Multi-Task Processing in Vertex-Centric Graph Systems: Evaluations and Insights

Siqiang Luo*
Nanyang Technological University

Zichen Zhu*
Boston University

Xiaokui Xiao
National University of Singapore

Yin Yang
Hamad Bin Khalifa University

Chunbo Li
Nanyang Technological University

Ben Kao
University of Hong Kong

ABSTRACT

Vertex-centric (VC) graph systems are at the core of large-scale distributed graph processing. For such systems, a common usage pattern is the concurrent processing of multiple tasks (*multi-processing* for short), which aims to execute a large number of unit tasks in parallel. In this paper, we point out that multi-processing has not been sufficiently studied or evaluated in previous work; hence, we fill this critical gap with three major contributions. First, we examine the tradeoff between two important measures in VC-systems: *the number of communication rounds* and *message congestion*. We show that this tradeoff is crucial to system performance; yet, existing approaches fail to achieve an optimal tradeoff, leading to poor performance. Second, based on extensive experimental evaluations on mainstream VC systems (e.g., Giraph, Pregel+, GraphD) and benchmark multi-processing tasks (e.g., Batch Personalized PageRanks, Multiple Source Shortest Paths), we present several important insights on the correlation between system performance and configurations, which is valuable to practitioners in optimizing system performance. Third, based on the insights drawn from our experimental evaluations, we present a cost-based tuning framework that optimizes the performance of a representative VC-system. This demonstrates the usefulness of the insights.

1 INTRODUCTION

Vertex-centric graph processing systems (*VC*-systems in the following) are a major paradigm for distributed processing of massive graphs [28]. Well-known VC-systems include Google Pregel [27] and Apache Giraph [5]. The latter has been proven successful in building Facebook's large data-processing platform [9]. In recent years, several new VC-systems have been developed (e.g., Pregel+ [34], GraphD [36]), which achieve significantly improved performance. The main idea of VC-systems is "think like a vertex", i.e., they adopt a local, vertex-oriented perspective of graph processing. In particular, in the beginning, each vertex is only aware of its immediate neighbors, which constitute the initial local view of the vertex. The local view is iteratively updated by executing a user-defined function over vertices of the input graph. The function takes the messages sent from other vertices as input, conducts a local computation, and sends the output packed as messages to a user-specified set of vertices. The computation is executed iteratively for a number of synchronous communication rounds until a user-defined convergence property is met. To achieve high performance, it is critical to minimize the number of *communication rounds* as well as *per-round message congestion* (i.e., number of messages sent within a round). These two objectives, however, are often conflicting in practice. In particular, fewer communication rounds often lead to increased per-round message congestion, forming a round-congestion tradeoff.

*The first two authors are equally contributed and ordered alphabetically.

The Problem: Multi-Processing in VC-systems. Previous studies on VC-systems mostly focus on the optimization for a single benchmark task, e.g., computing the results of a single-source shortest path or a single-source personalized PageRank query. However, in practice, multiple tasks are often bundled and processed in parallel (e.g., [6, 19]), which we refer to as *multi-processing*. Particularly, multi-processing is a collection of unit tasks that can be independently computed. Multi-processing may optionally involve an aggregation of the output of the unit tasks. Multi-processing is common in distributed system settings. Examples include (i) batch Personalized PageRank (BPPR) [23, 24], which involves the computation of multiple single-source personalized PageRanks, (ii) multi-source shortest path distance queries (MSSP) [7], which evaluates multiple single-source shortest path distance queries, and (iii) batch k-Hop Search (BKHS) [37, 38], which collects the statistics of the k-hop neighbors from multiple source nodes. These are fundamental graph-processing tasks that have found a wide spectrum of applications. For example, BPPR has been applied in Pinterest's related Pins [20] and Twitter's Who-To-Follow service [13]; MSSP is useful in diameter estimation [2]; BKHS has been applied in link information analysis in Twitter social networks [37, 38].

Despite the practical importance of multi-processing in VC-systems, to our knowledge this problem has largely been neglected in existing work. As this experimental study demonstrates, there is still significant room for improving the performance of multi-processing in VC-systems, in the following aspects.

A. Inadequacy of Existing Optimization Strategies. At first glance, a multi-processing task could be handled by simply applying the respective algorithms and optimization strategies for its unit tasks. The problem with this approach is that it overlooks the *round-congestion tradeoff*, mentioned earlier. For a single task (e.g., computation of a single-source shortest path), little flexibility is allowed to trade the number of rounds for message congestion (i.e., the number of messages generated per round) or the other way round. In contrast, multi-processing associates an intrinsic tradeoff. Specifically, suppose we need to compute m queries, then we have a large spectrum of round-congestion tradeoff, by computing approximately m/x queries for x batches, where x can be any integer from 1 to m. A larger x incurs less message congestion, at the expense of involving more rounds (Note that a batch may involve multiple rounds, and the number of rounds is positively correlated with the number of batches). Therefore, setting different x's gives us a wide range of design choices that either lean more towards small communication rounds or towards

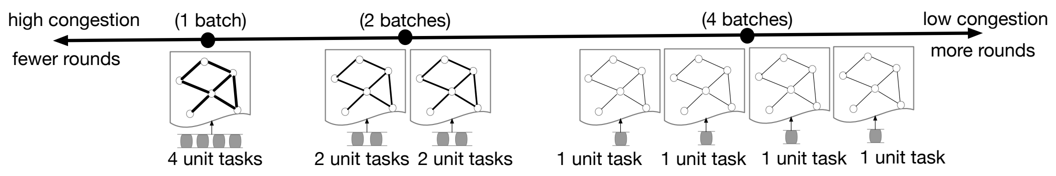


Figure 1: Rounds and message congestion tradeoff in multi-processing. The number of rounds is positively correlated with the number of batches.

little message congestion. Three settings to achieve different tradeoffs between rounds and message congestion for $m = 4$ are shown in Figure 1.

Such a round-message tradeoff impacts the performance significantly. Particularly, we observe that a suboptimal tradeoff may overload the system while a suitable one leads to a highly efficient system. Unfortunately, previous studies pay little attention to this crucial tradeoff. What can further confuse a practitioner is that some relevant choices in existing literature give different criteria for the tradeoff. For example, for a graph of n nodes, the standard congested-clique model [10] proposes to constraint at most $O(\log n)$ bits to be communicated between a pair of vertices in a communication round, while other models (e.g., [32]) allow $O(\log^3 n)$ bits-per-round communication between two vertices. In graphs with $n = 10^9$, $\log^3 n$ is about 430 times larger than $\log n$. Hence these different message congestion constraints imply round-message tradeoffs that differ widely. To fill this gap, we conduct a comprehensive set of experiments to evaluate the factors that impact the optimal round-congestion tradeoffs. The details will be given in Section 4.

B. Lack of General Optimization Strategies. Besides demonstrating the inadequacy of existing optimization strategies, this experimental study also sheds light on the potential of designing a general strategy to determine a suitable tradeoff that works on a variety of typical multi-processing tasks, which, to our knowledge, has not been addressed in previous work. Specifically, for the tuning framework to be general, it must not rely on specific properties of the algorithms for the underlying tasks; instead, the tuning should be performed based on monitoring the system performance measurements, and the strategy should be robust across different tasks. For example, our experimental results show that a suitable round-congestion tradeoff depends on a number of factors, which include the physical memory constraints of the servers, system implementation and optimization strategies (e.g., message mirroring in Pregel+, out-of-core mechanism in GraphD), as well as the characteristics of batch tasks. These heterogeneous factors should be deeply understood to tune hyperparameters to optimize system performance for multi-processing tasks.

Contributions. We have conducted a set of experiments, across seven common VC-systems (or system modes), three representative multi-processing tasks, and three machine clusters with various workloads. By analyzing a broad set of experiment results, we provide a number of observations and insights that are orthogonal to the findings in existing work (e.g., [3, 22]), as follows.

• **High-parallelism Can Be Fragile.** Contrary to common belief, in multi-processing, fewer communication rounds may lead to worse performance. The pursuit of a minimal round complexity alone may lead to significant system latency, as this is often at the expense of a substantial system communication cost or memory consumption. This observation hints towards a somewhat surprising result that "high-parallelism is not always effective". In particular, when a heavy batch workload is assigned to a system with a relatively small cluster, enforcing a degree of parallel-processing (and therefore a small number of rounds) is sometimes suboptimal.

• **One Tradeoff Does Not Fit All.** There is no static round-congestion tradeoff that fits all workload settings. This applies to all combinations of VC-systems and batch tasks that we have tested. The determination of the best round-congestion tradeoff is closely related to the task workloads and system configurations. Behind the dynamics of a suitable round-congestion tradeoff often do we see a number of important factors that play crucial roles. These factors include the physical memory constraints of the servers, the programming languages (e.g., JAVA or C++) of the VC-system, the characteristics of the multi-processing tasks, as well as the system optimization strategies used (e.g., mirroring). Our control experiments are carefully designed to identify the impact of each factor in the round-congestion tradeoffs.

• **Asynchronous VC-systems Can Be Suboptimal.** Asynchronous VC-systems, such as GraphLab [21], often performs better than its synchronous counterpart in processing a single classic task. For multi-processing, however, our experiments show that sometimes asynchronous VC-Systems can be inferior to the synchronous version of GraphLab.

• **Learning-based Approaches Can Help Achieve a Suitable Tradeoff.** Our experimental insights remove some impediments in tuning VC-Systems. We propose a learning-based tuning framework for optimizing multi-processing on Pregel+, as a showcase for the usefulness of the insights. Given an algorithm $\mathcal{A}$ for the unit task of the multi-processing tasks, our framework involves a light-weight training process of $\mathcal{A}$, and outputs an effective concurrency scheme which divides the unit tasks into multiple batches for performance optimization. Our experiments show that the output scheme significantly outperforms the baselines.

