OPTiC: Opportunistic Graph Processing in Multi-Tenant Clusters

Muntasir Raihan Rahman
Microsoft
muntasir.raihan@gmail.com

Indranil Gupta
University of Illinois Urbana-Champaign
indy@illinois.edu

Akash Kapoor
Princeton University
kapoor@princeton.edu

Haozhen Ding
Airbnb
hzding621@gmail.com

***Abstract*—We present OPTiC, a multi-tenant scheduler intended for distributed graph processing frameworks. OPTiC proposes opportunistic scheduling, whereby queued jobs can be pre-scheduled at cluster nodes when the cluster is fully busy running jobs. This allows overlapping of data ingress with ongoing computation. To pre-schedule wisely, OPTiC's novel contribution is a profile-free and cluster-agnostic approach to compare progress of graph processing jobs. OPTiC is implemented inside Apache Giraph, with YARN underneath. Our experiments with real workload traces and network models show that OPTiC's opportunistic scheduling improves run time (both at the median and at the tail) by 20%-82% compared to baseline multi-tenancy, in a variety of scenarios.**

***Keywords*-graph-processing, scheduling, multi-tenancy**

I. Introduction

Prominent distributed graph processing frameworks include Apache Giraph [11] (widely used at Facebook), Pregel [25] (used at Google), PowerGraph [14], GraphLab [24], PowerLyra [10], and others. Today these systems are used inside production deployments for computing a variety of metrics (e.g., PageRank, Shortest Paths, Connected Components, etc.) on graphs with millions of vertices and billions of edges [11] (e.g., social networks, financial networks, Web graphs, etc.). Such frameworks divide a graph processing job into two phases: the graph preprocessing phase and the computational phase. The preprocessing phase loads the graph from a distributed file system, partitioning the graph if needed across servers in the cluster. The computational phase runs the actual graph algorithm, and is iterative in nature. It is a known shortcoming of this approach that the first, preprocessing phase constitutes a significant fraction of the total runtime for a job [21], [30], [36].

As the area of distributed graph processing matures, there is an increasing need to support multi-tenancy. Multiple tenants, or users, share the same cluster and run graph processing jobs written using a common graph processing framework. Multi-tenancy has many benefits including: consolidation of resources, loading of input datasets (graphs) from a common file system, higher resource utilization, and reduction of capital expenses (Capex), operational expenses (Opex), and total cost of ownership (TCO). While today one could ostensibly run graph processing engines (like Giraph) atop schedulers (like YARN [2] and Mesos [20]), such a software stack is not (yet) optimized for graph processing in particular.

In this paper, we investigate the first scheduler intended specially for graph processing jobs in a cluster that is shared, over-subscribed, and non-preemptive. We present OPTiC, a new Opportunistic scheduler for Graph Processing on Multi-Tenant Clusters. The key idea is to guess which job J in the full cluster might complete the earliest, and then to opportunistically schedule the next queued (waiting) job at J's machines. This allows us to opportunistically start the graph preprocessing phase of the next job concurrently with the graph computation phase of the currently running job, essentially allowing the next job to start earlier and thus finish earlier. To guess which job(s) might finish first, we use the notion of job progress. We show that: 1) our approach (theoretically) provides optimal run-time performance under certain assumptions, and 2) practically, when these assumptions are relaxed (in our deployments) significant benefits still prevail.

We explored several policies for opportunistic overlapping in OPTiC. The most promising is a new policy that we call Progress Aware Disk Prefetching (PADP). Here we prefetch the graph input of a waiting job onto the disks of server(s) running the maximum progress job (thus creating an additional, though transient, replica of that graph). This way when the new jobs starts it only needs to incur a local fetch from disk to memory, instead of a remote network fetch (as would be the case without opportunistic scheduling). This is an acceptable tradeoff: today, disk bandwidth continues to be higher than network bandwidth. We show measurements in Table I from a private cloud and a public cloud that confirm this. Other recent studies have indicated that this disk-network gap will likely be a continuing trend [4]–[6], [16], [22]. The cost of the OPTiC PADP policy is the increase of the input graph replication factor by (at most) 1 transient replica (per job using that graph). Thus this policy trades off storage for improved job run-time.

While the idea of overlapping computation for multiple jobs has been explored in other domains [2], [12], [13], [20], unique in this paper is the design of techniques specifically intended for graph processing domain, which entails some

This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, a VMware Graduate Fellowship, Yahoo!, and a generous gift from Microsoft.

new challenges. Prominent among them is the challenge of estimating and comparing progress of multiple concurrent graph processing jobs (to decide which is the maximum progress job). To this end we propose a novel progress estimator for graph computation based on the percentage of active vertices in the graph. This allows us to estimate and compare progress in a profile-free and cluster-agnostic manner. Profile-freedom means that we do not track a myriad number of parameters to characterize the profile of each individual job. Cluster-agnostic means that we do not rely on cluster metrics like CPU, memory, network utilization, etc. These properties make our techniques easy to implement, deploy, and debug. Our ideas might be applicable to other (non-graph processing) domains, but exploring these is beyond our scope.

Cloud	Disk	Network
private (Open Cirrus) [8]	157.8	117.7
public (Amazon EC2)	141.5	73.2

Table I: Disk vs. Network bandwidth (MBps) (mean).

Our implementation extends Apache Giraph [1] running atop Apache YARN [2]. Giraph is a popular graph processing system, especially used at Facebook. Neither map-reduce engines [13] nor cluster schedulers like YARN [2] or Mesos [20] are aware of computation running above them, and thus are not graph-aware. OPTiC is essentially a cross-layer approach that makes the graph engine and the underlying cluster scheduler coordinate better with each other.

We report experimental results using realistic network latencies, and on a range of network speeds (1 Mbps to 10 Gbps). Due to the paucity of multi-tenant graph processing frameworks (since our scheduler is the first in this space!) our evaluation instead uses production traces from multi-tenant MapReduce clusters at Facebook and Yahoo!–this is realistic because: i) MapReduce clusters are a good representative of multi-tenant scenarios, and ii) Apache Giraph converts graph jobs to MapReduce jobs anyway. Our experiments show that OPTiC improves job completion time, both at median and tail, under workloads that involve various job sizes, job mixes, network delays, and network to disk bandwidth ratios.

II. Background and OPTiC Summary

A graph processing system performs computation on a graph loaded into one or more servers. The processing consists of two phases: (1) graph loading (and partitioning if needed), from the distributed file system (DFS) into the memory of the computation nodes; and (2) the actual computation, in iterations.

