experimental framework for fine-grained performance characterization of distributed graph processing workloads. Grade10 captures the graph workload execution as a performance graph from logs and application traces, and builds a fine-grained, unified workload-level and system-level view of performance. Grade10 samples sparsely for lightweight monitoring and addresses the problem of accuracy through a novel approach for resource attribution. Last, it can identify automatically resource bottlenecks and common classes of performance issues. Our real-world experimental evaluation with Giraph and PowerGraph, two state-of-the-art distributed graph processing systems, shows that Grade10 can reveal large differences in the nature and severity of bottlenecks across systems and workloads. We also show that Grade10 can be used in debugging processes, by exemplifying how we find with it a synchronization bug in PowerGraph that slows down affected phases by $1.10 - 2.50\times$. Grade10 is an open-source project available at https://github.com/atlarge-research/grade10.**

***Index Terms*—Performance analysis, Distributed graph processing, High performance computing, Performance Engineering**

I. Introduction

Graphs are convenient data structures used in a variety of domains, such as social media, Internet technologies, bioinformatics, logistics, road networks, and machine learning [1], [2]. Following trends in increasingly big data [3], graphs are becoming ever larger and their processing pipelines ever more complex. Although many distributed graph processing frameworks already exist [4]–[8], they need to adapt to these trends and process graphs increasingly faster. There are many timely performance challenges specific to graph processing—e.g., irregular workloads, idiosyncratic operation of graph processing frameworks—, which *generic* performance characterization frameworks [9]–[11] cannot fully address. In contrast, in this work we propose Grade10, a *specialized* performance characterization framework for distributed graph processing.

Performance characterization and analysis have a long and successful tradition. Generic tools and frameworks [9]–[11] are helpful in identifying performance issues, scalability bottlenecks, and execution inefficiencies. However, performance analysis and characterization of a distributed *graph* processing framework remains daunting. We identify and address in this work three challenges. Firstly, graph workloads are irregular and dynamic. Unlike typical data processing, where the runtime depends mostly on input size, graph processing times depend on the graph size, structure, and associated vertex values and edge semantics [12]–[15], thus leading to irregular work per iteration. For example, in a top-down graph traversal algorithm, per iteration work is proportional to the size of the traversal-frontier as well as the number of outgoing edges in the frontier [16]. Furthermore, many graph algorithms, like PageRank, dynamically iterate until the output of the algorithm converges, so the number of steps in the algorithm typically depends on the graph structure and per vertex values [17]. Hence, due to the irregular and dynamic nature, modeling graph workload remains challenging. Instead, the community uses empirical approaches to characterize, validate, and calibrate real-world performance profiles for graph workloads.

Secondly, there is a semantic gap between the data required for performance characterization from *system-level* and *workload-level* monitoring. As large-scale graph processing is distributed in nature, system-level distributed monitoring tools, such as Graphite [18] and Ganglia [19] can reveal typical performance issues, such as resource saturation or skewed load across multiple machines. However, as black-box approaches, they are unable to provide insights into performance characteristics across graph workload-level concepts, such as iteration boundaries, and workload skew due to poor graph partitioning. In contrast, workload-level monitoring as provided by frameworks themselves (e.g., Giraph [5], Spark [20]), can correlate some metrics like runtime, with workload-level concepts of iteration counts, partition sizes, or number of active vertices. Yet, workload-level monitoring also falls short of providing observability by linking workload-level performance issues (e.g., slow iteration) to a potential system-level root cause (e.g., network under-utilization). We posit combining both approaches to bridge the semantic gap is key to understanding the performance of graph processing workloads.

Lastly, there is a trade-off between the accuracy of performance data and monitoring overhead. As with distributed graph processing many performance-critical events, e.g., synchronization, task scheduling, message passing, happen on a fine time granularity, we need to monitor and collect data accurately and with fine granularity to identify rapidly shifting performance bottlenecks in the system [21]. However, fine-grained instrumentation of a data processing framework can be invasive and has high overheads [22]–[24]. We posit that a lightweight, low-overhead monitoring process is possible for graph processing, but it requires specialized approaches to upscale the low-resolution monitoring data.

To tackle these challenges, in this paper we propose Grade10, a framework for fine-grained performance characterization of distributed graph processing workloads. Our main contributions are:

1) We systematically study and analyze the requirements (R1-R5 in § II) for characterizing the performance of distributed graph processing frameworks.
2) Based on this analysis, we design Grade10 (§ III) for graph-workload peformance characterization. Grade10 collects detailed workload-level execution traces and augments them with coarsely collected systems-level monitoring data through a novel resource-attribution process to produce a fine-grained performance profile (§ III-D). From this profile, Grade10 identifies resource and performance bottlenecks, and associates them with individual graph workload-level concepts.
3) We conduct a real-world experimental evaluation of Grade10 (§ IV) to validate the ideas proposed in this work on two state-of-the-art graph processing frameworks, Giraph [5] and PowerGraph [4].
4) We open-source the Grade10 software, and make it available at: https://github.com/atlarge-research/grade10.

II. Performance Characterization Requirements

In this section, we start by synthesizing the requirements for graph performance analysis and characterization.

A. System Model

Graph processing is often an iterative, multi-phase operation [2], [12], [25]. Distributed graph processing frameworks have in common some architectural patterns, such as iterative and concurrent processing, distributed load balancing, and communication by message passing. However, beyond these high-level commonalities, graph frameworks span a broad range of different programming and execution models [1].

Due to the large variety of graph processing systems available—including native implementations written using low-level languages, and systems implemented on top of managed runtimes [21]—many kinds of resources and services can be used to process graphs. These resources include hardware provided by the underlying infrastructure, such as CPU, network, storage, and accelerators. Software resources include locks and queues, or even runtime services such as a garbage collector (GC), a data store, or a synchronization service.

TABLE I: Overview of related work and requirements synthesized from § II-B. We indicate for each work if it does (✓), does not (✗), or partially (~) fulfills a requirement.

Approach	R1	R2	R3	R4	R5
Blocked time analysis [9]	~	✓	~	✓	?
Retro [10]	✓	✗	✗	✓	✓
Tian et al. [11]	✓	~	~	✓	✓
Distributed tracing [26]–[28]	✗	✓	✓	~	✓
Critical path analysis [29]	✗	~	✓	✓	✓
Benchmarks [13], [30]	✗	✗	✗	✓	✓
Comparative studies [31]–[33]	~	~	✗	✓	~
Profiling/tracing [34], [35]	✓	~	~	~	✓
Logging [24], [36]–[38]	✗	✓	✓	~	✓
Distributed monitoring [39]–[42]	✓	✗	~	✓	✓
Grade10	✓	✓	✓	✓	✓

B. Requirement Synthesis

We now synthesize requirements from both the need for generic performance characterization and the graph-specific challenges. Table I summarizes our discussion and comparison with various state-of-the-art approaches (more in § VI).

