Spinner: Scalable Graph Partitioning in the Cloud

Claudio Martella[*], Dionysios Logothetis[†], Andreas Loukas[‡], Georgos Siganos[§]
[*]VU University Amsterdam, [†]Facebook, [‡]École Polytechnique Fédérale de Lausanne, [§]Qatar Computing Research Institute
Email: c.martella@vu.nl, dionysios@fb.com, andreas.loukas@epfl.ch, gsiganos@qf.org.qa

Abstract—**In this paper, we present a graph partitioning algorithm to partition graphs with trillions of edges. To achieve such scale, our solution leverages the vertex-centric Pregel abstraction provided by Giraph, a system for large-scale graph analytics. We designed our algorithm to compute partitions with high locality and fair balance, and focused on the characteristics necessary to reach wide adoption by practitioners in production. Our solution can (i) scale to massive graphs and thousands of compute cores, (ii) efficiently adapt partitions to changes to graphs and compute environments, and (iii) seamlessly integrate in existing systems without additional infrastructure.**

We evaluate our solution on the Facebook and Instagram graphs, as well as on other large-scale, real-world graphs. We show that it is scalable and computes partitionings with quality comparable, and sometimes outperforming, existing solutions. By integrating the computed partitionings in Giraph, we speedup various real-world applications by up to a factor of 5.6 compared to default hash-partitioning.

I. Introduction

Frameworks like Hadoop and Spark provide rich ecosystems for managing the entire life-cycle of data and are widely adopted at all scales. Practitioners typically deploy them across shared-nothing clusters in private or public clouds. In recent years, these ecosystems have been extended with solutions designed specifically for graph-related workloads responding to an increasing demand for tools to manage connected data [21], [14], [25], [28]. Due to the size of graphs available today, whether a social network, the Web, or a protein interaction network [16], graph management solutions, such as graph stores and graph processing engines, must distribute their workloads across such clusters.

Within such distributed systems, efficient operation relies on effective graph partitioning. Typically, the graph must be cut across k partitions, known as *k-way partitioning*, such that the number of edges spanning partitions is minimized, while keeping the number of edges in the partitions *balanced*. By assigning each of the k partitions to one of k machines, network communication between machines is reduced and load is distributed evenly, resulting in lower latency and increased throughput. For example, graph partitioning has been used to speedup both low-latency friends-of-friends lookup queries [31] and data-intensive graph analytics computations [29], [27].

A graph partitioning system is hence a key component of graph management systems. However, in order to be practical and simple to use in production, it cannot focus uniquely on computing partitionings that optimize locality and balance, but it also needs to have the following characteristics.

Scalable. Graphs may be massive, reaching billions of vertices and trillions of edges. At this scale, it is hard to provide good partitionings with locality and balance at the same time. A graph partitioning system must be able to provide an effective partitioning by exploiting parallelism, potentially even at the cost of some partitioning quality.

Adaptive. Graphs are *dynamic* with vertices and edges being constantly added and removed, and *elastic* compute systems can grow and shrink according to needs. In the face of these changes, a partitioning must be adapted without wasting resources, e.g. by repartitioning from scratch. Moreover, relevant particularly to graph stores, it must be *stable*, avoiding the need to reshuffle the graph across the machines upon adaptation.

Integrated. The system should exploit as much existing storage and compute infrastructure in the practitioner's compute environment without the need for new, ad-hoc infrastructure or data conversion and filtering. Every new component in a distributed environment introduces extensive development and engineering, and requires additional operational resources.

For example, Facebook uses Giraph for a number of production applications as it matches their requirements for a system that (i) scales graph analytics to their massive social graphs reaching one trillion edges, (ii) re-utilizes the Hadoop compute infrastructure, and (iii) leverages the Hive-based data warehouse for loading data and offloading results [12]. In other words, their approach favors systems that integrate well with their pipelines and re-use existing resources, avoiding new and ad-hoc infrastructure. Introducing a graph partitioning system within such framework and workflow requires the system to match the same requirements. An analogous choice was made by Google when designing a graph partitioning algorithm on MapReduce [9].

We have found that existing solutions do not present all the characteristics above. They either require to re-partitioning the graph from scratch upon changes, assume the partitioning to be performed on a single machine, introduce extensive redundant infrastructure, or depend on centralized components that hinder scalability. The fact that practitioners still use the ineffective but practical hash-partitioning scheme regardless of these solutions shows the importance of these characteristics towards wide adoption.

A. Our approach

Our work is motivated by the popularity of systems, like Pregel [21], that are designed for scalable graph mining. For instance, Giraph [1] and GraphX [15] are two systems inspired by Pregel, and are first citizens of the Hadoop and Spark ecosystems respectively. This has resulted in porting several graph algorithms on such systems [26], [24], [4]. However, no work has explored so far how the problem of graph partitioning can benefit from these architectures.

By implementing our solution as a Giraph application, we inherit (i) a scalable compute infrastructure for our partitioning system, and (ii) the ability to integrate the partitioning system seamlessly in Hadoop infrastructures. Giraph executes transparently as a Hadoop job and reads from and writes to most data stores in the ecosystem. This allows practitioners to both compute and use the partitionings with a range of data stores and compute engines.

We base our solution on the Label Propagation Algorithm (LPA) as it lends itself to a scalable implementation in the vertex-centric model. Moreover, the incremental nature of LPA allows us to tackle the challenges related to the adaptivity of the system.

B. *Contributions and organization*

In this paper, we make the following contributions.

- We introduce Spinner, a scalable graph partitioning algorithm based on LPA that computes k-way balanced partitions with good locality. Spinner avoids expensive centralizations of the partitioning algorithm that may offer strict guarantees about partitioning at the cost of scalability. To the best of our knowledge, this is the first implementation of a graph partitioning algorithm on the vertex-centric Pregel model.
- We extend Spinner to efficiently adapt a partitioning upon changes to the graph or the compute environment. By avoiding re-computations from scratch, Spinner reduces the time to update the partitioning by about 80% even for large changes (30%) to the graph, allowing for frequent adaptation, and saving computation resources. Further, the incremental adaptation prevents graph management systems from shuffling the graph upon changes.
- We evaluate Spinner extensively, using synthetic and real graphs. We show that Spinner scales near-linearly to billion-vertex graphs and adapts efficiently to dynamic changes. Further, we show that Spinner significantly improves real-world applications performance when integrated into Giraph with a speedup up to a factor of 5.6 compared to standard hash-partitioning.
- We provide an open source implementation of the algorithm on top of the Giraph graph processing system. It can effectively run as a Hadoop job leveraging any existing Hadoop infrastructure.

The remaining of this paper is organized as follows. In Section II, we describe the Spinner algorithm. Section III describes the implementation of Spinner in the Pregel model, while in Section IV we present our evaluation. In Section V, we discuss related work, and Section VI concludes our study.

II. Spinner

We have designed *Spinner* based on LPA, a technique that has been used traditionally for community detection [11]. We choose LPA as it offers a generic and well understood framework on top of which we can build our partitioning algorithm as an optimization problem tailored to our objectives. In the following, we describe how we extend the formulation of LPA to achieve our goals.

Before going into the details of the algorithm, let us introduce the necessary notation. We define a graph as $G = \langle V, E \rangle$, where V is the set of vertices in the graph and E is the set of edges such that an edge $e \in E$ is a pair (u, v) with $u, v \in V$. We denote by $N(v) = \{u\colon u \in V, (u, v) \in E\}$ the neighborhood of a vertex v, and by $deg(v) = |N(v)|$ the degree of v. In a k-way partitioning, we define L as a set of labels $L = \{l_1, \ldots, l_k\}$ that essentially correspond to the k partitions. α is the labeling function $\alpha\colon V \to L$ such that $\alpha(v) = l_j$ if label l_j is assigned to vertex v.

The end goal of Spinner is to assign partitions, or labels, to each vertex such that it maximizes edge locality and partitions are balanced.