A. Partitioning and Loading

Before computation the graph must be partitioned across the machines in the cluster, and then each machine fetches its own partition from the DFS. There are many partitioning algorithms in literature, from hash-based [21] to edge-cut and vertex-cut [10], [14], [30]. (OPTiC's techniques are agnostic to these.). Facebook stores the graph in HDFS or as Hive tables [11]. Thus the fetching (loading) can involve significant network overhead, and use up a significant fraction of the job runtime, some studies indicating as high as 90% of runtime [21]. The authors in [17] report a 18:1 ratio for load:compute time for Apache Giraph jobs.

B. Computation

Most graph processing systems rely on the synchronous, Gather-Apply-Scatter (GAS) model of computation [11], [14], [21]. In this model, computation occurs in iterations or *supersteps*, wherein vertices gather values from neighbors, aggregate and apply the values to do local computation, and finally scatter the results to neighbors. Supersteps are separated by global synchronization barriers, and a vertex can only see updates from neighbor vertices at the end of a superstep. A vertex can be in either of two states: active or inactive. Depending on the computation, initially a few to many vertices are active, and then as supersteps move along, this number changes. When a vertex has no more pending messages in a superstep (from its neighbors), it becomes inactive. The computation terminates when all vertices become inactive. While some systems provide an asynchronous barrier-free mode where the next superstep can start before the previous one ends, the synchronous mode (with barrier) is more popular because it is reproducible, easier to debug, and predictable. Hence we limit our focus to the synchronous mode.

C. OPTiC Summary

The key idea of our system is depicted in Figure 1. OPTiC opportunistically pre-schedules the next job early (at a server), before the current job running on that server is done. OPTiC does so by picking the running job that is expected to finish the earliest. We use a new metric for comparing progress of ongoing graph processing jobs, and OPTiC picks the running job with the maximum progress. It then overlaps the graph preprocessing (partitioning and loading) phase of the next scheduled job in FIFO order with the computation phase of the maximum progress job. OPTiC does this in parallel on multiple servers, each server(s) concurrently running a job and loading the next job. OPTiC pre-schedules up to one job at each cluster server (if there are sufficient jobs waiting to be scheduled).

The OPTiC framework allowed us to explore different policies for opportunistic overlapping. The best policy we found is called the *progress-aware disk prefetching (PADP)* policy. This policy prefetches the graph of the next queued job onto the disk(s) of the maximum progress job. The cost of PADP is to increase the input graph replication factor by at most 1 transient replica (per job using the graph). When

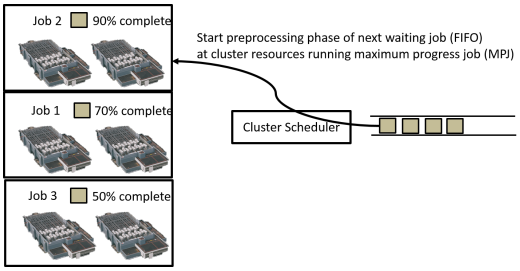


Figure 1: Key idea of OPTiC: overlap the graph preprocessing phase of the next waiting job with the graph computation phase of the maximum progress job.

memory is insufficient to prefetch the graph, PADP still brings the input data closer to the computation by placing it on disk.

In order to measure progress of a job, we eschew both: 1) approaches that involve extensive profiling each job individually, due to its complexity, and sensitivity to jobs, as well as 2) approaches that rely on cluster-based metrics, due to its dependence on the particular distributed scheduler (e.g., YARN, Mesos). Instead, we looked at the temporal variation of the active vertex count percentage (which we call AVCP), and observed that across a wide swathe of graph processing application types, this temporal variation has two phases: an non-decreasing phase followed by a decreasing phase. We leverage this to design a coarse-grained policy to estimate where a job might be in its computation, relying solely on the phase and AVCP value for the job. OPTiC uses this to compare progress of jobs and locate the maximum progress job. This is a less complex approach than full-blown profiling: only one metric (AVCP) needs to be tracked for each job.

III. Exploring the Policy Space

Alternate Policies: The OPTiC framework allowed us to consider several policies, before we settled on Progress Aware Disk Prefetching (PADP). We briefly discuss one of these alternate policies, called progress-aware *memory prefetching*. The main idea was to start fetching the graph input of the next job, directly into the memory of the servers currently running the max progress job (MPJ). Since the cluster is over-subscribed and non-preemptive, typically no free memory is available to store the prefetched graph. We considered setting aside a fixed quota of memory for graph prefetching, e.g., 20% of containers can be set aside for prefetching while the rest continues to be used for computation of other ongoing jobs. However, from the viewpoint of the scheduling challenge, this is isomorphic to the case where only 80% of each server's memory were available for foreground computation jobs (and the 20% quota were considered to be part of the "disk"). A third variant of this approach is to use free memory wherever available– in fact OPTiC does this (overflowing to disk where memory is insufficient).

PADP: PADP prefetches the next job on to disk, leveraging free memory where available. Suppose J_{max} is the current maximum progress job (how to estimate this will be described later, in Section VI). Maximum progress means that among all jobs currently running in the cluster, J_{max} is likely to finish first. Even if we did not use prefetching, the next queued job J_{new} would have naturally been scheduled on the free resources of J_{max}, but only after J_{max} terminates. Instead of waiting for J_{max} to finish PADP prefetches J_{new}'s input graph into J_{max}'s servers right away (unless the input data is already present at those servers). This creates an additional transient replica of J_{new}'s input graph. When J_{max} finishes, J_{new} can start immediately (or soon) by fetching its input from disk into memory. If the network transfer for J_{new} is not complete when J_{max} finishes, then PADP waits for the transfer to finish before starting the computation of J_{new}.

PADP works best if disk bandwidth is higher than network bandwidth. This assumption is consistent with today's datacenter architectural trends. For big data systems, disk locality has been a critical factor that has resulted in significant performance gains for many systems [5], [22]. Although some studies suggest a narrowing of the bandwidth gap [6], [16], [28], others have shown [4] that a gap will remain. In Table I, we showed disk and network bandwidth in a 20 server private cluster [8], and a 20 server virtual cluster on Amazon EC2 [4]. We observe that the mean disk:network bandwidth ratio for the private cluster is 1.34, whereas for the public cloud the ratio is 1.93. (Later, some of our experiments will use similar ratios.)