(R1) Identify resource bottlenecks: A *resource bottleneck* refers to a workload execution period during which a particular hardware/software resource is saturated, thus, creating a bottleneck. Bottleneck identification can guide implementation and optimization efforts toward the resource(s) that limits the performance of the workload.

(R2) Identify performance issues: Beyond bottlenecks, graph processing systems suffer from a variety of performance issues. For example, workload imbalance can cause stragglers, synchronization can impose significant overhead, and applications can fail to saturate any resource due to a variety of reasons. Identifying common classes of performance issues is key to comprehensive characterization of graph workload performance.

(R3) Characterize performance with fine granularity: Due to irregular and rapidly changing workloads, fine-grained, per-resource performance characterization is necessary to study the effect of short-term bottlenecks. Furthermore, attributing bottlenecks and performance issues to fine-grained stages in a workload execution, as opposed to treating large parts or even the entire workload as a black box, aids in bridging the semantic gap between system-level and workload-level monitoring.

(R4) Impose low monitoring overhead: Fine-granular performance characterization needs fine-granular workload monitoring. However, fine-granular monitoring can lead to high overheads, which can cause perturbations in an application's execution and change the order or nature of bottlenecks and other performance issues [22]–[24]. Furthermore, due to large variations in graph workload performance across inputs, algorithms, and setups, performance characterization at smaller scale or in isolated environments may not be representative. Thus, monitoring must be lightweight to characterize the performance of production workloads at a fine time granularity.

(R5) Maintain framework- and workload-independence: The diversity of graph frameworks, datasets, algorithms, and runtime environments necessitates that any performance characterization tool must be graph framework or workload agnostic. With a modest engineering effort it must be portable across many distributed graph processing frameworks, as the system model in § II-A indicates.

III. Grade10 Design

Grade10 is a novel performance characterization framework for distributed graph processing workloads. Given a graph processing framework executing a workload, Grade10 generates a fine-grained performance profile and analyzes it automatically to find performance issues. It generates this profile through a complex but predominantly self-managed lifecycle (described in § III-A). At the core of this process are execution and resource models given by the user (§ III-B), which are augmented with fine-granularity traces (§ III-D) built from monitoring data (§ III-C) to identify resource bottlenecks and performance issues (§ III-E and III-F).

A. Life Cycle of a Performance Characterization Operation

We begin with an overview of a performance characterization operation conducted with Grade10. Figure 1 shows the overall steps and components. As input, the user provides the system under test (SUT) and the graph workload. The SUT (component 1 in the figure) encompasses both a graph processing framework (e.g., Giraph) and associated infrastructure. The workload (component 2) can either be a benchmark (e.g., Graphalytics [13]) or a specific, user-defined workload like PageRank on a web dataset. During execution, the SUT captures monitoring and logging data (component 3), for both the framework and the infrastructure (§ III-C).

Grade10's performance characterization process takes two additional inputs from the user: an *execution model* and a *resource model* of the SUT (components 4 and 5, explained in § III-B). The execution model defines how the framework executes the various *phases* of a workload, whereas the resource model describes the diverse hardware and software resources. Grade10's data collection process (component 6) uses these models to combine monitoring and logging data into a structured view of the application's execution. The resulting *execution* and *resource traces* feed the characterization.

Grade10's performance characterization proceeds in three stages. Firstly, in the resource attribution stage (component 7, § III-D) the execution and resource traces are combined (by *upsampling*) and analyzed to generate a fine-grained resource utilization *attribution* to individual execution phases of a workload. The upsampling process helps to attribute resource traces to an execution phase at a finer granularity than the resource traces are collected at. We analyze the accuracy of upsampling in § IV-B. Secondly, based on the attribution process, Grade10 *identifies bottleneck resources* which are fully consumed in different phases of execution (component 8). Finally, the fine-grained attribution and execution trace are further analyzed via a simulation process to *identify several classes of performance issues* and their *estimated impact* on the execution time of the graph application (component 9).

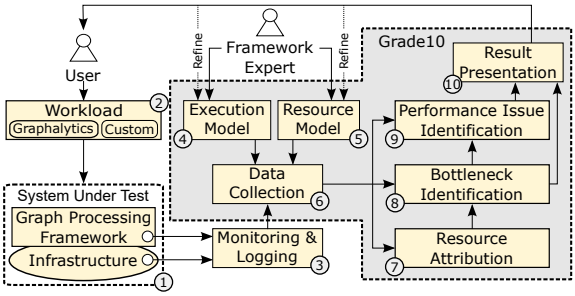


Fig. 1: The Grade10 architecture.

Grade10's result visualization (component 10) can help an analyst to identify promising optimization and tuning directions, and a domain-expert to refine performance models.

B. Execution and Resource Models

We now detail the execution and resource models used by Grade10 to facilitate systematic performance analysis across diverse graph processing systems and workloads, meeting **R5**.

The *execution model* describes the types of operations or *phases* that occur throughout the execution of a workload on a specific framework. A phase is a single logical operation. An execution model is a hierarchical direct acyclic graph (DAG) in which nodes represent phases, and directed edges represent the order of execution (Figure 2(a)). The model is nested: a node can be itself a DAG, to decompose a high-level phase into a graph of lower-level phases. For example, an execution model for Giraph may describe a Giraph application as three sequential, high-level phases: (P1) load the graph, (P2) perform the graph algorithm, and (P3) write results. P2 is further decomposed into a sequence of low-level superstep phases (i.e., iterations of a graph algorithm, P2.x), where each superstep phase is further decomposed into (P2.x.1) preparing Giraph's workers, (P2.x.2) processing a set of graph partitions, and (P2.x.3) synchronizing through a global barrier. By modeling graph applications as nested, hierarchical DAGs, Grade10 can characterize application performance by first relating system-level performance to fine-grained, low-level phases, and then propagating performance data up the hierarchy to characterize the performance of high-level phases.

The *resource model* defines which resources are available in the SUT. We use the term "resources" broadly, to include typical systems resources (e.g., CPU or network), software resources (e.g., queues or locks), and runtime services (e.g., garbage collection). Grade10 models the diverse resources used in graph processing as two distinct classes, *consumable* resources and *blocking* resources. Consumable resources (e.g., CPU cycles, network bandwidth) have a limited capacity. If a workload's demand for a consumable resource exceeds its capacity, then the workload slows down. Blocking resources (e.g., locks, queues) do not affect the execution of a phase when available, but block a phase's execution when unavailable. Grade10 models such resources as a sequence of

blocking events, i.e., time intervals during which the resource was not available. Runtime services such as garbage collectors can also be modeled as blocking resources; e.g., workload execution is periodically blocked to perform garbage collection.

We envision that the execution and resource *models are defined once, typically by a domain expert*. Then, with calibration, they can be used repeatedly by multiple users for the same graph framework and infrastructure.