2 PRELIMINARIES

2.1 Vertex-Centric Programming Model

In the vertex-centric programming model, the n vertices of a given graph G are partitioned across the machines, and communicate with each other via message passing. Initially, the graph G is loaded into the main memory of the cluster of machines. A user needs to specify a function $compute(v)$ for vertex v. The computation is performed in synchronous rounds. In each round, a vertex v receives the messages (if any) that are sent to it in the previous round. The operations in the *compute* function are typically based on these messages to update the vertex's local information. The output of *compute* is packed as messages, respectively sent to some other vertices specified in the function. Usually, initially each vertex knows only the IDs of its neighbors in G. Through communications, a vertex may learn the IDs of other vertices in the graph. Following existing works (e.g., [15, 32]), we focus on two efficiency measures: (1) The number of communication rounds taken during the program execution, and (2) message congestion, which is the average number of messages sent within a round [1].

2.2 VC-systems

Pregel. The Pregel system [27] is the implementation of the aforementioned vertex-centric programming model inside Google. Pregel is built upon GFS (Google File System) [12]. Besides allowing each user to define the *compute* function, there is also an interface for each user to vote to halt after the execution of the function at each vertex. If a vertex is voted to halt, its state is changed to inactive. When all vertices become inactive, the computation terminates.

Giraph. Apache Giraph [5] started as an open-source implementation of Pregel [27], written in JAVA and built on top of Apache Hadoop [14]. It utilizes Hadoop's MapReduce implementation to process graphs. It has been widely adopted in industry and deployed in-production. For example, Facebook built its Graph Search Services based on Giraph, and contributed to the improvement of the Giraph in the following aspects [9]: (i) fine-grained parallelism with multithreading in each worker machine; (ii) optimized memory consumption by serializing the edges and messages; (iii) split a message-heavy superstep into several sub-steps for message reduction. Giraph has also integrated an asynchronization mode, denoted by **Griaph(async)**, to partially reduce the synchronization cost across communication rounds. The modules of Griaph(async) for message-receiving and message-processing are decoupled into separate threads to reduce resource contention.

Pregel+. Pregel+ [34] is another open-source vertex-centric computation system. One major difference between Giraph and Pregel+ is that the latter is implemented in C++ using MPI. Pregel+ also introduces a new feature called *mirroring*, which is designed to reduce communication costs and eliminate skew in communication. The main idea of mirroring is to construct copies (a.k.a. mirrors) of each high-degree vertex and store them in different machines (called *workers*). Mirrors act as proxies. Particularly, a mirror is created for each high-degree vertex v on all other workers that contain v's neighbor(s). The adjacency list of v is partitioned among its mirrors, where each mirror maintains the sub-list of v's neighbors in its local worker. When forwarding a message from v to its neighbors, the mirror workers (i.e., the workers that contain the mirrors) act as v's proxies such that v first forwards the messages to its mirror workers, who then forward the messages to its neighbors. Since the mirroring mechanism can have significant impact on system performance, in our experiments, we evaluate both the system with mirroring, referred to as **Pregel+(mirror)** and without mirroring, referred to as Pregel+.

GraphD. GraphD [36] is designed for out-of-core execution (i.e., when the server does not have sufficient main memory to store the local graph or message buffers), implemented in C++. GraphD adopts a distributed semi-streaming model. In each machine, the main memory holds a portion of the vertex states, whereas the disk is ready to receive the stream of edges and messages. GraphD also uses multithreading to perform parallel computation regarding the message generation and transmission.

[1]A message contains a constant number of integers.

GraphLab. GraphLab [21] is a representative system that can be tuned to have synchronous or asynchronous program execution. We use GraphLab to refer to the default synchronous mode and **GraphLab(async)** for the asynchronous mode. In asynchronous systems, the vertex execution can happen whenever its input resources are ready. GraphLab(async) introduces a graph-based data model to capture both data and computational dependencies. The dependencies determine the execution order of the vertex execution. The design of GraphLab has inspired other system designs such as [8].

In our experiments we test seven representative VC-system settings, namely, Giraph, Giraph(async), Pregel+, Pregel+(mirror), GraphD, GraphLab and GraphLab(async). As our purpose is to discover multi-processing oriented principles and optimization strategies, we select systems implemented in different programming languages (e.g., Giraph vs. Pregel+), with different optimization strategies on mirroring (e.g., Pregel+ vs. Pregel+(mirror)), out-of-core execution (Pregel+ vs. GraphD), synchronization mechanisms (GraphLab vs. GraphLab(async)). Finally, there are also a few earlier VC-Systems, such as GPS [31]. As their programming models are similar to Giraph and Pregel+, we omit experiments on these systems.

2.3 Benchmark Tasks

A multi-processing job is defined on top of a unit task. Specifically, a multi-processing job composes multiple unit tasks which are processed concurrently and independently. The unit task can be any query required by the application, for example, a single-source shortest path distance query. We consider three common unit tasks as benchmarks: personalized PageRank, single-source shortest paths, and k-hop search. Their respective multi-processing versions are described in the following, which we treat as benchmark multi-processing tasks.

Batch Personalized PageRank (BPPR). Given a graph G, the Personalized PageRank (PPR) with respect to a node s is a classic measure that measures the closeness of other nodes from s. It is *personalized* as it is with respect to a query node. The Batch Personalized PageRanks (BPPR) computes $PPR(s)$ for each node $s \in V$, where V is the node set of G. Here, *batch* indicates multiple PPR queries are processed together. Computing PPRs for any two nodes are independent of each other, and each PPR is approximated by running *α-decay random walks*: with a probability α, the random walk stops at the current node, and with the remaining $1 - \alpha$ probability, the random walk jumps to a neighbor of the current node uniformly at random. The *workload of BPPR* is represented by the number W of random walks that are required to be conducted for each node $v \in V$. In practice, a larger W leads to a more accurate PPR approximation. The space complexity of computing BPPR can achieve $O(n^2)$ instead of $O(n)$, where n is the number of graph vertices. Batch PPR has been used in several important applications including Pinterest's related Pins [20], Twitters' who-to-follow service [13] and Tencent's user ranking service [24].

Multiple Source Shortest Path Distance Queries (MSSP). MSSP is a classic task in graph processing problems [6, 7, 16], and has found applications in route planning and graph diameter estimation. Given a graph G, the single-source shortest path distance query (SSSP) computes for each node the shortest path distances from a given node s to the other nodes. MSSP is a batch version of SSSP, such that given a node set S, MSSP computes for each node $s \in S$ the SSSP from s. The workload of the MSSP is represented by the size of S.

Batch k-Hop Search (BKHS). Given a graph G, a set of source nodes S and a constant k, the batch k-Hop Search (BKHS) task computes for each node $s \in S$, the set of nodes that are within k-hops of s in G. The BKHS is widely applied in link analysis. For example, the works [37, 38] analyze the link information in the graph, and search two-hop neighbors within the ego-network as the friend-recommendation candidates. The workload of the BKHS is represented by the size of S.

2.4 Related Strategies

In this section we review existing strategies for round-congestion tradeoffs, which include BPPA, and Congest. This section discusses why the existing strategies confining message bandwidth in VC-systems may not be suitable for multi-processing.

BPPA. Yan *et al.* [35] propose conditions for defining a balanced practical Pregel algorithm (BPPA): (i) linear space usage: each vertex v uses $O(d(v))$ space, where $d(v)$ is the degree of node v; (ii) linear computation cost: the time complexity of the vertex function is linear to its vertex degree; (iii) linear communication cost: at each round there are at most $O(d(v))$ messages received/sent for each vertex; (iv) at most logarithmic rounds: at most $O(\log n)$ rounds are allowed to finish the computation, where n is the number of vertices in the input graph. Further, Yan *et al.* define the Practical Pregel Algorithm (PPA) as a relaxation of BPPA, by considering the *average-per-vertex cost* instead of *every-vertex cost*. That is, PPA requires that on average each vertex v uses $O(d(v))$ space, costs $O(d(v))$ in its local computation, as well as sends/receives $O(d(v))$ messages. While the authors have shown that it is possible to design PPAs for tasks such as computing List Ranking and Connected Component, we find that chances are lower to successfully design a PPA for a typical multi-processing task. Consider a multi-processing task for a graph of n nodes that requires every vertex to run $\log n$ α-decay random walks, which is the key module to compute the batch Personalized PageRanks. If for each vertex we compute random walks one after another, we will need $O(L \cdot \log n)$ rounds, where L is the maximum walk length and $\log n$ is the number of random walks. Particularly, in a PPA algorithm, each pair of nodes can only exchange $O(1)$ message per-round, which means each round can only handle one random walk step of $O(1)$ random walks. A walk of length-L needs L rounds. We assume that there are $\log(n)$ walks starting at each node and in total there are $O(L \log(n))$ rounds. It can be shown that with probability at least 1-1/n, the maximum walk length is $O(\log_{\frac{1}{1-\alpha}} n) = O(\log n)$, considering that α is a constant. In total, this leads to $O(\log^2 n)$ rounds, violating the condition of logarithmic rounds. On the other hand, if we run $\log n$ walks from each vertex concurrently, each node v would have to send $\Omega(\log n \cdot d(v))$ messages even in the first round, violating the condition of linear communication cost.