A. *K-way Label Propagation*

We first describe how to use basic LPA to maximize edge locality and then extend the algorithm to achieve balanced partitions. Initially, each vertex in the graph is assigned a label l_i at random, with $0 < i \leq k$. Subsequently, every vertex iteratively propagates its label to its neighbors. During this iterative process, a vertex acquires the label that is more frequent among its neighbors. Specifically, every vertex v assigns a different *score* for a particular label l which is equal to the number of neighbors assigned to label l

$$score(v, l) = \sum_{u \in N(v)} \delta(\alpha(u), l) \tag{1}$$

where δ is the Kronecker delta. Vertices show preference to labels with high score. More formally, a vertex updates its label to the label l_v that maximizes its score according to the update function

$$l_v = \arg\max_{l} score(v, l) \tag{2}$$

We call such an update a *migration* as it represents a logical vertex migration between two partitions.

In the event that multiple labels satisfy the update function, we break ties randomly, but prefer to keep the current label if it is among them. This break-tie rule improves convergence speed [11], and in our distributed implementation reduces unnecessary network communication (see Section III). The algorithm halts when no vertex updates its label.

Note that the original formulation of LPA assumes undirected graphs. However, very often graphs are directed (e.g. the Web). To use LPA as is, we would need to convert a graph to undirected. The naive approach would be to create an undirected edge between vertices u and v whenever at least one directed edge exists between vertex u and v in the directed graph.

This approach, though, is agnostic to the communication patterns of the applications running on top. In fact, if we placed two vertices that are connected in the data graph by two edges (with inverted direction) in two different partitions we would produce two edge cuts, and hence potentially two worker-worker messages in the a Pregel computation. This would have higher impact than separating two vertices connected by only one edge. Thus, we want to maintain this information when we convert the graph to undirected.

Spinner considers the number of directed edges connecting u, v in the original directed graph D by introducing a weighting function $w(u,v)$ such that

$$w(u,w) = \begin{cases} 1, & \text{if } (u,v) \in D \oplus (v,u) \in D \\ 2, & \text{if } (u,v) \in D \wedge (v,u) \in D \end{cases} \tag{3}$$

where $\oplus$ is the logical XOR. We extend now the formulation in (1) to include the weighting function

$$score'(v,l) = \sum_{u \in N(v)} w(u,v)\delta(\alpha(u),l) \tag{4}$$

In practice, the new update function effectively counts the number of messages exchanged locally in the system.

Notice that, so far, the formulation of LPA does not dictate in what order and when vertices propagate their labels or how we should parallelize this process. For instance, the propagation can occur in an asynchronous manner, with a vertex propagating its label to its neighbors immediately after an update. A synchronous propagation, instead, occurs in distinct iterations with every vertex propagating its label at the end of an iteration. In Section III, we will see how we constraint the propagation to occur in a synchronous fashion to retro-fit LPA to the Pregel model.

B. Balanced Label Propagation

Next, we extend LPA to produce balanced partitions. In Spinner, we take a different path from previous work [31] that balances partitions by enforcing strict constraints on the partition sizes. Such an approach requires the addition to LPA of a centralized component to ensure the satisfaction of the balance constraints across the graph. Essentially, it calculates which of the possible migration decisions will not violate the constraints. This component is used after each LPA iteration, potentially increasing the algorithm overhead and limiting scalability.

Instead, as our aim is to provide a practical and scalable solution, Spinner relaxes this constraint, only *encouraging* a similar number of edges across the different partitions. In particular, to maintain balance, we integrate a *penalty function* into the vertex score in (4) that penalizes migrations to partitions that are nearly full. Importantly, we define the penalty function so that it is scalable to compute.

In the following, we consider the case of a *homogeneous* system, where each machine has equal resources. This setup is often preferred, for instance, in graph processing systems like Pregel, to eliminate the problem of stragglers and improve processing latency and overall resource utilization.

We define the *capacity* C of a partition as the maximum number of edges it can have so that partitions are balanced, which we set to

$$C = c \cdot \frac{|E|}{k} \tag{5}$$

where $c > 1$ is a constant, and the *load* $b(l)$ of a partition l as the actual number of edges in the partition

$$b(l) = \sum_{v \in G} deg(v)\delta(\alpha(v),l) \tag{6}$$

The capacity C represents the constraint that Spinner puts on the load of the partitions during an LPA iteration, and for homogeneous systems it is the same for every partition. Notice that in an ideally balanced partitioning, every partition would contain $|E|/k$ edges. However, Spinner uses parameter c to let the load of a partition exceed this ideal value. This allows for vertices to migrate among partitions, potentially improving locality, even if this is going to reduce balance.

At the same time, to control the degree of unbalance, we introduce the following penalty function that discourages the assignment of vertices to nearly full partitions. Given a partition l, we define the penalty function $\pi(l)$ as

$$\pi(l) = \frac{b(l)}{C} \tag{7}$$

The closer the current load of a partition to its capacity is, the higher the penalty of a migration to this partition is, with the penalty value ranging from 0 to 1. Next, we integrate the penalty function into the score function. To do so, we first normalize (4), and reformulate the score function as follows

$$score''(v,l) = \sum_{u \in N(v)} \frac{w(u,v)\delta(\alpha(u),l)}{\sum_{u \in N(v)} w(u,v)} - \pi(l) \tag{8}$$

This penalty function has the following desirable properties. First, using parameter c we can control the tradeoff between partition unbalance and convergence speed. A larger value of c increases the number of migrations allowed to each partition at each iteration. This possibly speeds up convergence, but may increase unbalance as more edges are allowed to be assigned to each partition over the ideal value $|E|/k$.

Second, it allows us to compute the score function in a scalable way. Notice that the locality score depends on per-vertex information. Further, computing the introduced penalty function only requires to calculate $b(l)$. This is an aggregate across all vertices that are assigned label l. As we describe in Section III, we can leverage the Giraph aggregation primitives to compute $b(l)$ for all possible labels in a scalable manner. As we show later, the introduction of this simple penalty function is enough to produce partitions with balance comparable to the state-of-the-art.

C. Convergence and halting

Although proving the convergence properties of LPA is a hard problem in the general case [11], our additional emphasis on partition balance, namely that a vertex can migrate to improve balance despite a decrease in locality, allow us to provide some analytical guarantees about Spinner's convergence and partitioning quality. One of the difficulties in proving convergence is that approaches based on LPA sometimes reach a limit cycle where the partitioning fluctuates between the same states. Spinner converges in bounded time without guarantees about the partition quality, however due to lack of space we present the proof in the extended version of this paper [2].

From a practical perspective, even if limit cycles are avoided often it is not worth spending compute cycles to achieve full convergence. Typically, most of the improvement in the partitioning quality occurs during the first iterations, and the improvement per iteration drops quickly after that. We validate this with real graphs in Section IV-A. For practical purposes, in this section we also provide a heuristic for deciding when to halt the execution of the algorithm.

In LPA, convergence is detected by the absence of vertices changing label, referred to as the halting condition. A number of strategies have been proposed to guarantee the halting of LPA in synchronous systems, such as Pregel. These strategies are either based on heuristics for tie breaking and halting, or on the order in which vertices are updated [35]. However, the heuristics are tailored to LPA's score function, which maximizes only locality. Instead, our score function does not maximize only locality, but also partition balance, rendering these strategies unsuitable. Hence, in Spinner we use a heuristic that tracks how the quality of partitioning improves across the entire graph.

At a given iteration, we define the *score* of the partitioning for graph G as the sum of the current scores of each vertex

$$score(G) = \sum_{v \in G} score''(v, \alpha(l_v)) \tag{9}$$

As vertices try to optimize their individual scores by making local decisions, this aggregate score gradually improves as well. We consider a partitioning to be in a *steady state*, when the score of the graph is not improved more than a given threshold ε for more than w consecutive iterations. The algorithm halts when it reaches steady state. Through ε we can control the trade-off between the running time of the algorithm and the improvement in the partitioning as it executes more iterations. At the same time, with w it is possible to impose a stricter requirement on stability; with a larger w, we require more iterations with no significant improvement until we accept to halt.