C. Monitoring, Logging, and Data Collection

In addition to the models, which describe properties of *any* graph application run on the SUT, Grade10 collects execution logs and monitoring data, which describe *a particular* execution of a workload. Monitoring collects periodically resource utilization data from existing cluster monitoring systems (e.g., Ganglia). Execution logs provide workload-specific data that includes (with help from a domain expert) timestamps for many performance critical events.

Grade10 discretizes time into a sequence of *timeslices*, assuming that during a timeslice the SUT is in a steady state, i.e., resource consumption during a timeslice is constant and phases can only start/end at the start/end of a timeslice. This assumption can be held true for reasonably small timeslices, thus supporting requirement **R3**. Hence, the timeslice duration is an important parameter in tuning Grade10's performance characterization process, because it controls how fine-grained Grade10's analysis is with respect to time. In practice, *the timeslice duration can be set as low as tens of milliseconds*.

Ideally, both monitoring data and execution logs should match the short timeslice duration, i.e., their data must be collected at least every timeslice. Though possible for execution logs, timeslice-granular monitoring generates large amounts of data and imposes a high overhead on the system. To solve this challenge, Grade10 includes a novel resource attribution method (see § III-D) through which coarsely collected monitoring data (often over multiple timeslices, thus with low overhead) can be converted to timeslice granularity. We show empirically in § IV-B that our upsampling method is effective and can accurately upsample multi-timeslice monitoring data to reach single-timeslice granularity (thus, achieving **R4**).

Grade10 uses the execution and resource models provided by the user to parse the monitoring and logging data, building an *execution trace* and a *resource trace*. The former is a detailed execution trace of phase executions, at the granularity of a timeslice. For each phase, Grade10 extracts a start and an end time from the execution logs. The resource trace is a list of all resources in the SUT that were monitored during an application's execution, including coarsely collected consumable resource monitoring data, and framework-specific resource usage metrics (e.g., queue occupancy, `GC` information, blocking events) extracted from execution logs. The two traces have different granularity: whereas the execution trace uses the granularity of a single timeslice, the resource trace is at a coarser, multiple-timeslice granularity. In the next section, we explain how the resource trace is upsampled to match execution trace during the resource attribution process.

D. Resource Attribution

As described in the previous section, performance analysis requires fine-grained data (to meet **R3**), but to maintain low overhead Grade10 only collects coarse-grained monitoring data (**R4**). To address this challenge, the Grade10 resource attribution process takes as input coarse-grained resource traces and *infers* per phase the resource consumption, over time, with fine granularity (i.e, one timeslice duration). The process consists of three steps. Firstly, Grade10 estimates the demand over a timeslice for a given resource. Secondly, for each resource, Grade10 upsamples the resource trace to increase its granularity to match the timeslice duration. Finally, for each timeslice, Grade10 attributes the consumption of each resource to individual phases. Figure 2 explains this process through a constructed example, about the execution of a graph workload with four phases, $P1$ through $P4$, which can utilize three available resources, $R1$, $R2$, and $R3$ (e.g., CPU, network, storage). Figure 2(a) shows the execution trace of the workload, depicting phase-lengths in timeslices on the horizontal axis and concurrent phase-execution on the vertical.

1) Resource Demand Estimation: The intuition behind this step is that, for a given workload, resource consumption is correlated with the resource demand of phases. Resource demand estimation is guided by *resource attribution rules* given by an expert user in Grade10. These rules define a matrix with columns for each phase P and rows for each resource R, each matrix element indicating the rule linking the demand for R to P. Figure 2(b) shows an example of such matrix. Rules may be different, leading to cells with different *kinds* of values. Currently, Grade10 supports three resource attribution rules:

R1. The **None** rule indicates that phase P does not use resource R. For example, "-" in Figure 2(b) indicates resources $R2$ and $R3$ are not used by phase $P1$.

R2. The **Exact** rule indicates that a phase P has an exact demand for a resource R. This rule has one parameter, the proportion. In the figure, phase $P3$ uses 50% of resource $R2$ (e.g., half of $R2$'s CPU cores).

R3. The **Variable** rule indicates that phase P may use resource R as much as possible, but with an unknown, variable, and/or relative demand. These are represented as variables (x, y) in the figure. In our running example, the exact demand of phase $P2$ on resource $R1$ is unknown, but its demand is twice that of phase $P1$ ($2x$ vs. $1x$).

From these basic rules, Grade10 builds a timeslice-granular resource demand estimation matrix. From the execution traces, Grade10 determines which phases are active (i.e., the phase has started, has not yet ended, and is not interrupted by a blocking event) at a given time slice (Figure 2(a)). Grade10 then sums up the demands of active phases with Exact rules to determine known demand, and sums up the active phases with Variable rules to determine the variable demand. The result of this process is a timeslice-granular resource demand estimation matrix for every resource, including expanded forms of the variables given by rules, as shown in Figure 2(c).

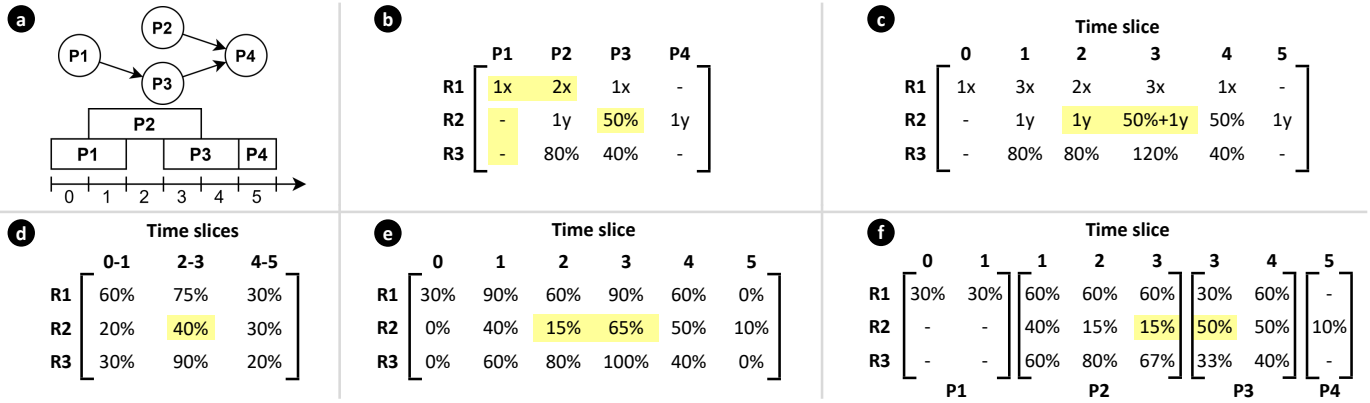


Fig. 2: Constructed example of Grade10's resource attribution process. Execution DAG and phase execution (in (a)) are derived from the execution model and trace. Resources $R1$, $R2$, and $R3$ can represent typical resources such as CPU, network, and storage. Monitoring data is collected at a $2\times$ timeslice quanta (shown in (d)). Highlighted entries are discussed in § III-D.