The OPTiC PADP policy trades storage cost for improved run-time performance. It increases the input graph replication factor from the default (3 replicas) to default (3 replicas) + at most 1 transient replica (per job using the graph). Although we incur increased storage and bandwidth cost, the effect on dollar cost is negligible since: (a) disk is much cheaper compared to main memory devices, and (b) most disk resources are heavily under-utilized in production.

IV. System Architecture

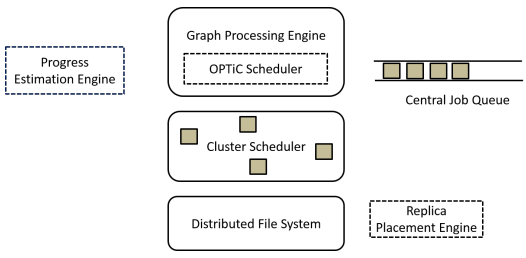


Figure 2: System Architecture (dotted boxes indicate new components from OPTiC).

OPTiC's system architecture is shown in Figure 2. It uses a central job queue, where jobs are queued in FIFO order

waiting to run on the cluster. By default, when the queue is non-empty and the cluster is not full, the graph processing engine scheduler fetches the next job from the queue and submits it to the cluster scheduler (YARN, Mesos).

OPTiC adds three new components, shown with dotted boxes in Figure 2: (1) the Progress Estimation Engine, (2) the OPTiC scheduler, and (3) Replica Placement Engine. The Progress Estimation Engine uses our progress metric from Section VI, and periodically reports this to the OPTiC scheduler, which in turn calls the Replica Placement Engine. Figure 3 summarizes the OPTiC scheduler logic. The OPTiC scheduler first compares progress reports to decide the maximum progress job (MPJ). Second it looks up the current server(s) resources S running the MPJ, and instructs the Replica Placement Engine to place onto S, an additional replica of the first waiting job's input graph. This scheduler check happens periodically.

These actions involve control communication between the graph engine and the cluster scheduler, thus making OPTiC a cross-layer scheduling approach. Later when the MPJ finishes running on S, the underlying cluster scheduler (e.g., YARN) independently schedules that next job to start running on S.

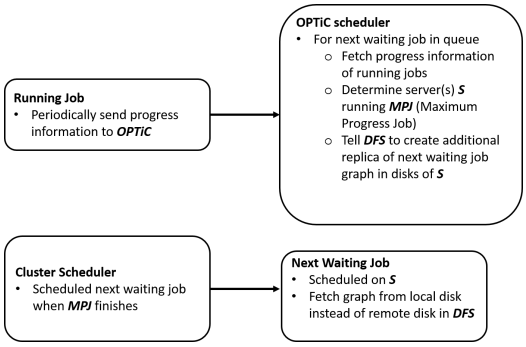


Figure 3: OPTiC Scheduling Algorithm.

V. Optimality of OPTiC

OPTiC's progress-aware scheduling is optimal under certain assumptions. Let $p_i(t)$ denote the progress of job J_i at time t. $p_i(.)$ must be a non-decreasing function of time, with higher values of $p_i(.)$ indicating the job is closer to completion.

Theorem 1: Let $p_1(t), \ldots, p_n(t)$ be the progress metrics at time t for a set of jobs $J_1, \ldots, J_n$ respectively, currently running in an over-subscribed non-preemptive cluster. Without loss of generality assume that: $p_1(t) \geq \ldots \geq p_n(t)$. Assume that: 1) jobs use the same number of servers, and 2) jobs complete (in the future) in the same relative order as their current progress values (i.e., no job overtakes another [1]). In opportunistic prefetching, placing the input data of the next job J_{next} onto the servers of the highest progress job (J_1) minimizes completion time for J_{next}.

Proof: Denote by S_i the set of servers currently running job J_i. The total runtime for J_{next} running on any server set S is $t^S_{total} = t^S_{load} + t_{compute}$, i.e., the sum of graph fetching time and compute time. The value of $t_{compute}$ is independent of whichever $S = S_i$ that J_{next} is assigned to (assumption 1). However, $t^{S_i}_{load}$ may be different across S_i's because there might be a gap between when J_{next} finishes loading and when the computation starts. This gap (which we consider to be part of $t^{S_i}_{load}$) may occur if J_i's remaining processing time exceeds $t^{S_i}_{load}$, i.e., if the prefetching finishes before the current job completes. However, since J_1 has the highest progress $p_1(t)$, this gap will be minimized when J_{next} is assigned to S_1. Other jobs (J_2 and onwards) may catch up to J_1, but they will never overtake J_1 (assumption 2). Thus t^S_{total} is minimized when $S = S_1$. □

[1]Our implementation and experiments relax this assumption.

VI. Progress Estimation Metric

OPTiC's PADP policy needs to rely on a way to estimate progress of a graph processing job. Accurately estimating the progress metric for a job is a potentially complex undertaking, since it could depend on several factors including the graph algorithm being run, the size of the graph, cluster metrics, etc. We first discuss two directions that we pursued initially but found to be inadequate (Sections VI-A, VI-B). Then we present the algorithm that worked best for us (Section VI-C), along with its analysis.

A. Profile-based Approaches

We could profile each job using fine-grained metrics, perhaps customized for the graph algorithm. Past performance of the job (or other jobs) in its class can be used to predict (perhaps via Machine Learning techniques) the progress. This approach has been used for Hadoop and Pig scheduling, e.g., see [35], [38]. However such fine-grained profiling can result in a high overhead as high as 30% [19].

While this might lead to somewhat accurate predictions, we realized that we do not need such fine-grained predictions in order for our PADP policy to work well. That is, the complexity of building, maintaining and updating a profile is not worth the marginal performance gains. Further, such profiling would be sensitive to extraneous factors like the cluster configuration.

B. Utilizing Cluster Scheduler Metrics

For staged dataflow jobs, mature cluster schedulers like YARN and Mesos offer job progress estimators for each stage. We could leverage this, especially for graph processing engines that rely on such staged dataflows, e.g., GraphX [15] and Apache Giraph [1] which are scheduled on Mesos [20] or YARN [2]. (Staged dataflows: GraphX maps Pregel [25] jobs to a sequence of join and group-by dataflow operators; Giraph maps graph jobs to map-reduce stages.)

This approach could work as follows: for two arbitrary jobs J_1 and J_2, we check their current dataflow stage (map/reduce), and percentage complete within that stage, e.g., if J_1 is in map stage, and J_2 is in reduce stage, we say J_2 has closer to completion. If both J_1 and J_2 are in the same stage (e.g., reduce) then the reducer progress metric indicates which job has more progress.