Note that this condition, commonly used by iterative hill-climbing optimization algorithms, does not guarantee halting at the optimal solution. However, as we present in Section II-D, Spinner periodically restarts the partitioning algorithm to adapt to changes to the graph or the compute environment. This natural need to adapt gives Spinner the opportunity to jump out of local optima.

D. Incremental Label Propagation

As edges and vertices are added and removed over time, the computed partitioning becomes outdated, degrading the global score. Upon such changes, we want to update the partitioning to reflect the new topology without repartitioning from scratch. Ideally, since the graph changes affect local areas of the graph, we want to update the latest stable partitioning only in the affected portions of the graph.

Due to its local and iterative nature, LPA lends itself to incremental computation. Intuitively, the effect of the graph changes is to "push" the current steady state away from the local optimum it converged to, towards a state with lower global score. To handle this change, we restart the iterations, letting the algorithm search for a new local optimum. In the event we have new vertices in the graph, we initially assign them to the least loaded partition, to ensure we do not violate the balance constraint. Subsequently, vertices evaluate their new local score, possibly deciding to migrate to another partition. The algorithm continues as described previously.

Note that the number of iterations required to converge to a new steady state depends on the number of graph changes and the last state. Sometimes, no iteration may be necessary, as certain changes may not affect any vertex to the point that the score of a different label is higher than the current one. Other changes may cause more migrations due to the disruption of certain weak local equaling. In this sense, the algorithm behaves as a hill-climbing optimization algorithm. As we will show in Section IV-C, even upon a large number of changes, Spinner saves a large fraction of the time of a re-partitioning from scratch.

E. Elastic Label Propagation

During the life-cycle of a graph, a system may need to re-distribute the graph across the compute cluster. For instance, physical machines may reach their maximum capacity, and the system may need to scale up with the addition of more machines, requiring to re-distribute the graph. Alternatively, we may perform such re-distribution just to increase the degree of parallelization and improve performance. Conversely, if the graph shrinks or the number of available machines decreases, we need to remove a number of partitions and, again, redistribute the graph.

In these scenarios, we want the algorithm to adapt to the new number of partitions without repartitioning the graph from scratch. Spinner achieves this in the following way. Upon a change in the number of partition, Spinner lets each vertex decide independently whether it should migrate using a probabilistic approach. In the case we want to add n new partitions to the system, each vertex picks one of the new partitions randomly and migrates to it with a probability p such that

$$p = \frac{n}{k+n} \tag{10}$$

In the case we want to remove n partitions, all the vertices assigned to those partitions migrate to one of the remaining ones (chosen uniformly at random).

In both cases, after the vertices have migrated, we restart the algorithm to adapt the partitioning to the new assignments. As in the case of incremental LPA, the number of iterations required to converge to a new steady state depends on factors, such as the graph size, and the number of partitions added or removed.

By introducing these random migrations upon a change, this strategy can disrupt the current partitioning, degrading the global score. However, it has a number of interesting characteristics. First, it remains a decentralized and lightweight heuristic as each vertex makes a decision to migrate independently. Second, by choosing randomly, the partitions remain fairly balanced even after a change in the number of partitions. Third, this random change from the current state of the optimization problem may allow the solution to jump out of a local optimum.

Note that, if the number n of new partitions is large, the time to adapt the partitioning may be quite high due to a large number of random migrations. However, in practice, the frequency with which partitions are added or removed is low compared, for example, to the number of times a partitioning is updated due to changes in the graph itself. Furthermore, although vertices are shuffled around, the locality of those vertices that do not migrate is not completely destroyed, such as if the partitioning was performed from scratch. The adaptation of

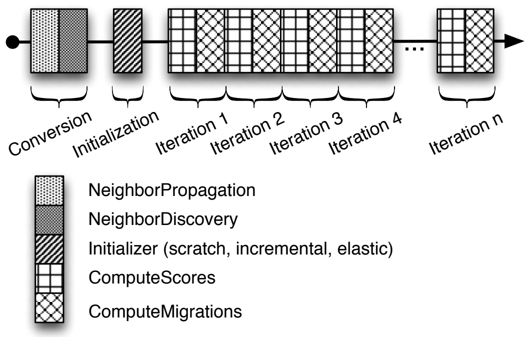


Fig. 1. Organization of the algorithm in multiple phases, each implemented by one or multiple steps (block). Each algorithm step is implemented as a Pregel superstep.

the partitioning to the new number of partitions will naturally take advantage of the late state of the partitioning.

III. PREGEL IMPLEMENTATION

We implemented Spinner in Apache Giraph [1] and open sourced the code[1]. Giraph is an open source project with a Java implementation of the Pregel model. Giraph is a batch graph processing system that runs on Hadoop [6], and can run computations on graphs with hundreds of billions of edges across clusters consisting of commodity machines.

In this section, we describe the implementation details of Spinner. We show how we extend the LPA formulation to leverage the synchronous vertex-centric programming model of a system like Giraph.

A. *Vertex-centric partitioning*

At a high-level, the algorithm is organized in three phases, depicted in Figure 1. In the first phase, since LPA assumes an undirected graph, if directed, Spinner converts it to a weighted undirected form as described in Section II-A. In the second phase, Spinner initializes the partition of each vertex depending on whether it partitions the graph from scratch or it is adapting an existing partitioning.

Subsequently, the third phase that implements the main LPA iterative migration process starts. In this phase, Spinner iteratively executes two different steps. In the first step, each vertex computes the label that maximizes its local score function. In the second step, vertices decide whether to migrate by changing label or defer migration. At the end of each iteration, Spinner evaluates the halting condition to decide whether to continue or stop computation.

We implement each of these phases as a series of Giraph supersteps. In the following subsections, we describe each phase in detail.

1) Graph conversion and initialization: We implement the first phase, the graph conversion, as two Giraph supersteps. Note that the Giraph data model is a distributed directed graph, where every vertex is aware of its outgoing edges but not of the incoming ones. For this reason, in the first superstep, each vertex sends its ID as a message to its neighbors. We call this step *NeighborPropagation*.

[1]http://grafos.ml

During the second superstep, a vertex receives a message from every other vertex that has an edge to it. For each received message, a vertex checks whether an outgoing edge towards the other endpoint is already present. If this is the case, the vertex sets the weight of the associated edge to 2. Otherwise, it creates an edge pointing to the other vertex with a weight of 1. We call this step *NeighborDiscovery*.

After this, Spinner executes the second phase, assigning partitions to each vertex. We call this step *Initialization*, and it corresponds to a single Giraph superstep. In the case Spinner partitions the graph from scratch, each vertex chooses a random partition. We will consider the case of adapting a partitioning in following sections. At this point, the LPA computation starts on the undirected graph.

2) Local computation of labels: The vertex-centric programming model of Pregel lends itself to the implementation of LPA. During an LPA iteration, each vertex computes the label that maximizes its local score based on the load of each partition and the labels of its neighbors. Each vertex stores the label of a neighbor in the value of the edge that connects them. When a vertex changes label, it informs its neighbors of the new label through a message. Upon reception of the message, neighboring vertices update the corresponding edge value with the new label.

We implement a single LPA iteration as two successive Giraph supersteps that we call *ComputeScores* and *ComputeMigrations*. In the first step, the *ComputeScores*, a vertex finds the label that maximizes its score. In more detail, during this step each vertex performs the following operations: (i) it iterates over the messages, if any, and updates the edge values with the new partitions of its neighbors, (ii) it iterates over its edges and computes the frequency of each label across its neighborhood, (iii) it computes the label that maximizes its local score, (iv) if a label with a higher score than the current one is found, the vertex is flagged as *candidate* to change label in the next step.

3) Decentralized migration decisions: Implementing our algorithm in a synchronous model introduces additional complexity. If we let every candidate vertex change label to maximize its local score during the *ComputeScores* step, we could obtain a partition that is unbalanced and violates the maximum capacities restriction. Intuitively, imagine that at a given time a partition was less loaded than the others. This partition could be potentially attractive to many vertices as the penalty function would favor that partition. As each vertex computes its score independently based on the partition loads computed *at the beginning* of the iteration, many vertices could choose independently that same partition label. To avoid this condition, we introduce an additional step that we call *ComputeMigrations*, and that serves the purpose of maintaining balance.