2) Upsampling Resource Traces: Recall from § III-C that resource monitoring data is collected at coarser granularity (multiple timeslices) than the execution traces (single timeslice). In Figure 2(d), the monitoring data for each resource is collected at two quanta, i.e., each data point represents a measurement over two timeslices. Each resource consumption measurement represents the average rate of consumption since the previous measurement. For coarse-grained monitoring, the period between consecutive measurements can be long (to reduce the monitoring overheads) and the average rate of consumption may not accurately reflect actual use of a resource (e.g., coarse-grained monitoring misses burstiness).

The aim of the upsampling process is to infer from monitoring accurate data at timeslice granularity. Intuitively, the *demand* (at timeslice granularity) and actual *consumption* (averaged over multiple timeslices) are known and can be superimposed to accurately generate resource comsumption at a timeslice granularity. Grade10 takes this approach, by considering independently each measurement for each resource. First, the total resource consumption is divided over the multiple timeslices, proportionally to the known resource demand in each time slice, without exceeding the demand or the capacity of the resource (whichever is lower). If the total resource consumption exceeds the Exact demand during a measurement period, the remainder is divided proportionally (load-balanced) to the estimated Variable demand. For example, the total demand for resource $R2$ for timeslices 2-3 is known to be $50\% + 2y$ (in Figure 2(c), sum of the two matrix elements), whereas the actual average consumption over 2 timeslices is 40% (in Figure 2(d), the single matrix element for $R2$ and 2-3). Putting these together, Grade10 computes the actual resource consumption to be at 15% and 65% for timeslices 2 and 3, respectively. This output is part of the complete timeslice-granular resource-consumption matrix computed by Grade10, which Figure 2(e) depicts.

3) Attribution to Phases: The final step in Grade10's resource attribution process is attributing resource consumption to individual phases of execution within a single timeslice. The attribution step is applied independently for each resource and timeslice. First, Grade10 identifies which phases with an Exact attribution rule (see § III-D1) are active during the timeslice. The upsampled resource consumption is attributed to each Exact phase proportionally to and not exceeding their resource demand. Next, any remaining resource consumption is proportionally distributed based on the relative demands (calculated from Variable attribution rules) of all active phases.

The result of this process is a *3d-array*: the timeslice-granular resource consumption *for each execution phase*, as exemplified in Figure 2(f). In the example, at timeslice 3, only the phases $P2$ and $P3$ are active. (To see why, compare Figures 2(a) and (f).) Both of these active phases need resource $R2$ for execution, with the total demand of $50\% + 1y$ (Figure 2(c), element for $R2$ and timeslice 3). The actual resource consumption is at 65% (Figure 2(e), same element). Hence, by giving the first 50% to $P3$ (Exact assignment), we are left with 15% for $P2$ (Variable assignment). These values appear in the corresponding elements, in Figure 2(f).

E. Resource-Bottleneck Identification

To address **R1**, the Grade10 performance characterization pipeline uses a specialized process for detecting resource bottlenecks. Resource bottlenecks are important guides for performance engineering. For example, whenever an application saturates the bandwidth of a network link, that part of the application is bottlenecked by the network and can only be sped up by increasing the performance of the network or reducing the amount of data sent across the network.

To identify resource bottlenecks, Grade10 uses separate approaches for blocking and for consumable resources. Bottlenecks on blocking resources are straightforward to detect. Whenever a phase is blocked on a resource, that resource is delaying the execution of the phase and thus forms a bottleneck. Because a list of blocking events is part of Grade10's input, Grade10 can compute for each phase how much time that phase spent bottlenecked (blocked) on each blocking resource. These bottlenecks are comparable to the notion of blocked time [9], but work in another context; they both refer to

a phase/task halting execution until the operation occupying some resource completes (e.g., a GC-event in Grade10, a disk read or network transfer in blocked time analysis).

Bottlenecks on consumable resources can occur in two different situations: when a resource is saturated, and when a phase reaches its upper limit on how much of a resource it needs. When a resource is saturated, i.e., reaches full (100%) utilization, all phases that compete for this resource are experiencing a bottleneck. Grade10 identifies such bottlenecks by analyzing the upsampled resource metric (see § III-D2) of each resource for extended periods of full utilization. Next, Grade10 determines which phases were using that resource during periods of saturation and marks those phases as bottlenecked. For example, in Figure 2(e), resource $R3$ is consumed 100% during timeslice 3. During timeslice 3, both $P2$ and $P3$ are active, and both are dependent on $R3$. Hence, Grade10 marks both $P2$ and $P3$ as resource bottlenecked on $R3$ during this timeslice, and suggests to developers that providing more of $R3$ would help with the performance of both phases.

One of the least understood phenomena in graph processing is that *resource bottlenecks can still occur even when a resource is not saturated*. When a phase is limited to using only part of a resource (e.g., a phase limited to using only 2 out of 4 CPU cores, expressed in Grade10 as Exact rule, 50%), that phase is bottlenecked on a resource if it uses as much as its upper limit allows, regardless of whether the resource itself is saturated. Grade10 identifies such bottlenecks by analyzing the per-phase resource usage produced by its resource attribution process for phases with an Exact attribution rules. For example, $P2$ can use 80% of $R3$ (Figure 2(b)). In the final outcome of the resource attribution process, in Figure 2(e), we observe $P2$ uses 80% of $R3$ during timeslice 2. Here, $P2$ is bottlenecked on $R3$, as its 80% Exact demand is met, despite the resource itself not being 100% utilized. Hence, Grade10 recommends to configure $P2$ to use 100% of $R3$ (instead of 80%), which would likely yield better performance.

F. Performance-Issue Detection

To detect performance issues (**R2**), Grade10 systematically determines for each phase in a given workload which kinds of performance issues might apply, from several classes of issues Grade10 has been designed to detect. For each possible performance issue, Grade10 uses simulation to estimate how the execution of an application would change if some potential performance issue was fixed. Our approach is similar to blocked time analysis [9], which uses simulation to estimate the performance that could be gained by removing disk or network bottlenecks, but applied to the different context of graph processing (with different execution models, different kinds of resources, and different types of performance issues).

Grade10 simulates applications by replaying the captured execution trace. Grade10's simulator assumes a simplified system model whereby each phase has a fixed duration as recorded in the execution trace, with no delays between phases. While replaying a trace, Grade10 obeys the precedence constraints (defined in the execution model), as well as scheduling constraints related to concurrency and locality (if available). For example, in many distributed graph processing frameworks, compute tasks cannot migrate between machines to avoid expensive data movement.

For each potential performance issue, Grade10 first determines how solving that issue would change the duration of a specific set of phases. Next, it simulates the application with adjusted phase-durations to derive an *optimistic* application makespan. Finally, Grade10 compares the optimistic makespan with the simulated makespan of the original execution trace, and obtains an upper bound on the reduction in makespan possible by fixing a specific performance issue. Grade10 only reports these performance issues if the possible reduction in makespan is larger than an arbitrary minimum threshold.