Congest. A number of theory papers [11, 17, 18, 29] assume that each pair of vertices are allowed to communicate at most $O(\log n)$ bits. With this constraint, they aim to minimize the number of computation rounds. While these $O(\log n)$-bit constraint is well known in theoretical computer science, it remains open to examine the effectiveness of these models for processing batch tasks in modern vertex-centric systems. Also, we note that the tasks considered in these papers are not multi-processing tasks. For example, they focus on the tasks such as computing the Maximum Independent Set [11, 17], Sorting [18] and Graph Coloring [29].

3 IMPLEMENTATIONS

The vertex-centric algorithms we implemented in the VC-systems fall into two categories: the Pregel-based and the mirror-mechanism-based. In particular, Giraph, GraphD and the basic Pregel+ systems follow the Pregel-based algorithm, while the Pregel+(mirror) system follows the mirror-mechanism-based algorithm.

Pregel (BPPR). We employ the standard Monto-Carlo method for computing Personalized PageRank. We initialize W α-decay random walks and after all walks terminate, we estimate the PPR for each node u as the portion of the random walks that stop at u. The implementation of this algorithm on the Pregel+ system is described as follows. Suppose each node is required to generate W α-decay random walks. The computation is done by simulating the W α-decay random walks. In the implementation, each round corresponds to one random walk step. In the first round, each of the W walks stops with α probability and with the remaining $1-\alpha$ probability randomly selects a neighbor. A message, which contains the source node ID of the walk, is sent to that selected neighbor. For the subsequent rounds, each node at the beginning receives the source IDs of some walks and the process repeats. The process ends if every walk stops. Hence, a larger W indicates more walks passing through a vertex, and vise versa.

Pregel-Mirror (BPPR). The implementation of the BPPR in the Pregel(mirror) system is substantially different from that on the basic Pregel+, because Pregel(mirror) only supports the broadcast interface. That is, whenever node v wants to send a message to a neighboring node u, it has to also send this message to any other neighbors. Under this mechanism, the implementation of a random walk step has to send out more messages than necessary. For example, consider a walk step from v_1 to v_2, using the broadcast interface, v_1 has to send to all its neighbors a common message, which is supposed to be sent only to v_2. Also, the message may need to contain the receiver ID so that every receiver of the message can properly handle the message. To reduce the message size, we employ a generalized random walk which is similar to the *forward push operation* employed in [4]. Particularly, when we disseminate τ walks from node v_1, each of v_1's neighbors receives a common message, which indicates that the number of random walks received at that particular neighbor is $\frac{\tau \cdot (1-\alpha)}{d(v_1)}$. This operation can be interpreted as that the random walk is fractionalized according to the number of neighbors and each walk fraction "jumps" to neighboring nodes. This operation over all nodes can be implemented in one round. Since this generalization does not change the expected portion of random walks that stop at a node, the estimation is still unbiased. It also suits the broadcast interface better.

Pregel (MSSP). Computing multi-source shortest path distance queries requires maintaining the shortest distances between node s from the source set S and any other nodes. We let (u, v, d) denote a message which indicates the existence of a length-d path from source u to target v. In the first round, for each source node $s \in S$ and each neighbor v of s, s sends a message $(s, v, d(s, v))$ to v. In each of the subsequent rounds, the messages received by node v are aggregated such that if there are multiple messages that have the same source and target, only the message with the smallest length is retained. As such, the shortest path is always recorded.

For each retained message (u, v, d) and each neighbor w of v, a message $(u, w, d + d(v, w))$ is sent from v to w, which indicates that there is a path of length $d(v, w)$ from u to w. The process ends if in one round no shorter paths are found compared with its last round of computation.

Pregel-Mirror (MSSP). The MSSP algorithm for basic Pregel+ can be slightly modified to work for the broadcast model. In particular, the message (u, v, d) sent from u to each of its neighbor v, can be broadcast with a message (u, d). This suffices to implement the MSSP in Pregel+ with mirror mechanism.

Pregel (BKHS) and Pregel-Mirror (BKHS). The implementations of BKHS are similar to those of MSSP except for the termination condition. In BKHS, the program stops after $k + 1$ communication rounds.

4 EVALUATING MULTI-PROCESSING

We evaluate how various factors affect the optimal round-congestion tradeoff for multi-processing. These factors include the physical memory cost, disk utilization [2], workload distinctions, graph data, number of machines, memory size, as well as the implementation, which can be categorized into three types:

• **Workload characteristics** that describe the inputs into the VC-system such as the workload and graph data.
• **Runtime system parameters** that vary during task execution. Examples include memory cost and disk utilization, which can change continuously when the VC-system is running.
• **Static system parameters**, which include the number of machines, memory capacity per machine, and implementation. The purpose of our experiments is to evaluate the relationship among these three categories of factors and how they eventually affect the optimal tradeoff.

Experiment Setup. Our evaluation is done on 1) seven representative VC-systems: Giraph, Giraph(async), Pregel+, Pregel+(mirror), GraphD, GraphLab and GraphLab(async); 2) three benchmark tasks: BPPR, MSSP and BKHS; 3) six widely used public graph datasets, and 4) three clusters. A summary is given in Table 1. Among the datasets, Web-St is a web graph published by Stanford. DBLP is a co-author network published by DBLP. All the other data are social graphs generated by different social network services. These datasets are widely used to benchmark the performance of graph algorithms and systems [19, 23, 24, 33]. All the datasets can be downloaded from SNAP [1]. The three clusters include an 8-machine local cluster (referred to as Galaxy-8), a 27-machine local cluster (referred to as Galaxy-27) and a 32-node cloud-based cluster (referred to as Docker-32), where all nodes are initialized with the same Docker image. Specifically, Galaxy-8 consists of 8 Linux machines, each with 16GB memory, 8 Intel(R) Core(TM) i7-3770 CPUs @ 3.40GHz, and HDD disks. Galaxy-27 has the same setting with Galaxy-8, except that it connects 27 machines. Docker-32 consists of 32 Linux nodes, each with 16GB memory and 15 virtual cores of Intel(R) Xeon(R) CPU E5-2637 v2 @ 3.50GHz, and SSD disks. We follow the default settings of each VC-system regarding graph partitioning, which has been done internally by each system. For example, GraphLab partitions the graphs by edges and the cut along vertices. Pregel+ uses random hash on vertices to partition the graphs.

[2] The percentage of the time the hard disk drive is performing at least one operation.

Table 1: Experiment Settings (K=10^3, M=10^6, B=10^9)

	Name	#Nodes	#Edges	d_{avg}	Source
Datasets	Web-St	281.9K	2.3M	8.2	stanford.edu
	DBLP	613.6K	4.0M	6.5	dblp.com
	LiveJournal	4.0M	34.7M	8.7	livejournal.com
	Orkut	3.1M	117.2M	36.9	orkut.com
	Twitter	41.7M	1.5B	35.2	twitter.com
	Friendster	65.6M	1.8B	46.1	snap.stanford.edu

	Name	# Machines	Memory	Type
Clusters	Galaxy-8	8	16GB×8	local
	Galaxy-27	27	16GB×27	local
	Docker-32	32	16GB×32	cloud

	Name	Synchronous	Out-of-core
VC-systems	Giraph	yes	no
	Giraph(async)	partial	no
	Pregel+	yes	no
	Pregel+(mirror)	yes	no
	GraphD	yes	yes
	GraphLab	yes	no
	GraphLab(async)	no	no

Workloads and Evaluation Metrics. The workload for each task has its own definition due to the distinctions of task properties. The workload for BPPR is defined as the number of random walks starting at each node. In contrast, the workload for MSSP or BKHS is the number of source nodes involved. To explore the round-congestion tradeoff, the workload is divided into batches, which are fed into the system sequentially with the workload within the same batch concurrently processed. We name the batching mechanism k-batch if the workload is divided into k equal batches. In particular, the 1-batch mechanism is referred to as *Full-Parallelism* as it requires all unit tasks to be processed concurrently. We also evaluate the performance of processing the workload divided into unequal batches. We report the running times of the multi-processing tasks. We mark a result as *overload* when the task cannot be finished within 6000 seconds. To avoid dense bars in a figure, we employ the *doubling* numbers, i.e., $\{1, 2, 4, 8, 16\}$, for the numbers of batches. For all our results, such settings are able to clearly plot the trends and narrow down to small ranges where the exact optimal batch locates. In the following discussions, we refer to the optimal batch as the optimum among the doubling batches (We include more results for batch settings with finer granularity in our additional materials [25]).