To avoid exceeding partition capacities, vertices need to coordinate after they have decided the label that maximizes their score during the *ComputeScores* step. To keep the solution decentralized, we opt for a probabilistic approach: a candidate vertex changes to a label with a probability that depends (i) on the remaining capacity of the corresponding partition and (ii) the total number of vertices that are candidates to change to the specific label.

More specifically, suppose that at iteration i partition l has a remaining capacity $r(l)$ such that

$$r(l) = C - b(l) \tag{11}$$

Suppose that $M(l)$ is the set of candidate vertices that want to change to label l, which is determined during the *ComputeScores* step of the iteration. We define the load of $M(l)$ as

$$m(l) = \sum_{v \in M(l)} deg(v) \tag{12}$$

This is the total load in edges that vertices in $M(l)$ would carry to partition l if they all migrated. Since $m(l)$ might be higher than the remaining capacity, in the second step of the iteration, we execute each candidate, and we only let it change with a probability p such that

$$p = \frac{r(l)}{m(l)} \tag{13}$$

Upon change, each vertex updates the capacities of the current partition and the new partition, and it updates the global score through the associated counter. It also sends a message to all its neighbors, with its new label. At this point, after all vertices have changed label, the halting condition can be evaluated based on the global score.

This strategy has the advantage of requiring neither centralized nor direct coordination between vertices. Vertices can independently decide to change label or retry in the next iteration. Moreover, it is lightweight, as probabilities can be computed on each worker at the beginning of the step. Because this is a probabilistic approach, it is possible that the load of a partition exceeds the remaining capacity. Nevertheless, the probability is bounded and decreases exponentially with the number of migrating vertices and super-exponentially with the inverse of the maximum degree.

Proposition 1. *The probability that at iteration $i+1$ the load $b_{i+1}(l)$ exceeds the capacity by a factor of $\varepsilon\, r_i(l)$ is*

$$\mathbf{Pr}(b_{i+1}(l) \geq C + \varepsilon\, r_i(l)) \leq e^{-2|M(l)|\Phi(\varepsilon)}, \tag{14}$$

where $\Phi(\varepsilon) = \left(\frac{\varepsilon\, r_i(l)}{\Delta - \delta}\right)^2$ and δ, Δ is the minimum and maximum degree of the vertices in $M(l)$, respectively.

Proof: Let X_v be a random variable which becomes 0 when vertex v does not migrate and $deg(v)$ otherwise. The expected value of X_v is

$$\mathbf{E}(X_v) = 0 \cdot (1-p) + deg(v)\, p = deg(v)\, p.$$

The total load carried by the vertices that migrate is described by the random variable $X = \sum_{v \in M(l)} X_v$ and has expected value

$$\mathbf{E}(X) = \mathbf{E}\left(\sum_{v \in M(l)} X_v\right) = \sum_{v \in M(l)} \mathbf{E}(X_v) = p \sum_{v \in M(l)} deg(v) = r(l).$$

We want to bound the probability that X is larger than $r(l)$, that is, the number of edges that migrate to l exceeds the remaining capacity of l. Using Hoeffding's method, we have that for any $t > 0$,

$$\mathbf{Pr}(X - \mathbf{E}(X) \geq t) \leq \exp\left(-\frac{2|M(l)|^2 t^2}{\sum\limits_{v \in M(l)} (\Delta - \delta)^2}\right) = \exp\left(-\frac{2|M(l)|t^2}{(\Delta - \delta)^2}\right),$$

where δ and Δ are the minimum and maximum degree of the vertices in $M(l)$, respectively. Setting $t = \varepsilon\, \mathbf{E}(X)$, we obtain the desired upper bound:

$$\mathbf{Pr}(X \geq (1+\varepsilon)\mathbf{E}(X)) = \mathbf{Pr}(X + b(l) \geq C + \varepsilon r(l))$$
$$\leq \exp\left(-2\,|M(l)|\left(\frac{\varepsilon\, r(l)}{\Delta - \delta}\right)^2\right)$$

■

We can conclude that with high probability at each iteration Spinner does not violate the partition capacity. To give an example, consider that $|M(l)| = 200$ vertices with minimum and maximum degree $\delta = 1$ and $\Delta = 500$, respectively, want to migrate to partition l. The probability that, after the migration, the load $b_{i+1}(l)$ exceeds $1.2\, r_i(l) + b_i(l) = C + 0.2\, r_i(l)$ is smaller than 0.2, where the probability that it exceeds $C + 0.4\, r_i(l)$ is smaller than 0.0016. Note that, being an upper bound, this is a pessimistic estimate. In Section IV-A1, we show experimentally that unbalance is much lower.

4) Asynchronous per-worker computation: Although the introduction of the *ComputeMigrations* step helps maintain balance by preventing excessive vertices from acquiring the same label, it depends on synchronous updates of the partition loads. The probabilistic migration decision described in III-A3 is based on the partition loads calculated during the previous superstep, and ignores any migrations decision performed during the current superstep. Consequently, a less loaded partition will be attractive to many vertices, but only a few of them will be allowed to migrate to it, delaying the migration decision of the remaining ones until the next supersteps.

In general, the order in which vertices update their label impacts convergence speed. While asynchronous graph processing systems allow more flexibility in scheduling of vertex updates, the synchronous nature of the Pregel model does not allow any control on the order of vertex computation.

However, Spinner leverages features of the Giraph API to emulate an asynchronous computation without the need of a purely asynchronous model. Specifically, Spinner treats each iteration within the same physical machine worker of a cluster as an asynchronous computation. During an iteration, each worker maintains local values for the partition loads that are shared across all vertices in the same worker. When a vertex is evaluated in the *ComputeScores* step and it becomes a candidate to change to a label, it updates the local values of the load of the corresponding partition asynchronously. As an effect, subsequent vertex computations in the same iteration and on the same worker use more up-to-date partition load information. Note that every vertex still has to be evaluated in the *ComputeMigrations* step for consistency among workers.

This approach overall speeds up convergence, and does not hurt the scalability properties of the Pregel model. In fact, the Spinner implementation leverages a feature supported by the Giraph programming interface that allows data sharing and computations on a per-worker basis. The information shared within the same worker is a set of counters for each partition and therefore does not add to the memory overhead. Furthermore, it still does not require any coordination across workers.

5) Management of partition loads and counters: Spinner relies on a number of counters to execute the partitioning: the global score, the partition loads $b(l)$, and the migration counters $m(l)$. The Pregel model supports global computation of commutative and associative operations through *aggregators*. During each superstep, vertices can aggregate values into named aggregators, and they can access the value aggregated during the previous superstep. In Pregel, each aggregator is computed in parallel by each worker for the aggregations performed by the assigned vertices, and a master worker aggregates these values at the end of the superstep. Giraph implements sharded aggregators, where the duty of the master worker for each aggregator is delegated to a different worker. This architecture allows for scalability through a fair distribution of load and parallel communication of partial aggregations. To exploit this feature, Spinner implements each counter through a different aggregator, making the management of counters scalable.

IV. Evaluation

In this section, we assess Spinner's ability to produce good partitions on large graphs. Specifically, we evaluate the partitioning quality in terms of locality and balance and use Spinner to partition billion-vertex graphs. Furthermore, we evaluate Spinner's ability to support frequent adaptation in dynamic environments. Finally, we utilize the partitioning computed by Spinner with the Giraph graph processing engine and measure the impact on the performance of real analytical applications. For our experiments, we use the Facebook and Instagram social graphs, and a variety of other large-scale real-world graphs for comparison and repeatability. Table II summarizes the real datasets we used.