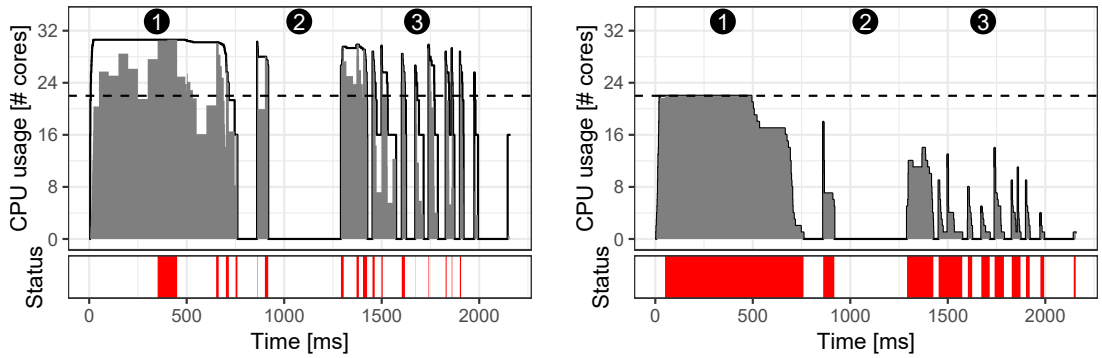
Using this general approach for detecting performance issues, Grade10 is currently able to identify two common classes of performance issues in distributed graph processing, extensive resource bottlenecks and imbalanced execution (see § IV-C and IV-D for evaluation). Firstly, to detect and evaluate the *impact of resource bottlenecks*, Grade10 systematically considers combinations of phases and resources, and simulates how completely removing bottlenecks on some resource could speed up the application. To simulate removing a bottleneck, Grade10 computes how much shorter a phase could become until another resource becomes bottlenecked. For the example in Figure 2, if we are to optimize $R3$, which is at 100% during timeslice 3, then $R1$ becomes the next bottleneck, and the gain margins are limited. However, removing the $R1$ bottleneck as well could yield more performance, as the next bottleneck is caused by $R2$, which is only at 65% at that moment.

Secondly, to detect and evaluate the impact of *imbalanced execution*, Grade10 identifies sets of concurrent phases of the same type and simulates how the application execution would change if those phases were perfectly balanced. Grade10 assumes that only the work performed by concurrent phases of the same type is interchangeable (e.g., compute phases during one iteration of a graph algorithm, but not from different iterations). Grade10 also assumes that, absent the imbalance, each phase in a set of concurrent phases would have the same duration and the total duration of all phases is preserved.

IV. Experimental Results

In this section we evaluate Grade10 by using it to characterize the performance of two state-of-the-art distributed graph processing systems (thus, meeting **R5**) on our experimental setup (§ IV-A). Our key findings are:

1) Coarse-grained system-level monitoring data can be used as a high fidelity source for fine-grained performance modeling using Grade10's resource attribution process (thus, meeting **R3, R4**). The process produces high-quality traces with less than 20% upsampling errors for tuned models where monitoring data is collected at the $8\times$ timeslice granularity. (§ IV-B)
2) Fine-grained performance profiles generated by Grade10 can correctly identify resource bottlenecks while confirming the apriori knowledge in the field (thus, meeting **R1,**



(a) Resource attribution *without* attribution rules.

(b) Resource attribution *with* attribution rules.

Fig. 3: Outcome of Grade10's resource attribution process for an exemplary Compute phase. *(top)* Attributed CPU usage (shaded area), estimated resource demand (curve), and the number of compute threads (dashed line) over time. *(bottom)* Presence of CPU bottlenecks (shaded red if present), over time.

R2). For example, message queues in Giraph, not the network as expected in a distributed setting, are the most impactful bottleneck for 7 out of 8 jobs, accounting for up to 57.8% of the processing time, thus confirming the observation (lack of CPU or network related bottlenecks) in a previous Giraph study [33] (§ IV-C).

3) Grade10 is an effective performance debugging tool. It helped us identify a synchronization bug in PowerGraph that slows down affected phases by $1.10-2.50\times$. (§ IV-D)

A. Experiment Setup

We use in our experiments two state-of-the-art distributed graph processing systems, Giraph [5] and PowerGraph [4]. We instrumented Giraph and PowerGraph with 38 and 37 additional logging statements to capture performance-critical events in the execution logs. As graph workloads, we run four graph algorithms (BFS, CDLP, PR, and WCC) from the Graphalytics benchmark [13] on two graph datasets (Datagen-1000 and Graph500-26). All experiments were run on a local cluster (DAS-5 [43]) with 10 dedicated machines with dual Intel Xeon E5-2630v3, 64 GiB memory, and FDR InfiniBand.

B. Validation of Resource Attribution Process

We validate Grade10's resource attribution process by closely analyzing two key steps in it. First, we study the impact of resource attribution rules (§ III-D1) on the resource attribution process. Second, we quantify the accuracy of the upsampling process (§ III-D2) by comparing its results to a (partial) ground truth. The aim of this section is to demonstrate that Grade10 can produce intuitive and sound results.

Impact of Attribution Rules: We manually inspect the results of resource attribution in two configurations: one with tuned resource attribution rules, and another without. We run a PageRank application on Giraph (more thorough analysis and examples available online [44]) and model the execution of one worker node during a single algorithm iteration as two concurrent phases: Compute and Communicate. The Compute phase in turn comprises a set of ComputeThread phases. The attributed resource usage is the sum over all ComputeThreads in a single Compute phase. In all settings, we calculate the Compute phase resource usage as well as demand. In case no rules are provided, Grade10 assumes an implicit Variable rule with a parameter of $1x$ for every phase.

Figure 3 depicts the result of our experiment. The top sub-plots depict the attributed CPU usage and estimated CPU demand of the Compute phase over time for two configurations (with and without rules). The bottom sub-plots depict the presence of CPU bottlenecks over time, as identified by Grade10. The figure shows three distinct situations: ❶ a period in which most compute threads were consistently active and message queues were not yet full, ❷ a GC pause halting execution, and ❸ a period in which message queues were full with short bursts of activity as messages left the queue.

When no attribution rules are provided (Figure 3a), we observe in ❶ that Grade10 erroneously estimates CPU demand above 23 cores, i.e., the number of compute threads Giraph was using. Grade10 wrongly concludes that during ❶, the Compute phase is rarely bottlenecked on the CPU, while intuitively compute threads should be primarily CPU-bound when not blocked on message queues or GC. During the GC pause (❷), Grade10 correctly finds that the Compute phase is not using the CPU. Finally, when Giraph's message queues are full (❸), Grade10 correctly identifies that CPU demand fluctuates over time, but overestimates the precise demand, which should be one CPU core per active thread (not depicted).