4.1 Full-Parallelism Can be Suboptimal

Contrary to the common belief, our experiments show that optimizing towards the smallest number of rounds (and hence Full-Parallelism) does not necessarily lead to the best performance. Figure 2 shows the execution times of running BPPR on the DBLP dataset with Galaxy-8 under various batches/workload/algorithms. We see that, for example, a system using Full-Parallelism typically runs significantly slower than those based on other settings. This contradicts a general intuition that fewer communication rounds lead to higher performance for a VC-system. We emphasize that

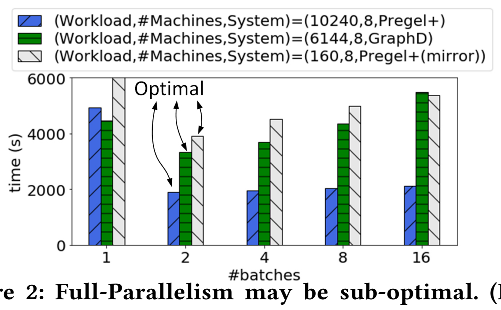


Figure 2: Full-Parallelism may be sub-optimal. (DBLP, Galaxy-8)

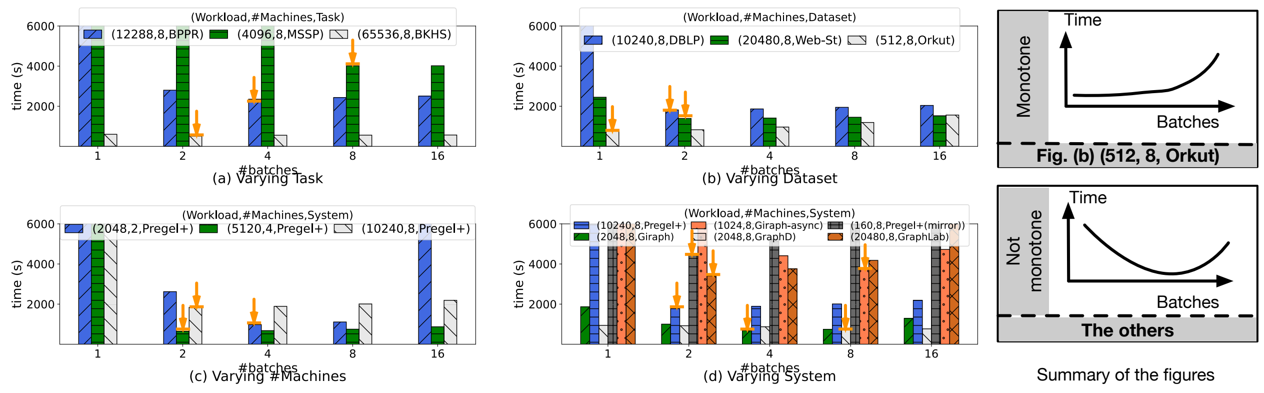


Figure 3: Various experiments on Galaxy-8. The yellow arrows point to the best batches. The default dataset, task and system are DBLP, BPPR and Pregel+ unless otherwise specified. The summary figure on the right demonstrates that the running times mostly are not increasing with the number of batches.

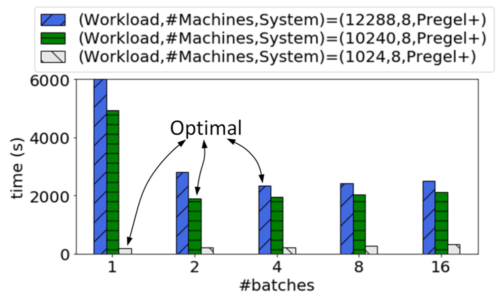


Figure 4: Optimal batching is workload-dependent. (DBLP, Galaxy-8)

this phenomenon is *not unique* as it is commonly observed regardless of the use of out-of-core design (i.e., both Pregel+ and GraphD, as shown in Figure 2), mirroring mechanism (i.e., both Pregel+ and Pregel+(mirror), as shown in Figure 2), cluster sizes (e.g., using different numbers of machines, as shown in Figure 3 (c), Figure 5 (c) and Figure 7 (c)), different implementation (e.g., Giraph, Pregel+, and GraphD, as shown in Figure 3 (d) and Figure 5 (d)), different benchmark tasks (e.g., BPPR, MSSP and BKHS, as shown in Figure 3 (a), Figure 5 (a) and Figure 7 (a)), different synchronization mechanisms (e.g., Figure 3 (d)), or different datasets (e.g., Web-St and DBLP as shown in Figure 3 (b) and Figure 5 (b)).

To our knowledge, the phenomenon mentioned above has not been brought to light before. Meanwhile, it has a crucial impact on VC-system optimization. For example, there exist cases that the system is overloaded when Full-Parallelism is applied (as shown in Figure 3 (c) 1-batch). In the following experiments, we aim to identify reasons for the phenomenon. By revealing the factors that impact the optimal round-congestion tradeoff, we provide new insights for system optimization.

4.2 Workload vs. Optimal Batches

Figure 4 shows the results of running BPPR on DBLP with Pregel+ deployed on Galaxy-8. The results show that a larger workload often favors more batches. In particular, with workload 1024 (random walks per node), Full-Parallelism achieves the best performance. In contrast, a workload of 12288 (random walks per node) favors the 4-batch setting, and a workload in-between (i.e., 10240) reaches its best performance with the 2-batch setting. Hence, these results suggest that a higher amount of workload tends to require more batches to reach the optimal performance.

To further investigate the results, Figure 6 shows the detailed statistics for Figure 4, i.e., the number of per-round messages sent with various workloads and batching settings. With light workload (e.g., 1024), the number of messages delivered per round is relatively small (63.7M for 1-batch). This number becomes roughly ten times larger (633.2M) if the workload increases by ten times (10240). However, the running time goes up super-linearly from 173.3s to 6641.5s, meaning a certain congestion threshold is met while increasing the workload. Once this happens, the cost due to the heavy congestion starts to dominate the overall cost. As an evidence, we observe that when we divide the workload into two batches, an increase of workload from 1024 to 10240 will roughly increase ten times both message congestion and running time. Hence, when we divide the workload into two batches, the congestion per-round is then reduced, and the system is protected from hitting the congestion threshold. To further verify our idea, we increase the workload 12 times from 1024 to 12288. We observe that the running time for 2-batches rises more than 12 times (from 178.3s to 2826.6s), indicating that dividing the workload into two batches cannot prevent hitting the threshold at this high workload, and the number of batches needs to be even higher for an optimal setting.

4.3 Memory vs. Optimal Batches

The relationship between workload and optimal batching scheme is built under the constraint on the memory cost. Particularly, a VC-system can be *memory-bound*, i.e., the optimal number of batches is determined by the run-time memory consumption.

For example, Figure 3 (c) shows the results of performing BPPR on relatively small number of machines (i.e., 2, 4, 8). The green bar indicates that Full-Parallelism overloads the system by the excessive messages transmitted between machines. The reason is that excessive messages cause the memory consumption to exceed the machine's physical memory capacity, thereby either triggering the virtual memory mechanism which leads to high latency, or causing a system failure due to overload. We refer to the state when the system uses up the memory as the memory-bound state. A similar conclusion can be drawn for a larger cluster (i.e., 27 machines), as shown in Figure 5 (c).

General Tradeoffs. We then consider how to tune the number of batches for a desirable memory cost so as to optimize the system performance. Table 2 shows the results of the memory cost and computation cost by varying workloads, number of batches and machines. With more machines/less workloads, the average

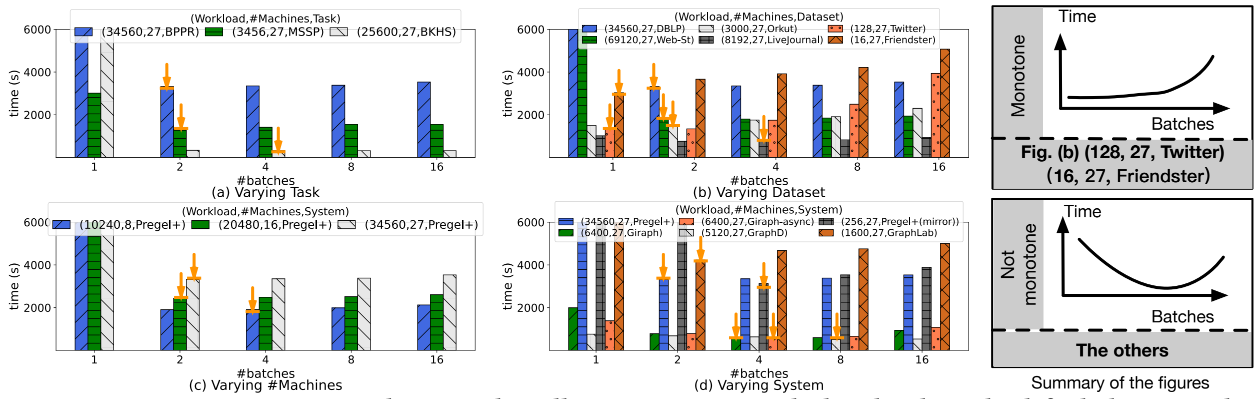


Figure 5: Various experiments on Galaxy-27. The yellow arrows point to the best batches. The default dataset, task and system are DBLP, BPPR and Pregel+ unless otherwise specified. The summary figure on the right demonstrates that the running times mostly are not increasing with the number of batches.

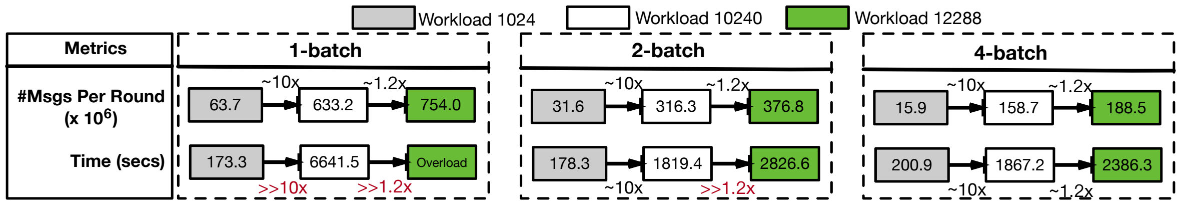


Figure 6: Statistics of Figure 4, illustrating that the processing time is not linear with the number of messages per round.