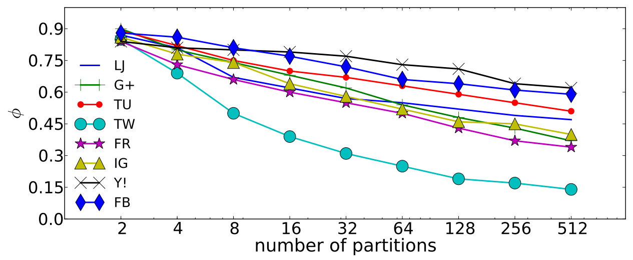
A. Partitioning quality

In our first set of experiments, we measure the quality of the partitions that Spinner can compute in terms of locality and balance. We measure locality with the *ratio of local edges* ϕ and balance with the *maximum normalized load* ρ, defined as

$$\phi = \frac{\#\ local\ edges}{|E|},\ \rho = \frac{maximum\ load}{\frac{|E|}{k}} \qquad (15)$$

where k is the number of partitions, # *local edges* represents the number of edges that connect two vertices assigned to the same partition, and *maximum load* represents the number of edges assigned to the most loaded partition. The maximum normalized load metric is typically used to measure unbalance and represents the percentage-wise difference of the most loaded partition from a perfectly balanced partition.

We chose these metrics for two reasons. First, they map naturally to the Pregel paradigm, as (i) the amount of data sent over the network is proportional to the number of edges spanning different partitions, in turn assigned to different machines, and (ii) the maximum load defines the amount work that the most loaded machines will have to manage (the load of a Pregel worker is often proportional to the number of edges assigned to it), which is particularly relevant in a synchronous system as we discuss in Section IV-F. Second, these are the metrics used by the techniques from the literature we compare to.



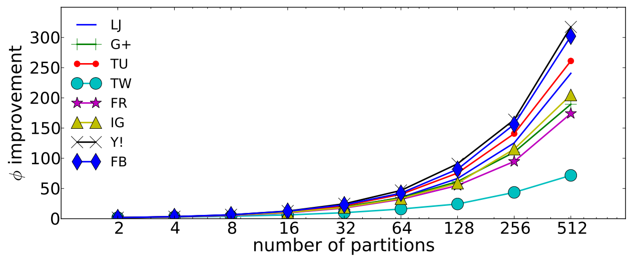


Fig. 2. (Top) Partitioning locality for varying number of partitions. X-axis is in log scale. (Bottom) Improvement in locality compared to hash partitioning. X-axis is in log scale.

First, we study how partitioning quality depends on the number of partitions. We vary the number of partitions and measure locality and balance for different graphs. For this and the remaining experiments, we set the algorithm parameters as follows: additional capacity $c = 1.05$, and halting thresholds $\varepsilon = 0.001$ and $w = 5$.

In Figure 2(a), we show that Spinner is able to produce partitions with high locality for all the graphs also for large numbers of partitions. With respect to balance, Spinner calculates fairly balanced partitions. Table III shows the average value of the maximum normalized load for each graph. For example, a ρ value of 1.059 for the Twitter graph means that no partition exceeds the ideal size by more than 5.9% edges.

To give perspective on the quality of the partitions that Spinner computes, Figure 2(b) shows the improvement in the percentage of local edges compared to hash partitioning. We perform this comparison for the same set of graphs. Notice that for 512 partitions Spinner increases locality up to over 300 times.

In Table I, we compare Spinner with state-of-the-art approaches. Recall that our primary goal for Spinner is to design a scalable algorithm for the Pregel model that is practical in maintaining the resulting partitioning, and that is *comparable* to the state-of-the-art in terms of locality and balance. Indeed, Spinner computes partitions with locality that is within 2-12% of the best approach, typically Metis, and balance that is within 1-3% of the best approach. In cases Spinner performs slightly worse than Fennel with respect to ϕ, it performs better with respect to ρ. These two metrics are connected as the most loaded partition will be the result of migrations to increase locality.

Approach	Twitter k=2 ϕ	Twitter k=2 ρ	Twitter k=4 ϕ	Twitter k=4 ρ	Twitter k=8 ϕ	Twitter k=8 ρ	Twitter k=16 ϕ	Twitter k=16 ρ	Twitter k=32 ϕ	Twitter k=32 ρ
Wang et al. [33]	0.61	1.30	0.36	1.63	0.23	2.19	0.15	2.63	0.11	1.87
Stanton et al. [29]	0.66	1.04	0.45	1.07	0.34	1.10	0.24	1.13	0.20	1.15
Fennel [30]	0.93	1.10	0.71	1.10	0.52	1.10	0.41	1.10	0.33	1.10
Metis [18]	0.88	1.02	0.76	1.03	0.64	1.03	0.46	1.03	0.37	1.03
Spinner	0.85	1.05	0.69	1.02	0.51	1.05	0.39	1.04	0.31	1.04

TABLE I. COMPARISON WITH STATE-OF-THE-ART APPROACHES. SPINNER OUTPERFORMS OR COMPARES TO THE STREAM-BASED APPROACHES, AND IS ONLY SLIGHTLY OUTPERFORMED BY SEQUENTIAL METIS. NOTICE THAT BECAUSE WANG ET AL. BALANCES ON THE NUMBER OF VERTICES, NOT EDGES, IT PRODUCES PARTITIONINGS WITH HIGH VALUES OF ρ.

Name	—V—	—E—	Directed	Source
Facebook (FB)	1.4B	400B	No	[3]
Instagram (IG)	-	-	Yes	[5]
Yahoo! (Y!)	1.4B	6.6B	Yes	[8]
Friendster (FR)	66M	1.8B	No	[35]
Twitter (TW)	40M	1.5B	Yes	[20]
Tuenti (TU)	12M	685M	No	[7]
Google+ (G+)	29M	462M	Yes	[13]
LiveJournal (LJ)	4.8M	69M	Yes	[10]

TABLE II. DATASETS USED FOR THE EVALUATION. GRAPHS ARE PRESENTED ORDERED BY NUMBER OF EDGES. WE OMIT THE INSTAGRAM GRAPH SIZE FOR CONFIDENTIALITY REASONS.

LJ	G+	TU	TW	FR	IG	Y!	FB
1.053	1.042	1.052	1.059	1.047	1.034	1.067	1.043

TABLE III. PARTITIONING BALANCE. THE TABLE SHOWS THE AVERAGE ρ FOR THE DIFFERENT GRAPHS.

In Figure 3 we present the evolution of the partitioning of the Yahoo! graph across 115 partitions. The initial partitioning presents high unbalance ($\rho = 1.41$), but after few iterations Spinner is able to reach fair balancing ($\rho = 1.05$). Note that the Twitter graph, known for the presence of high-degree hubs [14], produced even higher initial unbalance ($\rho = 1.67$). Looking at the shape of the $score(G)$ curve, notice that initially the global score is boosted precisely by the increased balance, while after balance is reached around iteration 10, it increases following the trend of ϕ.

In Table IV, we present the runtime of the first 10 iterations of Spinner on the three largest graphs. We fixed the number of iterations to simplify comparison, as Spinner requires a different number of iterations for each combination of graph and number of partitions. We found experimentally that the first 10 iterations tend to operate the bulk of the partitioning work, with the remaining iterations accounting for the tail of the computation. The results were obtained with an experimental Hadoop cluster of 200 machines.

Graph ($k = 64$)	Y!	IG	FB
First 10 iterations runtime (s.)	373	967	1562

TABLE IV. RUNTIME TO COMPUTE THE FIRST 10 ITERATIONS OF SPINNER. EVERY ITERATION INCLUDES TWO SUPERSTEPS.

1) Impact of additional capacity: Here, we investigate the effect of parameter c on balance and convergence speed. Recall that Spinner uses parameter c to control the maximum unbalance. Additionally, parameter c affects convergence speed; larger values of c should increase convergence speed as more migrations are allowed during each iteration.