Figure 3b depicts the same phase analyzed by Grade10 with comprehensive attribution rules. In particular, we indicate in our resource model that an active compute thread is expected to always use precisely one CPU core (Exact rule, $100\%/\#\text{cores}$). The outcome of Grade10's resource attribution process using this configuration matches better our intuition and understanding of Giraph. For example, we observe that Grade10's estimate of CPU demand never exceeds the number of threads during ❶, and that the CPU usage Grade10 attributes to the Compute phase during ❸ is (close to) one core per active compute thread. This leads Grade10 to correctly conclude that whenever a compute thread is not blocked, it is bottlenecked by the CPU.

TABLE II: Relative sampling error produced by resource attribution with or without Grade10's upsampling method.

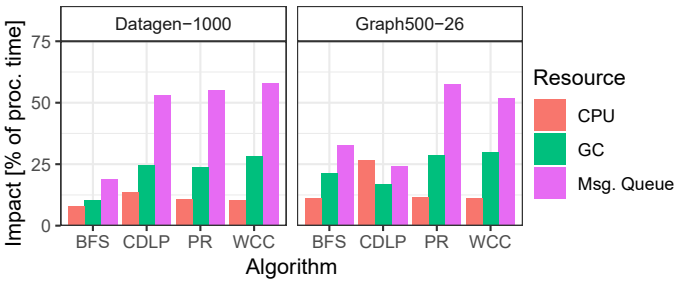
Framework	Upsampling Method	Tuned Model?	Monitoring Interval						
			50 ms	100 ms	200 ms	400 ms	800 ms	1600 ms	3200 ms
Giraph	Constant	N/A	0.00%	12.03%	20.43%	31.57%	47.32%	68.52%	82.97%
	Grade10	✗	0.03%	15.24%	26.89%	41.26%	58.30%	80.78%	91.02%
		✓	0.03%	7.37%	11.91%	18.83%	29.75%	46.37%	56.71%
PowerGraph	Constant	N/A	0.00%	32.82%	58.97%	81.95%	94.94%	97.96%	98.71%
	Grade10	✓	0.12%	7.36%	11.27%	11.62%	11.87%	13.23%	15.28%

Accuracy of the Upsampling Process: To quantify the accuracy of Grade10's upsampling process, we compare its output to a (partial) ground truth. Collecting resource usage metrics *per phase* at timeslice granularity (e.g., 10 milliseconds) is not feasible, so we prepare monitoring data collected *per machine* at 50 ms intervals as ground truth. We downsample the collected monitoring data by averaging up to 64 consecutive measurements to produce a coarse-grained resource trace. We apply Grade10's upsampling process to resource traces with varying granularity and compute for each 50 ms period how closely Grade10's upsampled resource trace matches the original monitoring data.

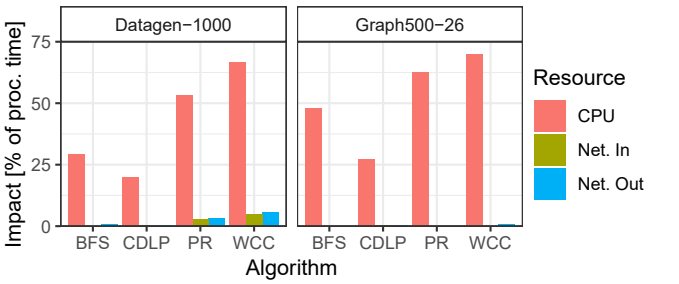
Table II depicts the relative sampling error of Grade10 attributing CPU usage of a PageRank job run on Giraph and PowerGraph. The relative sampling error expresses the sum of absolute differences between Grade10's upsampled resource trace and ground truth trace, as a percentage of the total resource consumption. We compare Grade10's upsampling method, using resource demand estimation and attribution rules, to a strawman approach of assuming constant resource usage during a measurement period. We find that the constant approach to upsampling performs poorly for coarse-grained resource traces (3200 ms, $64\times$ longer than the ground truth), with sampling errors of $82.97 - 98.71\%$. Grade10's upsampling method performs comparably poorly (91.02% error) for Giraph when the attribution process is not tuned. However, after tuning Giraph's model by modeling garbage collection events, Grade10's error for Giraph improves to 56.71%. For more moderate upsampling ratios, Grade10's error rate are significantly better (e.g., up to 18.83% at 400 ms, or $8\times$). Our execution model for PowerGraph is comprehensive and tuned, and it allows Grade10 to reach sampling errors under 15.28%, even when upsampling by $64\times$. Although we are not able to compare to a ground truth at timeslice granularity broken down per phase, we conclude that Grade10 can upsample resource traces with good accuracy. Based on our results, we conservatively recommend upsampling by up to $8\times$ to achieve a good balance between accuracy and reduced monitoring overhead, although higher upsampling ratios can be selected for finely tuned models (e.g., PowerGraph).

C. Resource Bottlenecks in Graph Processing

Grade10's resource bottleneck detection and performance issue identification processes provide an optimistic estimate of the impact of different types of bottlenecks (i.e., an upper bound on how much the application's performance could improve by eliminating all bottlenecks on a resource). In this experiment, we use Grade10 to detect bottlenecks in Giraph and PowerGraph on eight workloads (combining the two datasets and four algorithms introduced in § IV-A).



(a) Giraph's bottlenecks. Network bottlenecks ($< 0.1\%$) not depicted.



(b) PowerGraph's bottlenecks. `GC` and `Msg. Queue` not applicable.

Fig. 4: Optimistic estimation of impact of resource bottlenecks on the processing time of graph processing jobs spanning 2 frameworks, 2 datasets (`Datagen-1000` and `Graph500-26`), and 4 algorithms (defined in § IV-A).

Figure 4 depicts the estimated impact of bottlenecks on processing time (i.e., the algorithm execution phase, excluding loading the input and writing the output). For Giraph (Figure 4a), full message queues are the most impactful bottleneck for 7 out of 8 jobs, accounting for up to 57.8% of the processing time. The next largest bottleneck is garbage collection at around 25%. Bottlenecks on hardware resources are much less impactful: CPU bottlenecks account for around 10% (with an outlier at 26.6%), and the impact of network throughput bottlenecks is below 0.1% for all jobs. The lack of CPU and network bottlenecks confirms findings by Satish et al. [33], who found that Giraph achieved poor CPU and network utilization in each of their experiments.

In contrast, PowerGraph (Figure 4b) is primarily CPU-bound, although the impact of CPU bottlenecks varies be-

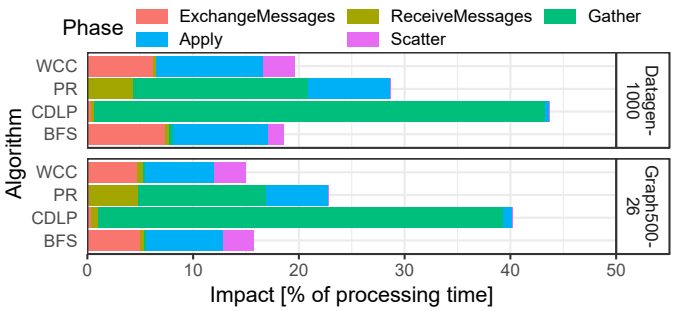


Fig. 5: Optimistic estimation of the impact of imbalance on the processing time of PowerGraph jobs, for key types of phases.