While this is profile-free, it does not generalize to other graph processing systems that do not rely on staged dataflows, e.g., Chaos [30]. However, comparing where jobs are in their progress is a practicable idea, and one that we exploit next, independent of the underlying engine or platform.

C. AVCP

We chose to characterize a job's progress using a single metric. Our metric of choice is the *active vertex count percentage (AVCP)*. The active vertex count (AVC) is defined as the current number of *active* vertices. A vertex is said to be active in a superstep if it will calculate a value in this superstep. Concretely, an active vertex has at least one unprocessed incoming message from the previous superstep (from one of the vertex's neighbors). We define the active vertex count percentage (AVCP) as the active vertex count AVC divided by the total vertex count in the graph. AVCP of a job varies over time, and always lies in the interval $[0, 1]$.

AVCP Evolution: AVCP is attractive because it removes dependency on complex metrics such as nature of the graph algorithm (in the job), input graph size, cluster configuration, cluster size, etc. Our goal is to establish certain common patterns, across jobs, of how the AVCP evolves over time. Using this, we can compare the progress of pair of jobs.

To understand AVCP evolution over time, we ran 8 different (and popular) graph algorithms on a 100 million vertex graph in PowerGraph [14]: Pagerank (PR), Single Source Shortest Path (SSSP), K-core Decomposition (KC), Connected Components (CC), Undirected Triangle Count (UTC), Approximate Diameter (AD), Graph Laplacian (GL), and Graph Partitioning (GP). We picked these because together, they cover up to 97% of algorithms used in practice, according to a recent survey [18]. For instance, PR and AD are representatives of a large general class of graph-level metrics including Betweenness Centrality and Vertex Connectivity. SSSP represents graph traversal algorithms (e.g., BFS, DFS). Graph clustering and community detection are represented by KC, CC, UTC, GL, and GP.

The results are shown in Figure 4. We first make individual observations about each benchmark, and then generalize. For Pagerank and Connected Components in Figures 4 (a) and (d), we observe an initial non-decreasing phase where the AVCP stays at 1, and a second decreasing phase where the AVCP starts to decrease towards 0. For SSSP (Figure 4 (b)), we see an initial phase where AVCP is increasing towards 1, and a second phase, where (like Pagerank) AVCP goes towards 0.

We see that for K-core decomposition (Figure 4 (c)), the AVCP starts at the second phase where it is already moving towards 0, thus it only has a decreasing phase. For the other graph algorithms–undirected triangle count, approximate diameter, graph Laplacian, and graph partitioning [14] (Figure 4 (e))–we observe that the AVCP abruptly jumps from 1 to 0 in one large superstep (because they are similar, only one line is shown). Thus these algorithms only have the second decreasing phase.

AVCP-based Progress Metric: Putting together our observations from these benchmarks, we can capture an arbitrary graph processing job (among the above classes) as a sequence of two phases:

- A *non-decreasing phase (INC)* where the AVCP is near or moves towards 100%. (This phase may be absent.)
- A *decreasing phase (DEC)* where the AVCP moves towards 0%.

The first INC phase may or may not be present (as is the case with KC, UTC, AD, GL, CP in the figures). The DEC phase is always present across all benchmarks.

More active vertices indicate more work left. If a job has AVCP of 100%, then all vertices are active and the computation is far from completion. The termination condition is AVCP=0% after the DEC phase, and any progress metric must be able to measure how close the job is to termination.

There are many ways of deriving progress out of AVCP. We show the reasoning we went through to arrive at our coarse-grained progress comparison technique.

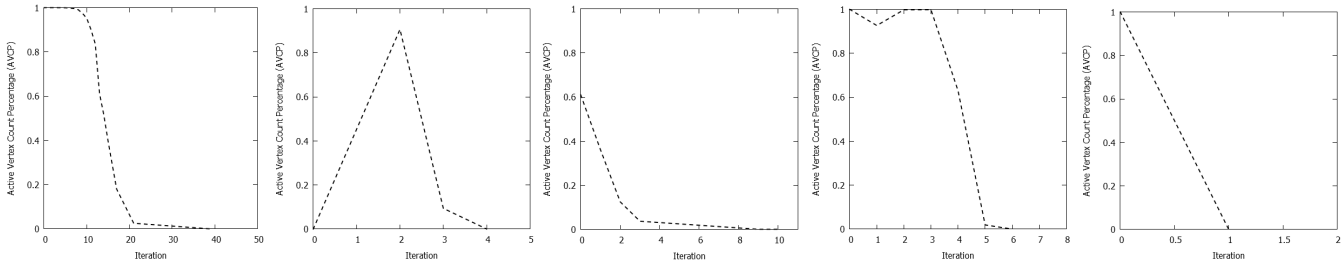
Pure AVCP-Based Progress: One approach is to mathematically derive a progress value from the AVCP value. Let t_s, t_f denote the job start and finish times respectively. Let t_p be the time when the job switches from the INC phase to the DEC phase. The relation between AVCP, denoted by $a(t)$, and progress $p(t)$ at time t can be modeled follows:

$$p(t) = \begin{cases} \frac{a(t)}{2}, & t_s \leq t \leq t_p \\ 1 - \frac{a(t)}{2}, & t_p \leq t \leq t_f \end{cases}$$

While a job is in the INC phase, progress (0% to 50%) grows linearly with AVCP (0% to 100%). When AVCP is 100%, the job can be said to have made half-way progress (50% progress). After the job switches to the DEC phase, AVCP starts to decrease from 100% towards 0%, which indicates further progress from 50% to 100%.

Such a fine-grained progress metric however may violate our optimality result from Theorem 1 (assumption 2) as jobs can overtake each other's progress, e.g., a job J_1 with slightly lower $p(t)$ than job J_2 might overtake J_2 right after a scheduling decision is made. Thus we need to *coarsen* the progress comparisons.

Progress Comparison (Fine-Grained): Our first coarsening explicitly accounts for job phase. Consider two jobs



(a) PR (b) SSSP (c) KC (d) CC (e) UTC, AD, GL, GP

Figure 4: Active Vertex Count Percentage (AVCP) on a 100 million vertex graph against superstep iterations. All applications show a two phase nature: (1) an INC phase where AVCP is monotonically non-decreasing, and (2) a DEC phase with monotonically decreasing AVCP.