Table 2: (workload, #batches, costs per machine)

Workload	Batches	Memory/Time/Network Overuse Time	
		4 machines	8 machines
1024	1	4.3GB/6.3min/2.2min	2.1GB/3.4min/1.2min
	2	3.6GB/6.7min/2.2min	1.8GB/3.6min/0.9min
	4	3.0GB/7.4min/1.8min	1.6GB/3.9min/0.6min
4096	1	15.0GB/30min/9min	7.6GB/13min/5min
	2	12.1GB/24min/9min	5.8GB/13min/5min
	4	9.6GB/25min/9min	4.7GB/14min/5min
12288	1	Overflow/Overload/–	15.1GB/Overload/–
	2	Overflow/Overload/–	15.1GB/51min/15min
	4	15.1GB/Overload/–	12.4GB/39min/15min

memory used in each machine decreases. In the meantime, using more batches reduces the memory cost. For example, when we increase the workload from 1024 to 12288 (i.e., 12 times larger), the average memory consumption increases from 3.0GB to 15.1GB (for 4 machines, 4 batches). Also, for a particular workload 4096, the average memory consumption drops from 15.0GB to 9.6GB when we increase the number of batches from 1 to 4. Interestingly, the optimal batch setting happens when the amount of memory used per machine is *close to* the machine' usable memory capacity ($\approx$ 14GB). The remaining memory has to be allocated for bootstrapping the operating system and necessary applications. For example, for workload 4096 with 4 machines, using 15.0GB memory (i.e., with 1-batch) exceeds the usable memory capacity and 9.6GB memory (i.e., with 4-batch) is much smaller than the usable memory. Hence, the performance of 1-batch and 4-batch settings are not as good as 2-batch, which uses about 12.1GB memory. In addition, we also evaluated the network overuse time, which is the duration of the time when the maximum network bandwidth is met. Our results show that the variation of the average overuse time is insignificant compared with the changes of the overall running time. With more batches, the network overuse time sometimes decreases while the overall time increases. This hints that the memory consumption is a more important factor than network overuse, regarding the performance of multi-processing in memory-bound systems. To conclude, our observations imply the following optimization strategy with respect to the memory constraint.

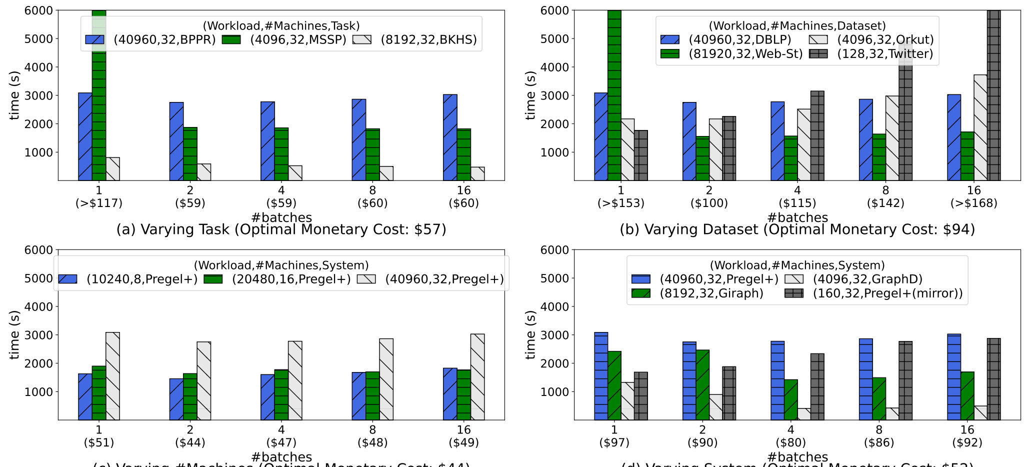
Optimization strategy (memory-bound). *We aim at the minimum number of batches that does not let a machine use up its usable physical memory.*

4.4 Disk Utilization vs. Optimal Batches

The out-of-core VC-systems are not subject to the memory constraints as they can explicitly control the maximum memory being used. However, they can be disk-bound. The disk utilization, i.e., the percentage of CPU time when the disk is performing I/O operations, measures how busy the disks are while data are streaming to the disks. A VC-system may require a large number of I/Os and thus the disk utilization becomes a major concern in choosing the optimal number of batches. If the disk bandwidth is close to fully occupied (i.e., close to 100% utilization), the messages will be streamed to the disk queue until more disk resources are available, causing significant delays. The GraphD

Table 3: #Batches v.s. Disk Utilization v.s. Network (27 Machines with Workload 2048 in GraphD)

#Batches	Overuse Time		Max Disk Utilization	I/O Queue Length	Total Time
	Network	I/O			
1	94s	176s	$\geq$ 100%	20256	285s
2	98s	38s	$\geq$ 100%	3981	236s
4(OPT)	**84s**	**0s**	**27%**	**19**	**201s**
8	66s	0s	27%	20	220s
16	44s	0s	24%	24	260s
32	8s	0s	27%	30	337s
64	8s	0s	27%	39	429s
128	6s	0s	26%	61	632s



(a) Varying Task (Optimal Monetary Cost: $57)
(b) Varying Dataset (Optimal Monetary Cost: $94)
(c) Varying #Machines (Optimal Monetary Cost: $44)
(d) Varying System (Optimal Monetary Cost: $52)

Figure 7: Performance and monetary costs in the cloud (Docker-32). The default dataset, task and system are DBLP, BPPR and Pregel+ unless otherwise specified.

system, for example, writes excessive messages whose total size is greater than a predefined memory budget, thereby gaining the benefit when the messages cannot be entirely stored in memory. A side-effect of this mechanism is that the disk I/Os become the performance bottleneck, and the system runs into a disk-bound state if the disks are fully used. Given a particular workload, the number of per-round disk I/Os increases when fewer batches (i.e., fewer rounds) are considered. Along the round-congestion tradeoff, there is a state where the disk utilization is 100%. Our experiments show that further optimizing the communication rounds after running into the disk-bound state will significantly worsen the system performance. To illustrate, Table 3 shows disk utilization statistics under various numbers of batches for a workload of 2048. The overuse time (I/O) is the duration when the disk utilization is 100% and the I/O queue length describes the average number of messages waiting to be streamed to the disk. Our first observation is that when only a small number of batches are used, the disk utilization can be up to 100%. For example, both 1-batch and 2-batch result in 100% disk utilization. When we increase the number of batches, the disk utilization starts to drop and then stays relatively stable. For example, the disk utilization first drops to 27% for 4-batch and then remains stable even if we further increase the number of batches to 128. One crucial observation is that the total cost can be significantly reduced if the disk utilization drops from 100% to some value below 100%, implying that disk utilization equaling 100% is an indicator when we should stop further optimizing the communication rounds. Also, if we further increase the number of batches, the running time can increase because of the round-synchronization overheads. We also tested the overuse time (network), the duration when the maximum network bandwidth is met. With the increase of the batches, the overuse time drops because less messages per-batch are generated and sent among machines. While this can partially impact the performance, we conjecture that the dominating factor owes to the disk bottleneck. To explain, when switching from 1-batch to 4-batch, the network overuse duration remains relatively stable, while the I/O overuse duration significantly drops from 176s to 0s. The combination effect ultimately makes 4-batch the optimal setting. In a nutshell, we have the following optimization suggestions for disk-bound systems.

Optimization strategy (disk-bound). *For out-of-core VC-systems, we minimize the number of batches until per-batch parallelization incurs %100 disk utilization.*

4.5 Large Graphs on Docker-32 and Galaxy-27

Our next set of experiments focuses on evaluating the billion-edge Twitter graph on Docker-32 and Galaxy-27. Figure 8 reports the running times for the various tasks (BPPR, MSSP, BKHS) and workloads on Twitter in Docker-32. We observe that Full-Parallelism is optimal for BPPR when processing a relatively small workload. This is because even when we set a small workload, the number of messages generated is still large as it is proportional to the number of graph nodes. A further investigation on the underlying statistics reveals the impact of the *residual memory cost*, i.e., the memory cost to hold the intermediate results computed by the previous batches. Such residual memory cost significantly impacts the optimal batches especially when the graph is large. For example, conducting BPPR on the Twitter graph incurs a large residual memory cost due to the huge intermediate results whose size is proportional to the number of nodes and per-batch workload. At the beginning of the h-th batch (for $h \geq 2$), the highest memory cost is the residual memory cost from the previous batch plus the memory used for holding the messages at the current round. Such a high memory cost overloads the system. Using only one round (i.e., Full-Parallelism), interestingly, can avoid the high cost because the peaks of residual memory and memory holding the messages happen at different times (the maximum residual memory is generated at the end of a round, and the maximum memory cost for sending messages is at the beginning of a round). Similar behavior is observed in Galaxy-27 (see Figure 5 (c)).

Note that if the residual memory is small compared with the memory cost by messages, then our previous discussion still applies. For example, when performing MSSP in the Twitter dataset, the memory cost for intermediate results can be small since it is mostly affected by the workload (i.e., the number of source nodes). In this case, we again observe that Full-Parallelism can be suboptimal, as shown in Figure 8 (the middle bars).

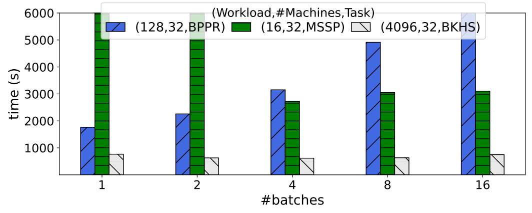


Figure 8: Different tasks on Twitter dataset in Docker 32.

4.6 Reducing Monetary Cost in the Cloud

We demonstrate that such a batch scheme optimization gives significant economic benefits in the cloud. In the Docker cloud, the monetary cost is positively correlated to the running time. The cost per-unit-time is determined by collectively considering the disk cost, memory cost, and CPU cost. We report the monetary cost in Figure 7, where the cost below each setting in the x-axis sums up the credit-costs corresponding to each experiment based on that setting. In the caption of the figure, we also show the optimum cost, which is the best possible total cost if we can optimize each workload setting individually. When the system is overloaded, we set a monetary cost with respect to the cut-off running time (i.e., 6000 seconds). As this is a lower bound, and in practice the cost would be significantly higher than that, their costs would be marked starting with '>'. The results show that an ill-setting can waste a lot of resources (longer running time) while incurring much higher monetary cost. For example, in Figure 7 (b), 1-batch setting or 16-batch setting would incur unpredictable cost, which is significantly larger than 150 credits, where the optimal credit cost is only 94. This shows evidence that optimizing the batch scheme immediately implies a cloud budget optimization.