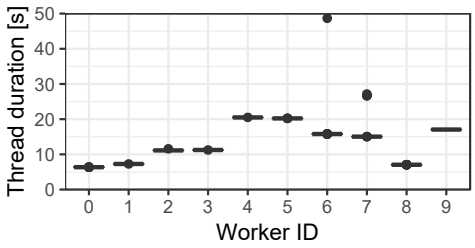


Fig. 6: Duration of *Gather* threads in the first superstep of PowerGraph running CDLP on Graph500-26. The horizontal "thick lines" are narrow boxplots. Points are outliers.

tween 20.0% and 69.9% across our selection of workloads. Network bottlenecks occur in PowerGraph, but their impact is insignificant at up to 5.5%. Due to differences in architecture, PowerGraph does not experience the garbage collection or message queue bottlenecks that are prevalent in Giraph. In particular, PowerGraph is written in C++ and therefore does not use a garbage collector, and its communication subsystem uses a different approach to message queues that does not lead to explicit stalls. Overall, our findings open up avenues for future work to improve Giraph's communication subsystem to better utilize available resources, or to identify and address the possible overhead that prevents PowerGraph from utilizing all available compute resources across different workloads.

D. Discovery of a Synchronization Bug in PowerGraph

To demonstrate how Grade10 can be used to detect performance issues, we describe in this section how we use Grade10's automated analysis of workload imbalance to discover a synchronization bug in PowerGraph. Grade10 is especially useful in identifying this bug, because Grade10's low overhead and automated process make it feasible to characterize the performance of many jobs, and thus find performance issues that occur only sporadically.

Figure 5 summarizes the output of Grade10's workload imbalance detection for eight different jobs run on PowerGraph. The figure depicts the estimated impact of workload imbalance in five key types of phases. We observe that workload imbalance accounts for a significant portion of PowerGraph's algorithm execution time, up to 43.7%. Notably, Grade10 finds that imbalance during "Gather" steps of the CDLP algorithm is highly impactful (38.3-42.7%). Thus, by improving PowerGraph's load balancing during Gather phases, the runtime of CDLP jobs could be reduced by up to 42.7%.

Next, we explore in more detail one of the jobs most affected by imbalance. Figure 6 depicts the duration of each worker thread used by PowerGraph during its first iteration of the Gather step. We observe that PowerGraph's imbalance has two causes. First, there is a large imbalance between the median duration of threads on different workers, ranging from 6.4-$20.5s$. This is most likely caused by poor workload distribution, which is typical for graph applications due to their irregularity. Second, we observe that some threads take significantly longer than other threads on the same worker. For example, one thread on worker 6 takes $2.88\times$ as long as the mean thread on that worker. In this example, Grade10 finds that the presence of such outliers increases the runtime of this Gather step from $20.5s$ (slowest thread without outliers, worker 4) to $48.7s$ (slowest outlier) for an estimated slowdown of $2.38\times$. Overall, Grade10 finds similar outliers affect 20% of non-trivial processing steps (phases lasting $> 1s$), with slowdowns of $1.10 - 2.50\times$.

After using Grade10 to detect outliers in gather threads when running the CDLP algorithm, we use traditional performance analysis tools to analyze one job in more depth and we locate a bug in PowerGraph's handling of cross-thread barriers. In particular, we find that PowerGraph interleaves computation and communication in each worker thread. Each thread processes a set number of vertices in the graph, then handles any pending communication, and repeats until the entire graph is processed. Then, all threads synchronize using a barrier, handle any remaining communication, and finally proceed to the next step in the graph algorithm. Occasionally, all threads but one reach the barrier after finding there are no pending messages. Next, a stream of messages arrives before the last thread checks for and starts processing pending messages. If the incoming message rate exceeds the rate at which one thread can process messages, that thread continues to process messages until no new messages arrive, while all other threads are idly waiting at the barrier.

V. Discussion, Limitations, and Ongoing Work

Need for expert input: Grade10's performance characterization processes currently rely on expert input (execution and resource models, resource attribution rules) to achieve good accuracy. However, expert input only needs to be defined and fine-tuned once for a given graph processing framework (not workload), and can be reused by many users. A complete modeling and fine-tuning of PowerGraph took only a week. Modeling and tuning of Giraph took an additional week to also include its complex, software resources such as internal message queues, and GC interference. From our experience, we believe Grade10's requirement of expert input is not inhibitive. As ongoing work, we are exploring how expert input can be reduced or even eliminated, e.g., by using machine learning techniques to infer resource attribution rules, or by extracting execution models directly from source code.

Current limitations: Though Grade10 identifies resource and performance bottlenecks by bridging the gap between workload-level and system-level performance data, it does not present solutions regarding how to alleviate identified bottlenecks. Some bottlenecks can be trivially solved by tuning a configuration parameter, others might require more intimate knowledge about the system. We envision that resource attribution at source or machine code level would significantly help a targeted performance optimization effort. Furthermore, the current Grade10 resource model does not support resources that do not fit in its consumable or blocking resource archetypes, e.g., CPU cache hit rates, or IPC counts. We are also aware that the simplified simulator (see § III-F) is limited by the execution trace it replays, and thus it does not assume performance variability, does not consider that eliminating a performance issue may cause another to gain prominence, etc.
Extending to other domains: Although designed for graph processing, we believe that Grade10's modeling and performance characterization approach can be extended to broader DAG-based data processing systems such as Spark and TensorFlow. We are in the process of characterizing Spark workloads by extending Grade10's methods to model Spark's fine-grained sub-millisecond phases and asynchronous remote data accesses (e.g., through distributed tracing [26]–[28]). Such extensions are necessary to maintain high resource attribution accuracy. For machine learning workloads, we are considering modeling of accelerator resources and for supporting new classes of performance issues, e.g., data movement between accelerator and system memory.

VI. Related Work

In this section, we survey related work on performance characterization and comparison of big-data applications, and on general-purpose performance characterization, including tracing, logging, and monitoring. Currently, Grade10 is the first specialized performance characterization framework for distributed graph processing.

Ousterhout et al. [9] propose blocked time analysis to estimate the impact of blockable hardware disk and network resources on Spark applications. In contrast, Grade10 includes a broader class of hardware as well as software resources to identify multiple types of performance issues, not just blocking. In a similar spirit to Grade10, Retro [10] does hardware and software resource modelling with usage attribution for efficient resource management in a multi-tenant setting. In comparison, Grade10 offers more fine-grained resources utilization by upsamling to individual execution phases for single-workload bottleneck detection. Closest to Grade10 is work from Tian et al. [11] which also does performance characterization using a DAG-based computation model with system-level resource monitoring. However, Grade10 captures a more comprehensive set of performance issues (e.g., including burstiness, imbalance), does more fine-grained attribution across time and execution phases (in comparison to coarser machine learning based attribution used by Tian et al.), and is more thoroughly evaluated with two state-of-the-art graph frameworks, 2 datasets, and 4 algorithms (than just two workloads by Tian et al.).