J_1, J_2 with respective AVCP values $a_1(t)$ and $a_2(t)$ ($a_i(t) \in [0, 1]$). Three cases arise:

- Case 1: If J_1 is in the initial non-decreasing (INC) phase, and J_2 is in the second decreasing (DEC) phase, then we conclude J_2 is the maximum progress job (MPJ).
- Case 2: If both J_1 and J_2 are in the INC phase $[0\% - 100\%]$, and $a_1(t) < a_2(t)$, then J_2 is the MPJ.
- Case 3: If both J_1 and J_2 are in the DEC phase $[100\% - 0\%]$, and $a_1(t) > a_2(t)$, then J_2 is the MPJ.
- Otherwise: Both jobs have equivalent progress.

However, jobs in the same phase may still overtake each other. Thus further coarsening is needed.

Progress Comparison (Coarse-Grained): Directly comparing AVCP values, as just outlined, is too fine-grained because two jobs with nearby AVCP values (in the same phase) should ideally be viewed as having equivalent progress. Thus, instead of using the fine-grained AVCP values we instead subdivide each phase into *coarsened intervals*. We divide each phase (INC, DEC) into three disjoint equi-sized intervals based on AVCP value: $H = [100\% - 67\%]$, $M = [67\% - 33\%]$, $L = [33\% - 0\%]$. We order the intervals in terms of progress towards AVCP=0% (termination) as follows:

- in the INC phase the ordering is $H > M > L$;
- in the DEC phase the ordering is $L > M > H$.

Here, $A > B$ indicates that A has higher progress than B. Job overtaking could still occur at interval and phase boundaries and could be addressed by increasing the number of intervals, but we found that the marginal benefit of increasing beyond 3 intervals was small.

When two jobs are in the same phase but different intervals (e.g., job 1 in L, job 2 in M), we use the interval ordering to decide the maximum progress job (e.g., job 2 if both in INC, job 1 if both in DEC). When two jobs are in the same phase and interval (e.g., both jobs in M in the same phase), we toss a coin and randomly pick a job as the MPJ. Thus two ongoing jobs with comparable progress will be picked equally likely as prefetch targets. In this way the coarseness of our metric allows us to introduce a desirable element of randomness among equivalent prefetching choices. This randomness also allows us to partially alleviate the error introduced by approximating the non-linear AVCP with two linear segments. It would be interesting future work to compare the performance of our partial random approach with fully randomized cluster scheduling policies like Sparrow [29].

VII. Implementation

We have implemented a scheduler prototype of OPTiC (around 500 LOC) that supports our PADP policy for the Apache Giraph [1] graph processing system running on the Apache YARN scheduler [2].

A. Incorporating Multi-tenancy in Giraph

While Giraph can today be run atop YARN, this does not create a truly multi-tenant graph processing system. Currently, to simultaneously schedule multiple graph processing jobs on YARN, we would need to run multiple distinct Giraph run-time instances, one for each job. This is cumbersome, and as we show via experiments, inefficient ("BASE" datapoints in all experiments of Section VIII).

We modified Giraph such that multiple graph processing jobs can be scheduled using a single Giraph run-time. We use a Java thread-pool for this purpose. We implemented a console for submitting new jobs to the Giraph run-time. The console interface parses every job command and delivers it to the scheduling component. Each newly submitted job is handed off to a new thread in the thread-pool.

B. PADP Policy

Our PADP policy relies on two pieces of information from the graph processing systems: active vertex count (which we normalize to get AVCP) and allocated containers for running jobs. In Apache Giraph, these two metrics are available in the master server. The PADP scheduler aggregates active vertex count at each superstep, and for each job this aggregated value (Section VI-C) is propagated

to a centralized log in the cluster. Also, in Giraph whenever a new container is allocated, we propagate the information to a centralized log in the cluster. We use a push approach to disseminate AVCP and container information: job servers periodically send AVCP and container logs to the central master server.

To implement PADP, the OPTiC Scheduler polls YARN to find the the next waiting (FIFO) job, and the list of currently running jobs. Then for each running job, it retrieves the progress information stored centrally, uses and infers the maximum progress job (MPJ) (Section VI-C). Next, it retrieves the list of allocated container(s) for the MPJ, and creates an HDFS file copy of the input graph of the next queued job at the server running those container(s). These steps are repeated periodically (default period 3 s).

We note two important points. First, OPTiC outsources handling of stragglers and abnormally long tasks to the underlying cluster scheduler mechanisms (e.g., speculative execution and task killing [7], [37]). Second, OPTiC uses a static policy for pre-scheduling resources for the job at the head of the FIFO queue.

C. HDFS Replication

Suppose we have an incoming job J_{next} with input graph G and the maximum progress job J_{max} is running on server(s) S. The Replica Placement Engine (Figure 2) is required to prefetch the input data into the local disks of J_{max}. To create the additional replica, we make an explicit copy (with replication factor 1) of G using HDFS's native file copy feature and place it in the disk(s) of S.

PADP can be enhanced with awareness of input graph location (in HDFS), and this would amplify our gains. However, as we wish to evaluate gains from purely opportunistic scheduling, our current OPTiC implementation only inherits Giraph's native input-aware placement of jobs.

VIII. Evaluation

We evaluate OPTiC via a series of experiments. Our experiments are choregraphed to be progressively realistic: we start from realistic network models and then add on workload traces. Concretely we perform experiments, (1) under realistic network delay models (Section VIII-B); (2) using production Facebook workload trace that has realistic job size distributions (Section VIII-C); (3) on the impact of job heterogeneity (Section VIII-D); (4) using production Yahoo! workload trace containing both realistic job sizes and job arrival time patterns, and a heterogeneous mix of jobs (Section VIII-E); (5) on scalability w.r.t. size, graph sharing among jobs (Section VIII-F); and (6) on impact of disk to network bandwidth ratio (Section VIII-G).

A. Experiment Setup

We performed our experiments on an Emulab cluster with nine 8-core servers, each with 64GB memory, 200GB disk, and running Ubuntu (version 14.04) OS. The average disk bandwidth of the cluster (measured using *hdparm*) is 450MBps, while the average network bandwidth (measured using *iperf*) is 1MBps. (Later in Section VIII-G we show that a lower disk:network bandwidth ratio of 2:1 still gives benefits.) We use delay emulation to mirror the network delays typically observed in a single datacenter environment [9].

We use the PADP policy in Apache Giraph graph version 1.10 [1] running on top of the Hadoop YARN Cluster (version 2.7.0) [2].