In Section III-A3 we showed that with high probability Spinner respects partition capacities, that is, *maximum load* $\leq$

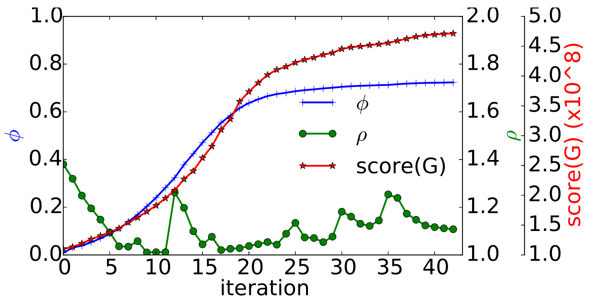
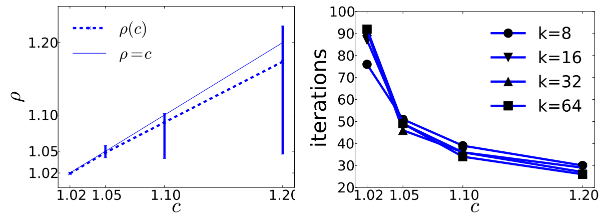


Fig. 3. Partitioning of the Yahoo! web graph across 115 partitions. The figure shows the evolution of metrics ϕ, ρ, and $score(G)$ across iterations.



(a) Balance (b) Convergence speed

Fig. 4. Impact of parameter c (a) on balance ρ, and (b) on convergence speed.

C. From the definitions of $\rho = \frac{maximum\ load}{\frac{|E|}{k}}$ and $C = c \cdot \frac{|E|}{k}$, we derive that with high probability $\rho \leq c$. Therefore, we can use parameter c to bound the unbalance of partitioning. For instance, if we allow partitions to store 20% more edges than the ideal value, Spinner should produce a partitioning with a maximum normalized load of up to 1.2.

To investigate these hypotheses experimentally, we vary the value of c and measure the number of iterations needed to converge as well as the final value of ρ. We partition the LiveJournal graph into 8, 16, 32, and 64 partitions, setting c to 1.02, 1.05, 1.10, and 1.20. We repeat each experiment 10 times and present the average and standard deviation for each value of c. As expected, Figure 4(a) shows that indeed on average $\rho \leq c$. Moreover, the error bars show the minimum and maximum value of ρ across the runs. We can notice that in some cases ρ is much smaller than c, and when it is exceeded, it is exceeded only by a small degree.

Figure 4(b) shows the relationship between c and the number of iterations needed to converge. Indeed, a larger value of c speeds up convergence. These results show how c can be used to control the maximum normalized load of the partitioning. It is up to the user to decide the trade-off between balance and speed of convergence.

B. Scalability

In these experiments we show that the algorithm affords a scalable implementation on modern large-scale graph processing frameworks such as Giraph. To this end, we apply our algorithm on synthetic graphs constructed with the Watts-Strogatz model [34]. In all these experiments, we set the parameters of the algorithm as described in Section IV-A. Using synthetic graphs here allows us to carefully control the number of vertices and edges, still working with a graph that resembles a real-world social network or web-graph characterized by "small-world" properties. On such a graph, the number of iterations required to partition the graph does not depend only on the number of vertices, edges and partitions, but also on its topology, and in particular on properties such as the clustering coefficient and the diameter.

For this reason, to validate the scalability of the algorithm we focus on the runtime of the *first* iteration, notably the iteration where all vertices receive notifications from all their neighbors, making it the most resource intensive iteration. In specific, we compute the runtime of an iteration as the sum of the time needed to compute the *ComputeScores* and the following *ComputeMigrations* supersteps. This approach allows us to factor out the runtime of algorithm as a function the number of vertices and edges.

Figure 5 presents the results of the experiments, executed on an AWS Hadoop cluster with 116 m2.4xlarge machines. In the first experiment, presented in Figure 5(a), we focus on the scalability of the algorithm as a function of the number of vertices and edges in the graph. For this, we fix the number of outgoing edges per vertex to 40. We connect the vertices following a ring lattice topology, and re-wire 30% of the edges randomly, setting the *beta* parameter of the Watts-Strogatz model to 0.3. We execute each experiment with 115 workers, for an exponentially increasing number of vertices, from 2 to 1024 million vertices (or one billion vertices) and we divide each graph in 64 partitions. The results, presented in a loglog plot, show a linear trend with respect to the size of the graph. Note that for the first data points the size of the graph is too small for such a large cluster, and we are actually measuring the overhead of Giraph.

In the second experiment, presented in Figure 5(b), we focus on the scalability of the algorithm as a function of the number of workers. Here, we fix the number of vertices to 1 billion, still constructed as described above, but we vary the number of workers linearly from 15 to 115 with steps of 15 workers (except for the last step where we add 10 workers). The drop from 111 to 15 seconds with 7.6 times more workers represents a speedup of 7.6.

In the third experiment, presented in Figure 5(b), we focus on the scalability of the algorithm as a function of the number of partitions. Again, we use 115 workers and we fix the number of vertices to 1 billion and construct the graph as described above. This time, we increase the number of partitions exponentially from 2 to 512. Also here, the loglog plot shows a near-linear trend, as the complexity of the heuristic executed by each vertex is proportional to the number of partitions k, and so is the cost of maintaining partition loads and counters through the sharded aggregators provided by Giraph.

C. Partitioning dynamic graphs

Due to the dynamic nature of graphs, the quality of an initial partitioning degrades over time. Re-partitioning from scratch can be a computationally expensive task if performed frequently and with potentially limited resources. In this section, we show that our algorithm minimizes the time to adapt the partitioning to the changes, making the maintenance of a well-partitioned graph a manageable task in terms of time and compute resources required. Specifically, we measure the savings in processing time and number of messages exchanged (i.e. load imposed on the network) relative to the approach of re-partitioning the graph from scratch. We track how these metrics vary as a function of the degree of change in the graph. Intuitively, larger graph changes require more time to adapt to an optimal partitioning.

For this experiment, we take a snapshot of the Tuenti [7] social graph that consists of approximately 10 million vertices and 530 million edges, and perform an initial partitioning. Subsequently, we add a varying number of edges that correspond to actual new friendships and measure the above metrics. Figure 6(a) shows that for changes up to 0.5%, our approach saves up to 86% of the processing time and, by reducing vertex migrations, up to 92% of the network traffic. Even for large graph changes, the algorithm still saves up to 80% of the processing time. Note that in every case our approach converges to a balanced partitioning, with a maximum normalized load of approximately 1.047, with 67%-69% local edges, similar to a re-partitioning from scratch.

D. Partitioning stability

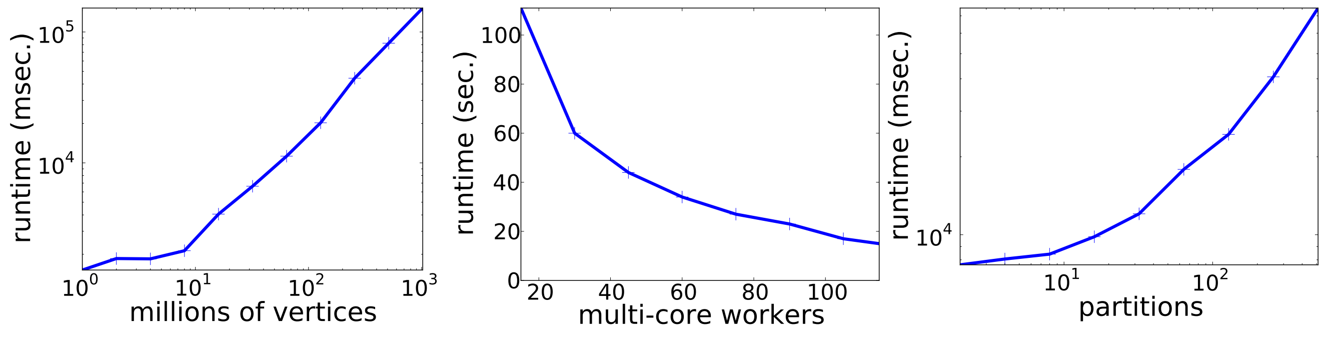
Adapting the partitioning helps maintain good locality as the graph changes, but may also require the graph management system (e.g. a graph DB) to move vertices and their associated state (e.g. user profiles in a social network) across partitions, potentially impacting performance. Aside from efficiency, the value of an adaptive algorithm lies also in maintaining *stable* partitions, that is, requiring only few vertices to move to new partitions upon graph changes. Here, we show that our approach achieves this goal.