4.7 Unequal Batches Can Be Beneficial

Previously, we assumed equal batches for an easier exploration of the general round-congestion tradeoff under the complex system environment. Here we further investigate the impact of unequal batches on the system performance. We test the BPPR task on Galaxy-8 and Galaxy-27, and divide a fixed workload W into two batches W_1 and W_2, with varying $W_1 - W_2$. For the round-congestion tradeoff to be effective, the total workloads ($W_1 + W_2$) are set to be 12800 and 40960 for Galaxy-8 and Galaxy-27, respectively. The results are shown in Figure 9, where the left and right figures are for Galaxy-8 and Galaxy-27, respectively. In each figure, two bars are grouped together where the left bar reports the total running time for each setting, and the right bar stacks the running times of executing workload W_1 and workload W_2 separately.

As shown in Figure 9 (a), with the increase of the $\Delta(= W_1 - W_2)$ from -10240 to 10240, the overall performance (left bar in each group) achieves the optimum around $\Delta = 2560$. A higher cost is incurred with a larger $|\Delta|$ value. This is because a heavier batch leads to a higher cost, as evidenced by the trends indicated in the right bars in each figure.

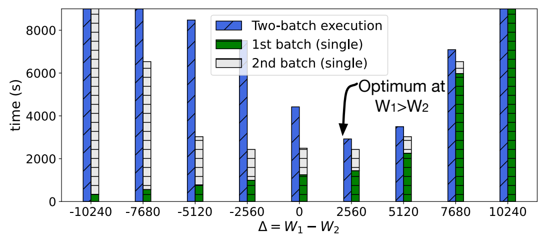
One interesting observation is that the overall running time can be much more significant than the accumulated running time of executing W_1 and W_2 alone. This also leads to an observation that the optimal performance happens when $W_1 > W_2$ instead of $W_1 = W_2$. For example, in Figure 9 (a) left bar, setting $\Delta = 2560$ gives the best overall performance. The reason is that combining the two batches into a single task requires to store the intermediate random walk results of the first batch for later use, whereas treating them separately means no intermediate results between them. The additional memory cost to store the intermediate results, called *residual memory*, indicates that less idle memory is available for the second batch. Therefore, giving a higher workload in the first batch balances the memory consumption in two batches, leading to a better performance than equally dividing the workload.

4.8 Asynchronous VC-systems Can Be Worse

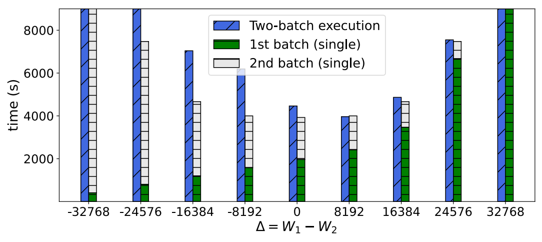
One trend in improving VC-systems is to incorporate asynchronous communication protocols among machines to replace the synchronization barrier at the end of each communication round. VC-systems with such asynchronous protocols are called asynchronous VC-systems. One representative system is GraphLab, which can be tuned to be asynchronous or synchronous. We denote the two modes by GraphLab(sync) and GraphLab(async), respectively. Being asynchronous, GraphLab(async) can have its vertex function executed immediately as long as some input resources are ready.

While it is widely acknowledged that GraphLab(async) is better than or comparable to GraphLab(sync) in handling many classic tasks such as PageRank computation, we find that the opposite sometimes can be true for multi-processing. Our results with various machine and workload settings are shown in Table 4, where we compare GraphLab(sync) and GraphLab(async) on a classic task (i.e., PageRank) and a multi-processing task (i.e., BPPR) for the DBLP dataset. PageRank is a global metric of node importance, and its computation workload is similar to a Personalized PageRank query that takes a single source as input. Hence, PageRank computation is in general lighter than BPPR. For the PageRank computation, GraphLab(async) is better than GraphLab(sync), and the benefit gains with the number of machines. This is because the synchronization cost increases with the number of machines, but the cost is largely eliminated in GraphLab(async). However, such a benefit has not been observed for BPPR. For example, in a 16-machine cluster, GraphLab(async) is about 2.5 ($\approx 9.6/3.9$) times better than GraphLab(sync) in computing PageRank, whereas GraphLab(async) is 2.8 ($\approx 245/88$) times worse in computing BPPR with a workload of 512.

There could be several reasons for the contrasting behavior in computing PageRank and BPPR. First, in processing BPPR the synchronization overhead becomes relatively minor compared with the network cost. For example, processing BPPR incurs messages up to 6.4GB as shown in Table 4. This is partly because the network overhead incurred is significantly higher when the workload is heavier. We further observe that GraphLab(async) can incur more messages than GraphLab(sync) in a high-load situation. To explain, at each round, PageRank simply requires every vertex to distribute some portion of the PageRank value to its neighbors. BPPR, in contrast, requires each vertex to extend random walks whose number is proportional to the workload setting. As a result, the performance for BPPR is dominated by the workload-related cost. Hence, the benefit of removing synchronization barrier diminishes in a high-load situation, possibly leading to inferior performance of the asynchronous VC-system. Furthermore, GraphLab(sync) has the advantage of combining and reducing the sizes of multiple messages. When random walks with the same source need to move to the same neighbor, they are combined into one message by editing the number of walks within the message. This explains the relatively smaller message size and lower cost of GraphLab(sync) in a high-load situation.



(a) BPPR 8 Machines (total workload = 12800)



(b) BPPR 27 Machines (total workload = 40960)

Figure 9: Unequal batches are beneficial. (Evaluated on DBLP dataset)

Table 4: Asynchronous GraphLab v.s. Synchronous VC-Systmes (seconds/bytes-per-machine).

Machine	PageRank		Batch Personalized PageRank GraphLab (Workload)							
	GraphLab(sync)	GraphLab(async)	sync(8)	async(8)	sync(32)	async(32)	sync (128)	async(128)	sync (512)	async(512)
1	12.9s/33M	**9.1s**/33M	**33s**/41M	35s/41M	**86s**/41M	91s/41M	**277s**/41M	283s/41M	**974s**/0.7G	994s/41M
2	11.6s/151M	**7.7s**/32M	**33s**/0.4G	**33s**/0.7G	**62s**/0.8G	70s/1.7G	**148s**/1.7G	179s/2.4G	**457s**/3.8G	581s/5.3G
4	10.5s/125M	**5.6s**/28M	32s/0.4G	**24s**/0.6G	**49s**/0.7G	46s/1.4G	**91s**/1.3G	114s/1.9G	**242s**/2.8G	339s/4.3G
8	10.3s/97M	**4.0s**/22M	27s/0.3G	**19s**/0.5G	40s/0.5G	**33s**/1.0G	**64s**/0.8G	77s/1.4G	**140s**/1.7G	241s/3.4G
16	9.6s/62M	**3.9s**/22M	28s/0.2G	**25s**/0.7G	34s/0.3G	**33s**/1.3G	**45s**/0.5G	54s/1.1G	**88s**/1.0G	245s/6.4G

Another reason is about locking. GraphLab(async) maintains a large thread pool, where each thread will apply the Gather-Apply-Scatter (GAS) computing model to an assigned vertex. In GraphLab(sync), they are called fibers (by default 1,000 fibers per machine), and a distributed lock is applied to ensure that no two neighbor vertexes are triggered at the same time. Compared to PageRank, the active period for each vertex lasts longer for BPPR, especially when the workload is high. The distributed locking mechanism results in higher overhead when the number of machines increases, which further explains that for BPPR the benefit of GraphLab(sync) is more significant compared with GraphLab(async) when the number of machines increases from 2 to 16 (i.e. the number of fibers increases from 2,000 to 16,000).

4.9 Further Discussions

While this paper uses VC-systems with default settings, it should not hinder a further exploration of other possible settings. We discuss several possible alternative settings.

Alternative Graph Partitioning. The default setting of graph partition in a VC-system is to partition the nodes/edges into different machines, leading to communication among machines. We can also set up a *whole graph access mode*, by deploying a VC-system respectively in each machine. As such, the whole graph can be accessed within each machine while the workload is partitioned equally across machines. For BPPR, each machine outputs the estimated BPPR results based on a subset of random walks. These values have to be aggregated to generate the final estimated BPPR results that correspond to the whole set of random walks. The whole graph access mode largely avoids communication among machines. The downside is that the graph occupies more memory in each machine than the default setting, making the machine easier to use up its main memory. Figure 10 shows the comparison of such setting using the same experimental environment as Figure 5 (c). Each bar is divided into two parts, where the upper part indicates the cost of the final aggregation. It shows that the whole graph access mode more easily overloads the machine if the workload is not properly divided. For example, with 1-batch and 2-batch settings, the performance of the system for a higher workload (e.g., 34560) is still unsatisfactory. The performance can be significantly improved when 4-batch scheme is applied. It even beats the performance with the default setting. Hence, a satisfactory performance can be achieved with a proper batch setting.

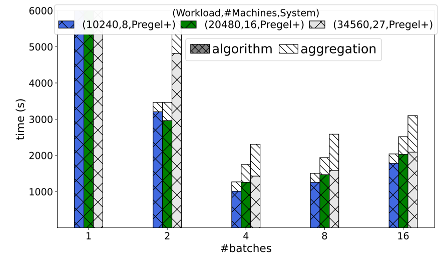


Figure 10: Graph is replicated to each machine, on which Pregel+ is built. Same setting as Figure 5 (c).