We assume all input graph data are stored in Hadoop Distributed File System (HDFS) [33], the default file system used with Apache YARN [2]. We configure one server as the Master running the YARN resource manager and the HDFS name node, while the remaining 8 servers act as slaves and contain YARN node managers, and HDFS data nodes. Each slave has one 8GB memory container, thus the total YARN cluster has 64GB memory for running jobs. The maximum number of concurrent jobs running in this cluster was 8.

Our main performance metric is *job turn-around time (TAT)*, defined as the difference between job finish time and job arrival time in the queue. For our network latency variation experiments (Section VIII-B), we also evaluate the *total completion time (TCT)* = (finish time of last completing job) - (start time of first job). Our cost metric is *replication factor (RF)* of the input graphs stored in DFS.

We compare PADP against BASE, the name we use for the baseline multi-tenant (non-opportunistic) implementation inside Apache Giraph. We measure performance improvement of our PADP (P) policy over the BASE (B) as $\frac{B-P}{B} \times 100\%$.

Since our techniques are not optimized for specific types of graphs, the key parameter is not the degree distribution but the size distribution among the graphs input across jobs. To model this we create graphs with sizes based on job sizes in our traces (e.g., Facebook and Yahoo! traces), and make the graphs large, random, and undirected. Due to existing studies such as by McSherry et al [23], [26], where possible, we try to fit a job on the minimum number of servers, preferring one server per job. In experiments with one application type the default is SSSP (other applications showed similar behaviors).

B. Impact of Realistic Network Delays

Multi-tenant graph processing clusters are typically deployed over a commodity datacenter network. To model this, we use a lognormal distribution to emulate network delay conditions–this distribution choice is motivated by existing work on modeling datacenter delays [37]. The results are shown in Figures 5 (TCT) and 6 (TAT).

We vary the lognormal distribution mean from 1 ms to 4 ms, and the standard deviation from 0.1 ms to 0.4 ms. This results in average network delay values of 2.73 ms, 7.61 ms, 20.1 ms, and 54.6 ms, for lognormal means 1 ms, 2 ms, 3 ms, 4 ms, respectively. We generate a synthetic workload of 10 SSSP jobs, with a constant inter-arrival time of 7 s.

Because this is a microbenchmark we use a fixed graph (25K vertices) for each job.

For the baseline policy, the TCT (total completion time) metric does not vary much. In general, Giraph jobs are mapped to map-only Hadoop jobs, thus without the shuffle phase, the network delay does not impact job completion time much. However the PADP policy shows improvement at all network conditions. The improvement (over baseline) increases from 29% for 2.73 ms average delay (lognormal with mean 1 ms) upto around 52% for 7.61 ms (lognormal with mean 2 ms) and 20.1 ms (lognormal with mean 3 ms) delays. The improvement starts to reduce as network delays become very high, beyond 20.1 ms. We explain this as follows. Stringent network conditions such as delays above 20 ms (atypical in a datacenter) limit the benefits of PADP. Too fast a network also results in less benefit because the disk is not much faster. PADP's highest benefit for TCT occurs if the network has moderate delays.

The job turnaround time (Figure 6) sees consistent benefits independent of network latency.

C. Facebook Workload Experiment

We now use a more realistic workload. First we replace the constant job inter-arrival time distribution with an exponential distribution with mean 7 s, under a lognormal latency with a mean of 3 ms. Next we sample job sizes from a production map-reduce cluster at Facebook [37].

We are not aware of large measurement studies of multi-tenant graph processing clusters. The Facebook map-reduce workload trace is realistic because: i) Facebook trace captures important characteristics of multi-tenant computing, e.g., most jobs are small and short-lived [3], and ii) Giraph treats graph jobs as map-reduce jobs. For instance, a typical social network company workload is to run daily jobs processing that day's worth of data. For community detection [27], after the graph is so clustered into groups, one needs to extract exponentially many subgraphs corresponding to these groups (many short jobs), and run targeted recommendation algorithms on them (many short jobs). Further, it is well-known that a large majority of jobs (in any cluster) are typically small "trial" jobs run by developers. This is reflected in Figure 7 which is derived from the Facebook dataset [37].

For our experiment, we decide job input sizes based on this distribution. For each job, we generate a graph with number of vertices proportional to the number of mappers, scaled up by a factor of 1000. This results in large graphs with several millions of vertices [2].

Figure 8 shows the CDF (cumulative distribution) of job completion time for the first 100 jobs. For the baseline policy, the median job turn-around time is around 140 s, and the quickest job takes about 50 s. For PADP, the median job turn-around time is around 33 s, about 80% jobs complete within 60 s, and the 95th percentile is less than 90 s. Overall OPTiC improves the median turn-around time by 73%, while the 95th percentile turn-around time improves by 54%.

[2]This range captures a large majority of graph sizes in production [32].

D. Impact of Job Heterogeneity

So far our experiments used a homogeneous trace of jobs of the same type. We now see how PADP performs with a *mixture* of different types of jobs on graphs of the same size. We experiment with a trace of 20 jobs consisting of 80% SSSP and 20% K-core jobs, each running on a 50K vertex graph. Job inter-arrival time is Poisson with mean 7 s, and network delay is lognormal with mean 3 ms.

Figure 9 shows that in cases when jobs have vastly differing run-times (K-core takes longer), our progress comparison mis-predicts the max progress job and does not offer too much improvement. However, we show this experiment only to depict the limits of OPTiC – in practical workloads (ensuing sections) the mix of different sizes causes OPTiC to give good benefits as its mis-predictions are outweighed by its benefits.

Next we make the run-time of K-core comparable to SSSP, by using more distributed workers for K-core. The resulting job run-time CDF is shown in Figure 10. Now we see improved performance. Particularly the median turn-around time for PADP improves by 82%, while the 95th percentile turn-around time improves by 70%.

E. Yahoo! Workload Experiment

To capture the realistic mix of jobs in a multi-tenant cluster, we now use a trace obtained from Yahoo!'s Production Hadoop clusters containing hundreds of servers, thousands of job submissions, over several hours. We use three types of jobs: shortest path (SSSP), K-core decomposition (KC), and pagerank (PR). These benchmark types are assigned in a round-robin fashion to the arriving jobs. For each job we set the graph size to be proportional to number of mappers: using a proportionality constant of 1000, this results in significantly larger graphs than the Facebook experiment in Section VIII-C. We injected 1 hour of traces with 300 jobs into our 9-server test cluster configured with 8 containers (each server has 1 8GB container). The delay is lognormal (mean 3 ms).

