(Re)partitioning for stream-enabled computation

Erwan Le Merrer, Yizhong Liang, Gilles Trédan

(Re)partitioning for stream-enabled computation

Erwan Le Merrer, Yizhong Liang
Technicolor
Rennes, France
surname.name@technicolor.com

Gilles Trédan
LAAS/CNRS
Toulouse, France
gtredan@laas.fr

***Abstract*—Partitioning an input graph over a set of workers is a complex operation. Objectives are twofold: split the work evenly, so that every worker gets an equal share, and minimize edge cut to achieve a good work locality (*i.e.* workers can work independently). Partitioning a graph accessible from memory is a notorious NP-complete problem. Motivated by the regain of interest for the stream processing paradigm (where nodes and edges arrive as a flow to the datacenter), we propose in this paper a stream-enabled graph partitioning system that constantly seeks an optimum between those two objectives. We first expose the hardness of partitioning using classic and static methods; we then exhibit the cut versus load balancing tradeoff, from an application point of view.**

With this tradeoff in mind, our approach translates the online partitioning problem into a standard optimization problem. A greedy algorithm handles the stream of incoming graph updates while optimizations are triggered on demand to improve upon the greedy decisions. Using simulations, we show that this approach is very efficient, turning a basic optimization strategy such as hill climbing into an online partitioning solution that compares favorably to literature's recent stream partitioning solutions.

***Keywords*-Graph-partitioning, Stream Processing, Load Balancing, Network Cuts.**

I. Introduction

Contemporary big-data applications ingest considerable amounts of data to produce meaningful and valuable information. As datasets keep on growing to unprecedented sizes, applications must rely on efficient and scalable computation means. Graph-based applications as social networks [1], search engines or recommender systems [2] have to deal with giant and constantly evolving networks of user or item interactions. As those terabytes of data cannot be efficiently processed and served by a single machine in the horizontal scalability model using commodity hardware, the solution is to *partition the interaction graph* onto multiples machines for parallel computation and request handling. As computation is fast with this scheme, the dataset evolution could be incorporated seamlessly by the application, so that fresh results are always available.

The *MapReduce framework* allows to process massive amount of information, in an offline manner [3]. Since very recently, there is a resurgence of interest about *stream processing*, with the proposal of open platforms such as Storm [4]. In this framework, data is treated as a flow, and each flow element is processed on the fly (and then possibly discarded). While more restrictive that MapReduce, this allows for online computation.

As the raison d'être of stream processing is to exhibit a low latency in its operation, relying on offline partitioning methods is not an option. As the graph structure continuously changes due to node/edge creations, calling a procedure that recomputes a partitioning from scratch at each change is overkill (see e.g. traditional approaches as [5]). In other words, incremental approaches to partitioning are mandatory. In this light, datacenter applications like Pregel [6] partition nodes onto machines based solely on their IDs; this is apparented to dispatch those nodes at random. A recent work proposes to load the graph stored on a disk as a stream of nodes, and to use cheap heuristics for node placement over one of the k processing machines on the fly [7]. Although the fact that this approach handles nodes as they are read, it assumes a full knowledge model, where the whole graph is accessible at a given time as the input.

In this paper, we propose a system that receives incoming edges, and places their endpoint nodes greedily in partitions, then performing online partitioning. Our system operates over a continuous flow of events arriving at a datacenter, then suiting the stream processing paradigm. Greedy placement is complemented with periodic partitioning reconfigurations at runtime, using solely little feedback from the application.

The contributions of this paper are:

- to exhibit *(i)* the instability of optimal partitions created by static partitioning algorithms, if they are run each time few new edges are added to the current graph, and *(ii)* the existence of a graph-related tradeoff between a well balanced graph (work is evenly divided among the parallel instances) and a low edge cut (workers should be able to process most of the request information local to their partition). Intuitively, this tradeoff pops up every time the system has to decide between favoring edge-cut at the price of well balancedness, or vice versa.
- based on these observations, to consider the problem of stream-enabled graph partitioning as a standard mathematical optimization problem. In this problem, the optimization parameters are the well balancedness and the edge-cut, and the metric to optimize is the

application performance, measured for instance by the average request processing time.
- to propose a stream-enabled graph partitioner built upon these observations. The realistic simulations we conducted show that a standard greedy optimization is efficient compared to current state-of-the-art stream-enabled partitioner [7], while executing in a more restricted model. Thanks to the greedy nature of the optimization, this performance is achieved for cheap.