Alternative System Settings. The scale-out solution (using distributed computation) and scale-up solution (using one strong machine) are two possible ways to scale the computing. There are suitable application scenarios for both solutions [30]. For example, Salihoglu et al. [30] highlight that *complex graph computations* usually require the scale-out solution. In our study, multi-processing (e.g, BPPR) entails such high computation complexity due to the concurrency of many unit-tasks (e.g., PPR queries). In distributed graph processing, VC-Systems stand out and the ones we evaluated in the paper are the representatives. Meanwhile, there is increasing availability of strong machines which can provide shared memory and allow better cache optimizations, though a strong machine is usually expensive. Hence, it can be up to each user to determine whether to go for a scale-up or scale-out solution depending on their budgets and existing facilities. As future work, whether multi-processing can be efficiently handled with a scale-up solution is worth further exploration.

Alternative Workload Settings. It is also natural to set the unit task for BPPR as a PPR query and the workload as the number of queries. In other words, a batch contains a subset of source nodes for PPR queries. Due to space limitations, we put some relevant experimental results in our additional materials [25].

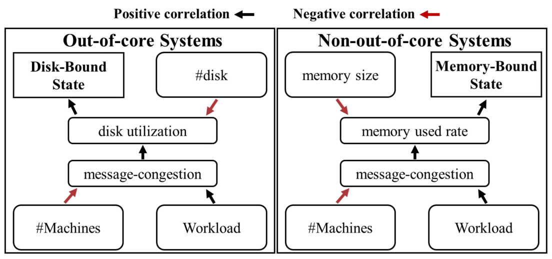


Figure 11: Correlations of different factors in a typical synchronous VC-system.

4.10 Summary

Traditional synchronous VC-systems could have larger room for optimizing multi-processing compared with asynchronous VC-systems. How to optimize the synchronous VC-systems is summarized in Figure 11. In general, the system state (i.e., memory-bound/disk-bound) is holistically affected by many factors. The memory-bound/disk-bound state is determined by run-time system parameters (e.g., disk utilization, memory cost) that depend on the message congestion in VC-systems, which are further influenced by the static system parameters (e.g., number of machines) and workloads. Other static system parameters, such as disk sizes or memory sizes, affect the system states such that more disk/memory resources keep away the disk-bound/memory-bound state. This holistic system view draws to the main design intuition as follows.

One Design Does Not Fit All. Figure 11 summarizes how a number of factors influence the optimal round-congestion tradeoff in a typical synchronous VC-system. We divide the cases into out-of-core systems (e.g., GraphD) and non-out-of-core systems (e.g., Pregel+). The former is illustrated in the left part of the figure: Following the black arrow, a workload increase brings in heavier message congestion, thereby leading to higher disk utilization and ultimately running the system into a disk-bound state. Similarly, the right part of the figure describes the case for non-out-of-core systems: An increase of the workload will eventually lead to a memory-bound state. As such, all the factors collectively determine the optimal batching strategy, resulting in an optimal round-congestion tradeoff. The optimal batch strategy depends on the given workload and the system settings such as the number of machines and the physical memory size in a machine, thus demonstrating one configuration does not fit all scenarios.

Practical Guidelines. Practitioners can use two steps to tune a typical VC-system. The first step is to gauge a suitable workload that will not overload the system. This can be monitored via a trial-and-error process using a binary search for the workload. In each trial, the overload situation can be detected by checking the memory consumption or disk utilization in the master machine. The second step is to ensure that later batches should have smaller workloads, as concluded from Figure 9. Another way of using our insights for tuning is to first collect statistics from light workloads, which are then used to automatically pick the suitable workload in each batch. We conduct a case study to demonstrate this in Section 5.

5 CASE STUDY: TUNING PREGEL+

We show one possible approach that applies our experimental insights in tuning VC-Systems for more efficient multi-processing. We select Pregel+ as the representative system throughout the section. It is important to note that we do not argue the method presented here is the only way to utilize our findings; instead, we aim to explore the potential of using our experimental insights for tuning systems. We give more case studies in our additional materials [25].

Given workload W (e.g., performing W α-decay random walks from each vertex in BPPR), we aim to learn an optimized batch execution strategy $\mathcal{S}^* = \{W_1, \ldots, W_t\}$ where $\sum_{1 \le i \le t} W_i = W$ and t is the number of batches which is also unknown. More specifically, processing with Strategy S means to process workloads batch-by-batch, where the i-th batch requires to process a workload of W_i *concurrently* and different batches are processed *sequentially*. We note that the intermediate results of the i-th batch have to be stored for final result aggregation. For example, the intermediate random walk results computed in earlier batches are important to compute the final PPR values.

Machine Overloading. A machine is overloaded if at least p percent of its physical memory is occupied, where p is the overloading parameter tuned by the users. A machine cluster is overloaded if at least one of its machines is overloaded. Based on our findings in Section 4, W_1 can be the maximum workload that does not overload any machine. For modeling purposes, we let $M(W_0, j)$ be the memory consumption of machine j if it performs a workload W_0. Also, we let $M^*(W_0)$ be the maximum memory cost incurred at any machine for workload W_0.

Residual Memory. As discussed in Section 4.5, the residual memory directly affects the runtime memory consumption; hence, it impacts the selection of $\mathcal{S}^*$ as well. In BPPR, particularly, we need to store the ending nodes of every random walk computed in each batch. We note that the number of random walks can be much larger than the number of nodes, because it is the product of the workload of the batch and the number of nodes. The storage of ending nodes incurs the residual memory cost, and we use a function $M_r^*(W_0)$ to denote the maximum residual memory caused by processing a workload of W_0. Therefore, in round j, the residual memory can be expressed by $M_r^*\left(\sum_{1 \le i \le j} W_i\right)$.

Objectives. As we discussed in Section 4, selecting a suitable S depends on the maximum run-time memory cost. Motivated by this, we aim to find $S = \{W_1, \ldots, W_t \mid \sum_{i=1}^{t} W_i = W\}$, such that

$$M_r^*\left(\sum_{1 \le i \le j} W_i\right) + M^*(W_{j+1}) \le pM, \forall 0 \le j \le t-1 \qquad (1)$$

As both M^* and M_r^* are positively correlated with the workload, we can model them as exponential functions.

$$M^*(W_0) = a_1 W_0^{b_1} + c_1, \; M_r^*(W_0) = a_2 W_0^{b_2} + c_2 \qquad (2)$$

Exponential functions are used because of their expressiveness. The degree of the correlation between the running time and workload can be controlled by the value b_1 (b_2). For example, $b_1 > 1$ indicates that $M(W_0)$ grows faster than that of the workload; the reverse is true if $b_1 < 1$. Combining Equations 1, 2 gives us the following objectives that for any $0 \le j \le t-1$, we have

$$p \cdot M \ge a_2 \cdot \left(\sum_{1 \le i \le j} W_i\right)^{b_2} + c_2 + a_1 \cdot (W_{j+1})^{b_1} + c_1 \qquad (3)$$

Training. To evaluate the parameter set $\mathcal{P} = \{a_1, a_2, b_1, b_2, c_1, c_2\}$, we introduce a training phase, which runs a few light-weight workloads on the cluster to collect statistics. Correspondingly, we

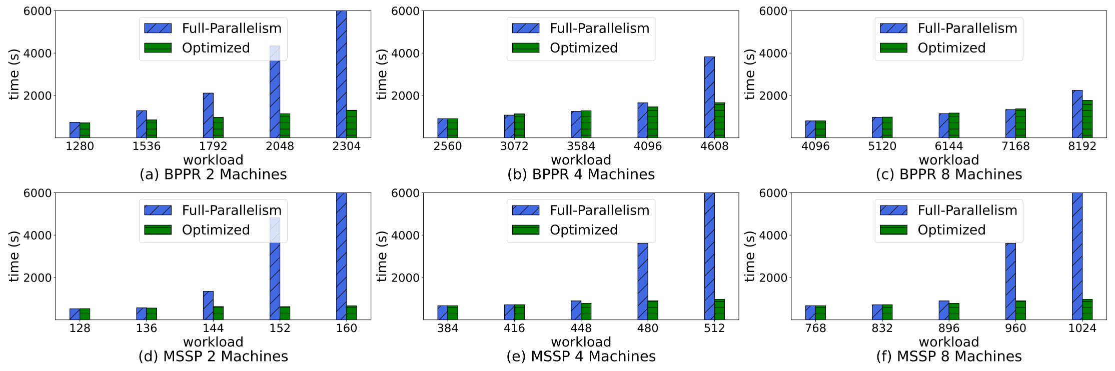


Figure 12: The impact of tuning Pregel+ with our findings (On DBLP dataset).

regard the procedure of running the real task as the evaluation phase. The training is affordable because it is done only once, and the training is configured to be significantly faster than the evaluation phase.