Figure 11 shows the job run-time CDF for both baseline and PADP policies for the Yahoo! workload trace with a mixture of job types. We observe that the median job turnaround time improves by 78%. Under a real trace, the rate of any mistakes in the comparison algorithm is small enough that the positive cases with significant benefit outweigh the mistakes. This occurs due to the following reason. Unlike the setting of Section VIII-D where the job size was fixed, here we realistically have different job sizes. A majority of the jobs in any real life workload will be short jobs (as is the case with this Yahoo! trace). When multiple

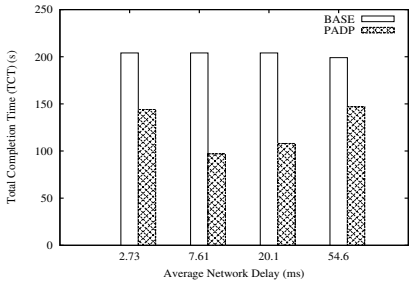


Figure 5: TCT against network latency. PADP improves performance by 26%-52%.

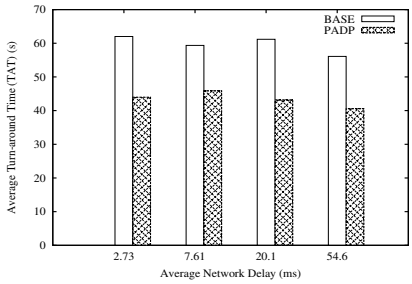


Figure 6: TAT against network latency. PADP improves performance by 22%-29%.

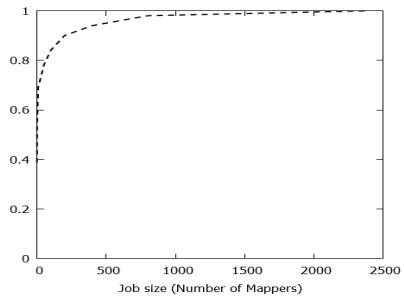


Figure 7: Map count in Facebook cluster [37].

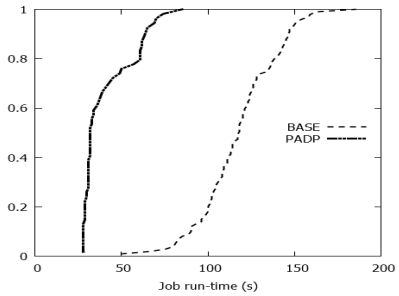


Figure 8: Run-time (s) under Facebook Trace. PADP improves performance by 44-54%.

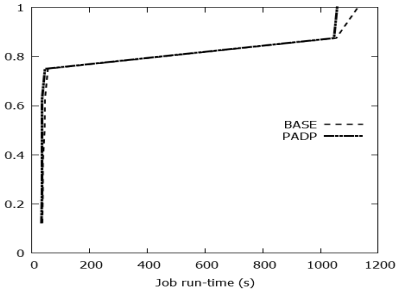


Figure 9: Run-time (s) (K-core 10 times faster than SSSP). PADP improves performance by 6%-8%.

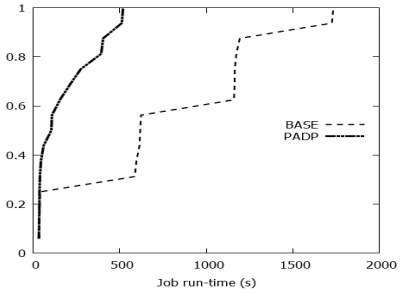


Figure 10: Run-time (s) (K-core and SSSP similar). PADP improves performance by 8%-70%.

short jobs are running simultaneously in the cluster, they have similar run-times and the OPTiC progress comparison algorithm does well.

F. Scalability and Graph Sharing

We explore the effect of larger jobs (graphs) and of jobs sharing the same input graph. To capture the latter we define a parameter called *g=graph commonality*. g ($\in [0, 100]$) is the percentage of jobs that share the same graph input. $g = 100\%$ means that all jobs share the same input graph, while $g = 50\%$ means there are two graphs and jobs are equally split across the two, and so on.

We use a trace of 50 jobs with a Poisson arrival process (mean 7 s). Network delay is lognormal with mean 3 ms. For $g = 100\%$ all jobs use the same graph of size 50K vertices. For $g = 50\%$ we use two graphs of size 50K and 25K vertices. For $g = 25\%$ we use four graphs of size 50K, 25K, 16K, and 12K vertices. For $g = 0\%$, each job gets a unique graph, in sizes from 1K to 50K vertices.

In Figure 12 each box represents the min, 25th, 50th (median), 75th, and 95th percentile values for job turn-around time. We first observe that the job turn-around time grows with increased commonality. This is expected because fewer input graphs imply a bottleneck at the distributed file system to fetch these graphs.

We observe that, 1) the benefits from PADP are highest when there is more job commonality, and 2) PADP is largely insensitive to commonality but the baseline worsens quickly at higher g, due to disk contention at the distributed file system node. PADP mitigates the bottleneck by spreading out the fetching of graphs (via prefetching) over time.

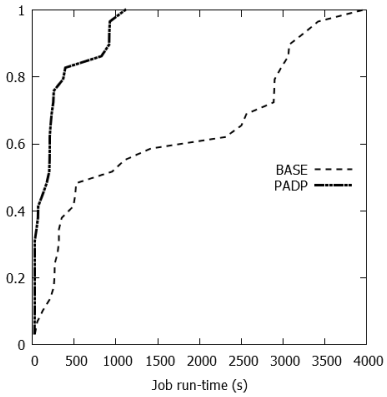


Figure 11: Run-time (s) under Yahoo! Trace. PADP improves performance by 24%-72%.

Since the average job size increases from left to right (for $g = 0\%$, average job size is $25K$; for $g = 100\%$ it is $50K$), the plot additionally indicates that with increased job size the PADP policy also scales much better than baseline.

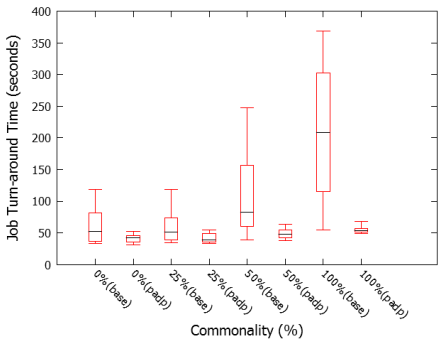


Figure 12: Run-time vs. Graph Commonality. PADP improves performance by 19%-74%.