The remaining of this paper is structured as follows: Section II exhibits the danger of seeking optimal partitionings in the context of streamed graphs, before presenting the model of execution considered. We then observe in Section III that traditional graph partitioning metrics are bound by a graph-dependent tradeoff that impacts application performance. Based on these observations, Section IV models the streamed-graph partitioning problem as an optimization problem, and proposes a greedy online partitioning mechanism, along with simulation results. A reconfiguration technique to be used periodically at runtime is then presented. Section V illustrates this partitioning scheme in an application context: graph-based recommendation. We finally present Related Work in Section VI before we conclude.

II. Model: processing over streamed graphs

A. Instability of Optimal Partitionings

This sections illustrates the difficulty of simply transposing traditional graph partitioning metrics in the context of graph streaming. The first major problem comes from the complexity of finding an optimal partitioning, which is NP-complete [8]. Considering the update rate of a streamed graph, such problem would have to be solved for every increment, which is not realistic.

Second problem stems from the unstability of such optimal partitioning. To illustrate this, consider the following example. Assume that k servers operate on a graph composed of a ring of $2k$ fully-connected clusters $C_1, \dots C_{2k}$, with $|C_i| = \frac{n}{2k}$. Assume for $i \in [1, k]$, we have A links between C_{2i-1} and C_{2i}, and B links between C_{2i} and $C_{2i+1[2k]}$, with $A, B < \frac{n}{2k}$. Figure 1 illustrates such topology in the case $k = 3$.

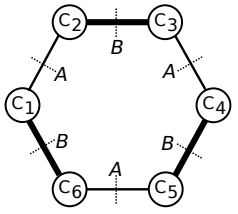


Figure 1. A ring graph for $k = 3$. Depending on the order of new edge arrivals, this topology triggers unstable decisions by partitioning methods.

Since any graph partition cutting through a cluster C_i would cut at least $\frac{n}{2k}$ links, balanced cuts are either along A, either along B. Assume $A = B - 1$: the optimal assignment is to map $(C_{2i} \cup C_{2i+1[2k]})$ to server i. Now assume that two new edges arrive on the A cut, for instance between C_1 and C_2. The new optimal assignment is to map $(C_{2i} \cup C_{2i-1[2k]})$ to server i. To reach this new optimal assignment, $\frac{n}{2}$ nodes need to be transferred during the reconfiguration. Observe that the arrival of two new edges along a B cut (or the removal of two edges along an A cut) can now generate the same amount of reconfiguration.

Let us imagine a worst case scenario where originally $A = 1$ and $B = 2$, and a stream of edges repeating the aforementioned scheme $(A, A, B, B, A, A, \dots)$ until $A = B = \frac{n}{2k}$. The described reconfiguration then happens $k\frac{n}{4k}$ times. This implies $\frac{n}{2}\frac{n}{4k} = \Omega(n^2)$ node transfers. Although this is a pathological worst-case scenario, bad situations cannot be discarded in dealing with real world graphs either. This illustrates the need for a specific approach to support graph updates, then adapted to the streamed graph model we now define.

B. Partitioning Model and Metrics for Streamed Graphs

We consider a streamed graph model, where edges arrive continuously to a central machine called the **partitioner**, $\mathcal{P}$. $\mathcal{P}$ is in charge of partitioning the streamed graph $G = (V, E)$ over a fixed set of k machines or cores, according to the computing hardware setup. Practically, it positions an incoming edge endpoints (*i.e.* nodes) on one or two of the machines that are hosting a partition of G, under the form of adjacency lists. G_∞ can be seen as the graph resulting from the aggregation of all arrivals, at the end of times. $P(i)$ denotes partition number i, and $|P(i)|$ the number of nodes currently in $P(i)$. Each partition has the capacity to host C nodes. $\bigcup_{i=1}^{k} P(i)$ then contains all nodes and edges from G seen so far.