Particularly, we conduct training on the task with workload 2^r $(1 \leq r \leq h)$ where $W \gg 2^h$ (the condition ensures the training cost is minor). Through the training we collect h sets of runtime statistics, including the maximum memory $\{y_r | 1 \leq r \leq h\}$ and the maximum residual memory $\{y_r^* | 1 \leq r \leq h\}$. We estimate the exponential function parameters by the standard Levenberg-Marquardt algorithm (LMA) [26]. LMA aims to find the parameters (a_1, b_1, c_1) and (a_2, b_2, c_2) to minimize the sum of the deviations for a given set of $(2^r, y_r)$ and $(2^r, y_r^*)$ pairs:

$$arg \min_{a_1,b_1,c_1} \sum_{r=1}^{h} \left(y_r - f\left(2^r, a_1, b_1, c_1\right)\right)^2$$
$$arg \min_{a_2,b_2,c_2} \sum_{r=1}^{h} \left(y_r^* - f\left(2^r, a_2, b_2, c_2\right)\right)^2$$

where $f(x, a, b, c) = ax^b + c$. LMA uses linearization to approximate the gradient as follows:

$$\begin{aligned} &f\left(x_i, a + \delta_a, b + \delta_b, c + \delta_c\right) \approx f\left(x_i, a, b, c\right) \\ &+ \frac{df(x_i, a, b, c)}{da}\delta_a + \frac{df(x_i, a, b, c)}{db}\delta_b + \frac{df(x_i, a, b, c)}{dc}\delta_c \end{aligned} \quad (4)$$

We can thus rewrite the object function with the approximation to solve (a, b, c). In practice, (a, b, c) will be initialized randomly and updated in a gradient-decent manner until they converge or maximum trials are reached.

Computing W_t. We compute W_j $(1 \leq j \leq t)$ iteratively based on the evaluated $\mathcal{P}$. Setting $j = 0$ in Equation 1 gives $M^*(W_1) \leq p \cdot M$. As the system is expected to process as many tasks as possible before system overloading, we compute W_1 by solving $M^*(W_1) = p \cdot M$. In general, when we have $W_1, \ldots, W_i$ such that $\sum_{1 \leq j \leq i} W_j < W$, we can compute $M^*(W_{i+1})$ by Equation 5.

$$M^*(W_{i+1}) = pM - M_r^*\left(\sum_{1 \leq j \leq i} W_j\right) \quad (5)$$

Further, by Equation 2, we have

$$W_{i+1} = \left(\left(pM - a_2\left(\sum_{1 \leq j \leq i} W_j\right)^{b_2} + c_2 - c_1\right)/a_1\right)^{\frac{1}{b_1}} \quad (6)$$

Evaluation. We test BPPR and MSSP tasks for the DBLP dataset in Pregel+. We deployed the system on Galaxy-8. We test various workloads, and label the batch scheme computed by Equation 6 as *Optimized*. We mainly compare the *Optimized* scheme with Full-Parallelism. Figure 12 shows the results of performing tasks in DBLP in Pregel+, where (a), (b), (c) respectively report the BPPR results for 2, 4 and 8 machines, and (d), (e), (f) report the MSSP results for 2, 4 and 8 machines. The *Optimized* scheme is very stable with respect to the workload and the number of machines, whereas Full-Parallelism easily goes to very high cost when workload increases, both for BPPR and MSSP tasks. This means an auto-tuning framework is beneficial to system performance. We further investigate the optimum batch strategy output by the Equation 6, and found that the workload in each batch properly matches the desirable workload that avoids the overloading cases. Particularly, the workload of a later batch is typically smaller than an earlier batch because there is a higher residual memory cost at the beginning of later batches, and therefore less memory cost is allowed for performing the task within a batch. For example, performing a workload of 5120 for BPPR task with four machines leads to a workload division of [2747, 1388, 644, 266, 75], which shows the workload in a batch decreases in monotone. Such batching scheme precisely reflects the design of the *Optimized* scheme, leading to low running times with various workloads. In contrast, if we use the typical Full-Parallelism, the system will be overloaded by excessive memory used, leading to a significant latency.

6 CONCLUSION

This paper presents an experimental study on the VC-systems, with the purpose to evaluate the multi-processing tasks. We highlight the importance striking a suitable round-congestion tradeoff in VC-systems, which has not been well studied in this context before. We have some interesting findings that can particularly guide the practitioners in system optimizations. These results include a detailed analysis of what are the main factors that impact the VC-system performance from the perspective of doing an optimized round-congestion tradeoff. Furthermore, we also present a case study on designing a tuning framework for Pregel+ using our experimental insights. The tuning model showcases the usefulness of our experimental insights.

ACKNOWLEDGMENTS

This research was supported Singapore Ministry of Education grants RG18/21 and MOE2018-T2-2-091, as well as NTU startup grant. This publication was made possible by grant NPRP11C-1229-170007 from Qatar National Research Fund (a member of Qatar Foundation). The findings herein reflect the work, and are solely the responsibility, of the authors.

REFERENCES

[1] 2021. Stanford SNAP. *http://snap.stanford.edu/data/index.html* (2021).

[2] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. 1999. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing(SICOMP)* 28, 4 (1999), 1167–1181.

[3] Khaled Ammar and M Tamer Özsu. 2018. Experimental analysis of distributed graph systems. *Proceedings of the VLDB Endowment(PVLDB)* 11, 10 (2018), 1151–1164.

[4] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S Mirrokni, and Shang-Hua Teng. 2007. Local computation of PageRank contributions. In *International Workshop on Algorithms and Models for the Web-Graph(WAW)*. 150–165.

[5] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[6] Uday Bondhugula, Ananth Devulapalli, Joseph Fernando, Pete Wyckoff, and P Sadayappan. 2006. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *IEEE International Parallel and Distributed Processing Symposium(IPDPS)*. 10–pp.

[7] Sergio Cabello, Erin W Chambers, and Jeff Erickson. 2013. Multiple-source shortest paths in embedded graphs. *SIAM J. Comput.* 42, 4 (2013), 1542–1571.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment(PVLDB)* 8, 12 (2015), 1804–1815.

[10] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. 2014. On the power of the congested clique model. In *Principles of Distributed Computing(PODC)*. 367–376.

[11] Mohsen Ghaffari and Merav Parter. 2016. MST in log-star rounds of congested clique. In *Principles of Distributed Computing(PODC)*. 19–28.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM Symposium on Operating Systems Principles(SOSP)*. 29–43.

[13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. Wtf: The who to follow service at twitter. In *International World Wide Web Conference(WWW)*. 505–514.

[14] Apache Hadoop. 2011. Apache hadoop. *URL http://hadoop. apache. org* (2011).

[15] James W Hegeman and Sriram V Pemmaraju. 2015. Lessons from the congested clique applied to MapReduce. *Theoretical Computer Science(TCS)* 608 (2015), 268–281.

[16] Philip N Klein. 2005. Multiple-source shortest paths in planar graphs. In *Symposium on Discrete Algorithms(SODA)*. 146–155.

[17] Christian Konrad. 2018. MIS in the Congested Clique Model in O(loglogΔ)$Rounds$. *arXiv preprint arXiv:1802.07647*(2018).

[18] Christoph Lenzen. 2013. Optimal deterministic routing and sorting on the congested clique. In *Principles of Distributed Computing(PODC)*. 42–50.

[19] Wenqing Lin. 2019. Distributed Algorithms for Fully Personalized PageRank on Large Graphs. In *International World Wide Web Conference(WWW)*. 1084–1094.

[20] David C. Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C. Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. 2017. Related Pins at Pinterest: The Evolution of a Real-World Recommender System. In *International World Wide Web Conference(Companion) (WWW Companion)*. 583–592.

[21] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[22] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment(PVLDB)* 8, 3 (2014), 281–292.

[23] Siqiang Luo, Xiaokui Xiao, Wenqing Lin, and Ben Kao. 2019. BATON: Batch One-Hop Personalized PageRanks with Efficiency and Accuracy. *IEEE Transactions on Knowledge and Data Engineering(TKDE)* (2019).

[24] Siqiang Luo, Xiaokui Xiao, Wenqing Lin, and Ben Kao. 2019. Efficient batch one-hop personalized pageranks. In *IEEE International Conference on Data Engineering(ICDE)*. 1562–1565.

[25] Siqiang Luo, Zichen Zhu, Xiao Xiaokui, Yang Yin, Li Chunbo, and Kao Ben. 2022. Supplementary Material. *https://sites.google.com/view/add-material-multi-task-in-vc/* (2022).

[26] Kaj Madsen, Hans Bruun Nielsen, and Ole Tingleff. 2004. Methods for non-linear least squares problems. (2004).

[27] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Special Interest Group on Management of Data(SIGMOD)*. 135–146.

[28] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[29] Merav Parter and Hsin-Hao Su. 2018. Randomized (Δ+ 1)-Coloring in O ($\log^* \Delta$) Congested Clique Rounds. In *International Symposium on DIStributed Computing(DISC) 2018*.

[30] Semih Salihoglu and M Tamer Özsu. 2018. Response to "scale up or scale out for graph processing". *IEEE Internet Computing* 22, 5 (2018), 18–24.

[31] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Scientific and Statistical Database Management Conference(SSDBM)*. 22.

[32] Atish Das Sarma, Anisur Rahaman Molla, Gopal Pandurangan, and Eli Upfal. 2015. Fast distributed pagerank computation. *Theoretical Computer Science(TCS)* 561 (2015), 113–121.

[33] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. 2016. Hubppr: effective indexing for approximate personalized pagerank. *Proceedings of the VLDB Endowment(PVLDB)* 10, 3 (2016), 205–216.

[34] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *International World Wide Web Conference(WWW)*. 1307–1317.

[35] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment(PVLDB)* 7, 14 (2014), 1821–1832.

[36] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2017. Graphd: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Transactions on Parallel and Distributed Systems(IPDS)* 29, 1 (2017), 99–114.

[37] Dawei Yin, Liangjie Hong, Xiong Xiong, and Brian D Davison. 2011. Link formation analysis in microblogs. In *Special Interest Group on Information Retrieval(SIGIR)*. 1235–1236.

[38] Jun Zou and Faramarz Fekri. 2014. Exploiting Popularity and Similarity for Link Recommendation in Twitter Networks.. In *RSWeb@ RecSys*.