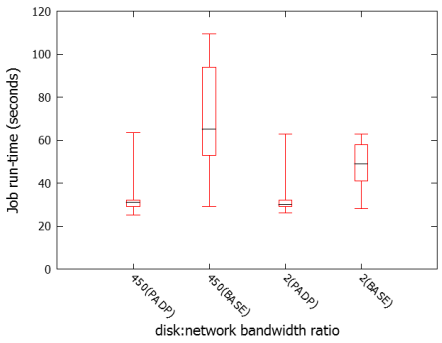


Figure 13: Run-time vs. disk:network bandwidth ratio. PADP improves performance by 38%-52%.

G. Disk:Network Bandwidth Ratio

In all our experiments so far, the disk to network bandwidth ratio was 450:1. To capture the faster networks in today's datacenters [16], we experimented with a 10Gb Ethernet interface, which resulted in a disk to network bandwidth ratio is 2:1 (this closely matches the ratio for EC2 clusters (Figure I)). For this experiment, we use a prefix of the Yahoo! trace (Section VIII-E) using SSSP jobs.

The results are shown using box plots in Figure 13. The Y axis is job run-time, and the X axis varies the disk:network bandwidth ratio for Baseline and PADP policy. The baseline improves when the network is faster (as expected) and PADP improves only slightly in comparison (showing it is more dependent on disk bandwidth than network bandwidth). Yet PADP is still better than the baseline by 38% at the median in the 2:1 disk:network setting - these savings are not significantly lower than the 52% improvement seen at the 450:1 setting. Thus, even as network bandwidth approaches disk bandwidth, PADP improves performance. It would be interesting future work to evaluate PADP on top of an RDMA enabled cluster.

IX. Related Work

Graph Processing Systems: Google designed the first distributed graph processing system called Pregel [25] based on message passing. Subsequently, GraphLab [24] proposed shared-memory-style graph computation. PowerGraph [14] optimizes GraphLab for power-law graphs, using vertex cuts. LFGraph [21] improves performance using cheap hash-based partitioning and a publish-subscribe based message flow architecture. XStream [31] looks at edge centric processing for graphs. Systems have also explored disk optimizations for single host processing in GraphChi [23]. Chaos [30] scales graph processing using disk storage (rather than purely in-memory). Graph mining has been studied in Arabesque [34]–studying progress metrics for such systems is an intriguing future direction.

Scheduling: Multi-tenancy has been explored in the context of cluster schedulers like YARN [2], Mesos [20], and Natjam [12]. Compared to these, we focus on a cluster running graph processing jobs, rather than arbitrary dataflow jobs. Map-reduce schedulers [37] support locality constraints by placing computation near data, whereas OPTiC's PADP policy attempts to move data near cluster resources most likely to be available soon. (This movement is unavoidable due to the way graph processing systems work.)

X. Summary

In this paper we presented OPTiC, a system that opportunistically overlaps the graph preprocessing phase of queued jobs with the graph computation phase of running jobs in order to improve performance in a multi-tenant graph processing cluster. To infer the maximum progress job, we proposed a novel graph processing progress comparison technique. We integrated OPTiC into Apache Giraph and YARN. Experiments indicate that compared to a baseline multi-tenant system, OPTiC improves job completion time by 20%-82%.

REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] Apache hadoop Yarn. https://goo.gl/2gHfmQ.

[3] Cloudera: What do real life hadoop workloads look like. https://goo.gl/hlmpkR.

[4] C. L. Abad, Y. Lu, and R. H. Campbell. DARE: Adaptive data replication for efficient cluster scheduling. In *Proc. IEEE Cluster Computing*, pages 159–168, 2011.

[5] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proc. ACM Eurosys*, pages 287–300, 2011.

[6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. USENIX HotOS*, pages 12–12, 2011.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. USENIX OSDI*, pages 265–278, 2010.

[8] A. I. Avetisyan et al. Open Cirrus: A global cloud computing testbed. *IEEE Computer*, 43(4):35–43, 2010.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of datacenters in the wild. In *Proc. ACM SIGCOMM IMC*, pages 267–280, 2010.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. ACM Eurosys*, pages 1:1–1:15, 2015.

[11] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proc VLDB Endowment*, pages 1804–1815, 2015.

[12] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. ACM SoCC*, pages 6:1–6:17, 2013.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, pages 137–150, 2004.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. USENIX OSDI*, pages 17–30, 2012.

[15] 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 *Proc. USENIX OSDI*, pages 599–613, 2014.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM*, pages 51–62, 2009.

[17] 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 *Proc. IPDPS*, pages 395–404, 2014.

[18] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking graph-processing platforms: A vision. In *Proc. ACM/SPEC ICPE*, pages 289–292, 2014.

[19] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. USENIX NSDI*, pages 295–308, 2011.

[21] I. Hoque and I. Gupta. LFGraph: Simple and fast distributed graph analytics. In *Proc. ACM SIGOPS TRIOS*, pages 9:1–9:17, 2013.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proc. ACM SOSP*, pages 261–276, 2009.

[23] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proc. USENIX OSDI*, pages 31–46, 2012.

[24] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endowment*, 5(8):716–727.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. ACM SIGMOD*, pages 135–146, 2010.

[26] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proc. USENIX HotOS*, pages 14–14, 2015.

[27] M. E. Newman. Modularity and community structure in networks. *PNAS*, 103(23):8577–8582, 2006.

[28] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. USENIX OSDI*, pages 1–15, 2012.

[29] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. ACM SOSP*, pages 69–84, 2013.

[30] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proc. ACM SOSP*, pages 410–424, 2015.

[31] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *Proc. ACM SOSP*, pages 472–488, 2013.

[32] M. Serafini. Graph search: A deceitfully parallel problem. In *Proc. ACM SIGOPS LADIS*, 2017.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. IEEE MSST*, pages 1–10, 2010.

[34] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proc. ACM SOSP*, pages 425–440, 2015.

[35] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic resource inference and allocation for mapreduce environments. In *Proc. ACM ICAC*, pages 235–244, 2011.

[36] S. Verma, L. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. In *Proc. VLDB*, 2017.

[37] 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 *Proc. ACM EuroSys*, pages 265–278, 2010.

[38] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Performance modeling and optimization of deadline-driven Pig programs. *ACM TAAS*, 8(3):14:1–14:28, 2013.