Other performance characterization approaches for big-data applications include distributed tracing [26]–[28] and critical path analysis [29], which can reveal many workload-level performance issues, but do not bridge the gap to system-level performance metrics. Benchmarks [13], [30] and comparative performance studies [31]–[33] use high-level metrics, such as makespan and aggregate resource utilization, to draw broad conclusions about the performance of a set of systems. In this work, we focus on characterizing the performance of individual graph applications in depth.

Beyond the big-data-oriented approaches already discussed, there is a rich field of general-purpose performance characterization techniques that can be applied. Traditional profiling and tracing techniques [34], [35] can be used to capture a wide range of performance metrics at machine and source code level. Our approach considers also workload-level attributes and metrics. Recent advances in logging allow for automatic placement of log statements [36], [37], lower latencies [24], and can aid in root-cause analysis [38]. These approaches are complementary to our work as they could make it possible to collect richer performance data with less development effort. State-of-the-art in (distributed) monitoring [39]–[42] enables rich data collection at large scale with low overhead, but does not reach the fine granularity that Grade10 requires.

VII. Conclusion

The current state of distributed graph processing, of increasing presence and demand for performance analysis, requires a new generation of specialized performance tools. In this work we have presented Grade10, a performance characterization framework for this context.

We designed Grade10 to facilitate building fine-grained performance profiles of graph workloads with low resource monitoring overheads, yet useful for performance analysis. Grade10 is a detailed and practical architecture, whose key design features are: (i) a fine-grained, unified workload-level and system-level view of performance; (ii) specialized execution and resource models, which capture in direct acyclic graphs the iterative, irregular nature of graph processing workloads, and both hardware (e.g., CPU, memory, network) and software resources (queues, locks); (iii) a novel resource attribution process, whose upsampling capabilities can compensate coarse-grained monitoring without significant loss of accuracy; (iv) a specialized process to identify resource bottlenecks; and (v) a specialized process to detect performance issues and their maximal impact on system performance, leading to meaningful advice given to users of distributed graph processing.

We are in the process of extending the Grade10 framework to other workloads such as relational data processing and machine learning. Grade10 is open-source:

https://github.com/atlarge-research/grade10

Acknowledgment

This research is supported by the Dutch NWO through project Vidi MagnaData.

REFERENCES

[1] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Surv.*, vol. 51, no. 3, Jun. 2018.

[2] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, Oct. 2015.

[3] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: the digitization of the world from edge to core," *Seagate*, 2018. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012.

[5] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *PVLDB*, vol. 8, no. 12, pp. 1804–1815, 2015.

[6] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, 2014, pp. 599–613.

[7] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, 2015, pp. 410–424.

[8] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 135–146.

[9] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making Sense of Performance in Data Analytics Frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, 2015, pp. 293–307.

[10] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, 2015, pp. 589–603.

[11] H. Tian, Q. Weng, and W. Wang, "Towards framework-independent, non-intrusive performance characterization for dataflow computation," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2019, Hangzhou, China, Augsut 19-20, 2019*, 2019, pp. 54–60.

[12] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An Experimental Comparison of Pregel-like Graph Processing Systems," *PVLDB*, vol. 7, no. 12, pp. 1047–1058, 2014.

[13] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. J. Anderson, I. G. Tanase, Y. Xia, L. Nai, and P. A. Boncz, "LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," *PVLDB*, vol. 9, no. 13, pp. 1317–1328, 2016.

[14] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 1, pp. 5–20, 2007.

[15] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[16] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing breadth-first search," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, J. K. Hollingsworth, Ed., 2012, p. 12.

[17] S. Au, A. Uta, A. Ilyushkin, and A. Iosup, "An elasticity study of distributed graph processing," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, 2018, pp. 382–383.

[18] "Graphite," Accessed on 2020-05-24. [Online]. Available: https://graphiteapp.org/

[19] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Comput.*, vol. 30, no. 5-6, pp. 817–840, 2004.

[20] "Spark – monitoring and instrumentation," Accessed on 2020-05-24. [Online]. Available: https://spark.apache.org/docs/latest/monitoring.html

[21] A. Uta, A. L. Varbanescu, A. Musaafir, C. Lemaire, and A. Iosup, "Exploring HPC and big data convergence: A graph processing study on intel knights landing," in *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, 2018, pp. 66–77.

[22] W. L. Ngai, T. Hegeman, S. Heldens, and A. Iosup, "Granula: Toward fine-grained performance analysis of large-scale graph processing platforms," in *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, ser. GRADES'17. New York, NY, USA: Association for Computing Machinery, 2017.

[23] C. Lai, J. Kimball, T. Zhu, Q. Wang, and C. Pu, "milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 92–102.

[24] S. Yang, S. J. Park, and J. Ousterhout, "Nanolog: A nanosecond scale logging system," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 335–349.

[25] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, 2014, pp. 395–404.

[26] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[27] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 378–393.

[28] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., 2016, pp. 603–618.

[29] M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. C. Dimitrova, S. Wicki, Z. Chothia, and T. Roscoe, "Snailtrail: Generalizing critical paths for online analysis of distributed dataflows," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, 2018, pp. 95–110.

[30] T. Rabl, M. Frank, M. Danisch, H. Jacobsen, and B. Gowda, "The Vision of BigBench 2.0," in *Proceedings of the Fourth Workshop on Data analytics in the Cloud, DanaC 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*, 2015, pp. 3:1–3:4.

[31] O. 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 2016, Taipei, Taiwan, September 12-16, 2016*, 2016, pp. 433–442.

[32] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An experimental survey on big data frameworks," *Future Generation Comp. Syst.*, vol. 86, pp. 546–564, 2018.

[33] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 979–990.

[34] S. Shende and A. D. Malony, "The TAU Parallel Performance System," *IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.

[35] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," in *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, 2008, pp. 139–155.

[36] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 565–581.

[37] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 293–306.

[38] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze, "Troubleshooting transiently-recurring errors in production systems with blame-proportional logging," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 321–334.

[39] A. Agelastos, B. A. Allan, J. M. Brandt, P. Cassella, J. Enos, J. Fullop, A. C. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. T. Showerman, J. Stevenson, N. Taerat, and T. W. Tucker, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 154–165.

[40] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 14–27.

[41] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, 2019, pp. 379–394.

[42] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt, and I. T. Foster, "Fsmonitor: Scalable file system monitoring for arbitrary storage systems," in *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*, 2019, pp. 1–11.

[43] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[44] "GitHub: Grade10," Contains code, documentation, and additional technical specifications. Accessed on 2020-05-25. [Online]. Available: https://github.com/atlarge-research/grade10