We quantify the stability of the algorithm with a metric we call *partitioning difference*. The partitioning difference between two partitions is the percentage of vertices that belong to different partitions across two partitionings. This number represents the fraction of vertices that have to move to new partitions. Note that this metric is not the same as the total number of migrations that occur during the execution of the algorithm which only regards the performance of the algorithm per se.

In Figure 6(b), we measure the resulting partitioning difference when adapting and when re-partitioning from scratch as a function of the percentage of new edges. As expected, the percentage of vertices that have to move increases as we make more changes to the graph. However, our adaptive approach requires only 8%-11% of the vertices to move compared to a 95%-98% when re-partitioning, minimizing the impact.

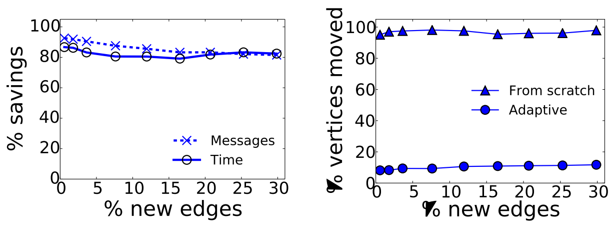
E. Adapting to resource changes

Here, we show that Spinner efficiently adapts the partitioning when resource changes force a change in the number



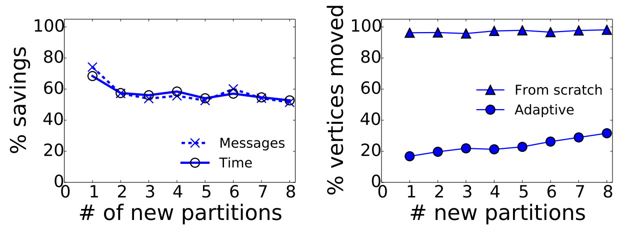
(a) Runtime vs. graph size (b) Runtime vs. cluster size (c) Runtime vs. k

Fig. 5. Scalability of Spinner. (a) Runtime as a function of the number of vertices, (b) runtime as a function of the number of workers, (c) runtime as a function of the number of partitions.



(a) Resource savings (b) Partitioning stability

Fig. 6. Adapting to graph changes. We vary the percentage of new edges in the graph and compare our adaptive approach to re-partitioning from scratch. We measure (a) savings in processing time and messages exchanged, and (b) the fraction of vertices that have to move upon re-partitioning.



(a) Resource savings (b) Partitioning stability

Fig. 7. Adapting to resource changes. We vary the number of new partitions and compare our adaptive approach to re-partitioning from scratch. We measure (a) savings in processing time and messages exchanged, and (b) the fraction of vertices that have to move upon re-partitioning.

of partitions. Initially, we partition the Tuenti graph snapshot described in Section IV-C into 32 partitions. Subsequently we add a varying number of partitions and either re-partition the graph from scratch or adapt the partitioning with Spinner. Figure 7(a) shows the savings in processing time and number of messages exchanged as a function of the number of new partitions. As expected, a larger number of new partitions requires more work to converge to a good partitioning. When increasing the capacity of the system by only 1 partition, Spinner adapts the partitions 74% faster relative to a re-partitioning.

Similarly to graph changes, a change in the capacity of the compute system may result in shuffling the graph. In Figure 7(b), we see that a change in the number of partitions can impact partitioning stability more than a large change in the input graph (Figure 6(b)). Still, when adding only 1 partition Spinner forces less than 17% of the vertices to shuffle compared to a 96% when re-partitioning from scratch.

Approach	Mean	Max.	Min.
Random	$5.8s \pm 2.3s$	$8.4s \pm 2.1s$	$3.4s \pm 1.9s$
Spinner	$4.7s \pm 1.5s$	$5.8s \pm 1.3s$	$3.1s \pm 1.1s$

TABLE V. IMPACT OF PARTITIONING BALANCE ON WORKER LOAD. WE SHOW THE TIME SPENT BY WORKERS TO FINISH A SUPERSTEP.

The high percentage when re-partitioning from scratch is expected due to the randomized nature of our algorithm. Note, though, that even a deterministic algorithm, like modulo hash partitioning, may suffer from the same problem when the number of partitions changes.

F. *Impact on application performance*

The partitioning computed by Spinner can be used by different graph management systems, to improve their performance. In this section, we use Spinner to optimize the execution of the Giraph graph processing system itself. After partitioning the input graph with Spinner, we instruct Giraph to use the computed partitioning and run real analytical applications on top. We then measure the impact on performance compared to using standard hash partitioning.

First, we assess the impact of partitioning balance on the actual load balance of the Giraph workers. In a synchronous processing engine like Giraph, an unbalanced partitioning results in the less loaded workers idling at the synchronization barrier. To validate this hypothesis, we partition the Twitter graph across 256 partitions and run 20 iterations of the PageRank algorithm on a cluster with 256 workers using (i) standard hash partitioning (random), and (ii) the partitioning computed by Spinner. For each run, we measure the time to compute a superstep by all the workers (Mean), the fastest (Min) and the slowest (Max), and compute the standard deviation across the 20 iterations. Table V shows the results.

The results indicate that with hash partitioning the workers are idling on average for 31% of the superstep, while with Spinner for only 19%. While the shorter time needed to compute a superstep can be imputed to the diminished number of cut edges, the decreased idling time is an effect of a more even load spread across the workers.

Second, we measure the impact of partitioning on processing time. We partitioned five graphs with Spinner and hash partitioning, and compared the time to run a variety of graph algorithms commonly used in analytical pipelines. Shortest Paths is used to study graph connectivity, PageRank is used at

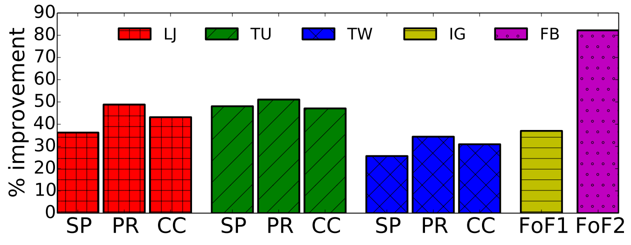


Fig. 8. Application runtime improvement over hash partitioning for Shortest Paths (SP), PageRank (PR), Connected Components (CC), and two Friend-of-Friend recommendation applications (FoF1, FoF2).

the core of ranking algorithms, and Connected Components is a general approach to finding communities. We also run two production Friend-of-Friend type of applications for follower and friend recommendation on Instagram and Facebook. We present these experimental results in Figure 8.

Using the partitionings computed by Spinner we significantly improve the performance across all graphs and applications. In the case of the Twitter graph, which is denser and harder to partition, the improvement ranges from 25% for SP to 35% for PR. In the case of LiveJournal and Tuenti, the running time decreases by up to 50%.

Friends-of-Friends workloads produce messages with total size in the order of $O(|E|^2)$, adding significant communication overhead. Due to the message size, they also require more memory resources, resulting in high garbage collection overhead. Using Spinner on Instagram, we decreased application runtime by 37%. We also noticed that the average memory used by worker machines during a superstep decreased by up to 35%, while the maximum memory used decreased by up to 47%. Effectively, this allows to scale to larger datasets with the same amount of resources, with no need for application-specific optimizations. On the Facebook graph, where communication overhead dominates computation due to the graph size, running time decreases by 82.2%. In practice, this allowed us to run an analytical task, which would otherwise take over a day, within 4-5 hours. Notice also that the running time of Spinner (Table IV) is small compared to the performance gain, amortizing its overhead almost immediately.

V. Related Work

In this section, we discuss the related work on large-scale k-way balanced graph partitioning.

(Par)Metis. Metis [18] is an offline partitioning algorithm, considered the golden standard against new approaches. However it has been shown [22] to scale poorly beyond few tens of million vertices due to the high computational and space complexity of its coarsening phase, including the parallel version ParMetis [19]. Moreover, it requires to transfer data out of the system, convert it to an appropriate format for partitioning, and then convert and transfer data back. This complicates processing pipelines, requires a separate compute cluster, and adds significant overheads. To summarize, it is not suitable for either the massive scale nor the computing infrastructures targeted by Spinner.

(Re-)streaming techniques. Spinner shares with streaming techniques [29], [30] the local greedy approach of positioning a vertex in the partition that maximizes locality and minimizes unbalance. While simple and effective, a scalable distributed implementation of one-shot streaming techniques is not obvious [23]. To achieve parallelism, restreaming techniques [23] partition the data stream across a number of workers that perform multiple passes of streaming partitioning on their portion of the stream. Because the operations of each worker are isolated, at the end of each pass workers exchange the new vertex assignments, which are used for updates during the following (re-)streaming pass. This sequence of passes is necessary to keep the local views of each worker consistent towards convergence. Note that restreaming passes are globally synchronized across workers. Other approaches, like Leopard [17], build on streaming approaches and augment them with vertex replication as a way to reduce the effective edge cut. However, such approaches are mainly targeted for read-only workloads. The replication overhead renders them unsuitable for graph processing systems, such as Pregel, or update-intensive graph database workloads.

We argue that restreaming techniques implement LPA, as vertex assignments are updated iteratively based on the (updated) assignments of their neighbors. Moreover, their parallel implementation is in practice a BSP computation, where workers compute their local state, exchange updates in a communication phase, and synchronize at a barrier. Hence, deploying such techniques requires implementing both an LPA-like algorithm and the underlying BSP machine. In other words, it requires implementing both Spinner and its compute environment. Instead, we implemented our approach as a Giraph application with less than 700 lines of code and without any additional parallel and distributed infrastructure. Moreover, the convergence speed of Spinner does not depend on the number of workers computing the partitioning. Restreaming techniques, on the other hand, require a larger number of iterations when more workers are involved. It is unclear how parallel restreaming techniques impact runtime performance, as no runtime statistics were reported. Finally, both streaming and re-streaming techniques require the graph to be stored and visited in a Breadth-First Search fashion to perform at their best, hence also not conducive to a general-purpose graph infrastructure.

LPA with centralized component. Two approaches are particularly relevant to Spinner [31], [33]. The former applies LPA to the MapReduce model, by attempting to improve locality through iterative vertex migrations across partitions. However, to guarantee balanced partitions, it executes a centralized linear solver between any two iterations. The complexity of the linear system is quadratic to the number of partitions and proportional to the size of the graph, making it impractical for large graphs. Moreover, MapReduce is known to be inefficient for iterative computations. The latter approach computes a k-way vertex-based balanced partitioning. It uses LPA to coarsen the input graph and then applies Metis to the coarsened graph. At the end, it projects the Metis partitioning back to the original graph. While the algorithm is scalable, we have found that for large number of partitions and skewed graphs the locality it produces is lower than Spinner. We also found it is very sensitive to its two parameters for which no intuition is available, differently from Spinner that requires only one parameter, for which we provide guidelines. Further, the approach is designed to run on the Trinity engine and is not suitable for implementation on a synchronous model such as Pregel. None of the two solutions investigates how to adapt a partitioning upon changes in the graph or the compute environment. On the other hand, xDGP [32] is a system designed for continuous Pregel-like computations that partitions the graph with LPA and migrates vertices *during* the computation. However, xDGP does not tackle balancing explicitly and often results in skewed partitions.

Finally, a similar approach to ours is proposed by Google in [9]. Their algorithm first embeds nodes of the graph onto a line, and then processes nodes in a distributed manner guided by the linear embedding order. Like Spinner, the approach leverages an off-the-shelf commodity platform, that is, MapReduce to run the algorithm at scale on existing infrastructure, minimizing the need for extra components. The system performs comparably and sometimes beats state-of-the-art techniques with respect to edge-cuts and balance. However, by requiring repartitioning from scratch it does not meet the adaptive property we set as a requirement in this work. Further, as they report no information on runtime (due to corporate restrictions), it is not clear how the choice of computing platform, i.e. MapReduce versus Pregel, impacts partitioning speed.

VI. Conclusions

We presented *Spinner*, a scalable and adaptive graph partitioning algorithm built on the Pregel abstraction. By sacrificing strict guarantees on balance, Spinner is practical for large-scale graph management systems. Through an extensive evaluation on a variety of graphs, we showed that Spinner computes partitions with locality and balance comparable to the state-of-the-art, but can do so at a scale of at least billion-vertex graphs. At the same time, its support for dynamic changes makes it more suitable for integrating into real graph systems. These properties makes Spinner possible to use as a generic replacement of the de-facto standard, hash partitioning, in cloud systems. Toward this, our scalable, open source implementation on Giraph makes Spinner easy to use on any commodity cluster.

Acknowledgments

The authors would like to thank SURFsara and, in particular, Jeroen Schot for supporting this work by providing one of the Hadoop clusters used in our experiments and helping with the required cluster setup, necessary for this kind of large-scale system research.

References

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

[2] Extended version of this work. http://arxiv.org/pdf/1404.3861v1.pdf.

[3] Facebook social network. http://www.facebook.com.

[4] GraphLab Open Source project. http://graphlab.org.

[5] Instagram social network. http://instagram.com.

[6] The Hadoop project. http://hadoop.apache.org.

[7] The Tuenti Social Network. http://www.tuenti.com.

[8] Yahoo Webscope Program. http://webscope.sandbox.yahoo.com.

[9] K. Aydin, et al. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 387–396. ACM, 2016.

[10] L. Backstrom, et al. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD*, Aug. 2006.

[11] M. Barber et al. Detecting network communities by propagating labels under constraints. *Physical Review E*, 80(2):026129, Aug. 2009.

[12] A. Ching, et al. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[13] N. Z. Gong, et al. Evolution of social-attribute networks: measurements, modeling, and implications using Google+. In *ACM Internet Measurement Conference*, Nov. 2012.

[14] J. E. Gonzalez, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*, Oct. 2012.

[15] J. E. Gonzalez, et al. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[16] H. He et al. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *ACM SIGMOD*, Vancouver, BC, Canada, June 2008.

[17] J. Huang et al. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. In *Very Large Data Bases*, volume 9, 2016.

[18] G. Karypis et al. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, University of Minnesota, Minneapolis, MN, 1995.

[19] G. Karypis et al. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *SIAM Review*, 41, 1999.

[20] H. Kwak, et al. What is Twitter, a social network or a news media? In *WWW*, 2010.

[21] G. Malewicz, et al. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[22] H. Meyerhenke, et al. Parallel graph partitioning for complex networks. *arXiv preprint arXiv:1404.4797*, 2014.

[23] J. Nishimura et al. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *ACM SIGKDD*, August 2013.

[24] B. Perozzi, et al. Scalable Graph Clustering with Pregel. In G. Ghoshal, et al., editors, *Workshop on Complex Networks*, volume 476 of *Studies in Computational Intelligence*, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] J. M. Pujol, et al. The little engine(s) that could: scaling online social networks. *ACM SIGCOMM Computer Communication Review*, Oct. 2011.

[26] L. Quick, et al. Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis. *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, 2012.

[27] M. Redekopp, et al. Optimizations and analysis of bsp graph processing models on public clouds. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 203–214. IEEE, 2013.

[28] B. Shao, et al. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.

[29] I. Stanton et al. Streaming Graph Partitioning for Large Distributed Graphs. In *ACM SIGKDD*, 2012.

[30] C. E. Tsourakakis, et al. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. *ACM International Conference on Web Search and Data Mining*, 2014.

[31] J. Ugander et al. Balanced label propagation for partitioning massive graphs. *ACM International Conference on Web Search and Data Mining*, 2013.

[32] L. M. Vaquero, et al. Adaptive partitioning for large-scale dynamic graphs. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 144–153. IEEE, 2014.

[33] L. Wang, et al. How to Partition a Billion-Node Graph. In *ICDE'14*, 2014.

[34] D. Watts et al. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[35] J. Yang et al. Defining and Evaluating Network Communities based on Ground-truth. In *IEEE International Conference on Data Mining*, May 2012.