We do not make any assumption on the order of arrival of elements in set E. The system operates on edges, implying that if one endpoint (*i.e.* one node) is unknown, its creation is handled by the system. $\Gamma(v)$ denotes the set of neighbors of node v.

$\mathcal{P}$ maintains a table mapping all nodes seen so far (*i.e.* edge endpoints) to their current partition assignment (an integer $[1, k]$), for being able to take greedy decisions on placement of incoming edges. State maintained at $\mathcal{P}$ is thus $O(|V|)$.

In this paper, we are interested in optimizing the average request processing time of the application on top of which our partitioning system is deployed. However, throughout the paper, we refer to two traditional metrics of graph partitioning:

- *Load balancing* is computed as the spread between the less and the more loaded of the k partitions. It can be

written as

$$\frac{\min_{i\in[k]}(|P(i)|)}{\max_{i\in[k]}(|P(i)|)},$$

where 0 denotes a very uneven load between the partitions, and 1 denotes a perfect balancing.

- *Cut* is the fraction of edges that have endpoints located in different partitions. This represents an important quantity, as traversing such edge will require data to be exchanged among the machines, therefore adding latency to the request processing time. Formally, the cut is defined as:

$$1 - \frac{|\{(a,b) \in E | a \in P(i), b \in P(j), i \neq j\}|}{|E|}.$$

Again, 0 denotes a non-desirable situation where all edges have endpoints in different partitions, and 1 denotes a partitioning cutting no edge.

III. Partitioning: cuts, load and applications

Let us first consider a rough model of the environment of an application running in a centralized setting. Upon arrival of a request r, this application will consume two quantities: memory and CPU time. Let $m(r)$ and $c(r)$ be these quantities. In this abstract model, we consider that the system knows instantly at the arrival of r what will be $m(r)$ and $c(r)$.

The system is able to provide memory and processing power at rate μ and χ per time unit, respectively. Therefore, if r is the only request on the system, we consider that its processing will take the time required to gather the required resources $t_r = m(r)/\mu + c(r)/\chi$. As a simple model for congestion, assume that n_r requests are processed on the system at each time. Then t_r becomes $n_r.(m(r)/\mu + c(r)/\chi)$: the system evenly splits its processor and memory supply to all the requests, and side effects (such as context switch) are neglected.

Now let us consider the same application running in a distributed setting. Consider that the input of this application is a graph G: the memory requirements of a request r can now be expressed as a subgraph of G: $m(r) \subset G$. We consider the following distributed setting: k machines are fully connected through synchronous equal links. Each machine i has enough memory to hold a subgraph $P(i) \subsetneq G$ such that $\{P(i)\}_{1\leq i\leq k}$ forms a partition of G.

A. Analyzing a Simple Locality Model

Let us assume the memory needs of a request consist in the ℓ-hop neighborhood of a node: $m(r) \simeq B(v,\ell)$, where v is the center of request r. We define ℓ as a measure of requests' *locality*. Let us illustrate this concept:

- the request "get v's neighbors" has a locality of 0: the neighbors of v are known locally by v.
- the request "get v's eccentricity" has a locality of D, the graph diameter, as the most eccentric nodes are D hops apart.

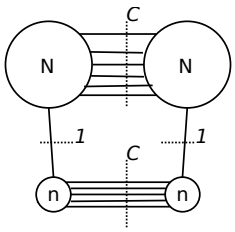


Figure 2. A vicious graph for partitioning methods. Two contradictory decisions can be made: favoring load balancing OR cut ratio.

- a damping random walk (jump with a probability $\alpha < 1$) can be modeled by an "expected" locality (e.g. $\ell = \lceil -1/\log(\alpha) \rceil$).

Let p be the machine holding node v, center of a request r of locality ℓ. If $B(v,\ell) \subset P(p)$, all the required information to process r is already available on p, the request processing time only depends on the processing resources available on p. However, if $\exists q, B(v,\ell) \cap P(q) \neq \varnothing$, then information will have to be fetched from machine q, and the duration of this fetch will add up to the request processing time.

More formally, let λ be the network latency induced by such fetching operation. We consider that remote fetches cannot be made parallely, mostly because in the streaming context the ℓ hop neighbors (and therefore the partitions holding them) are not known in advance besides direct neighbors. Therefore we model the processing time of request r when processed by p as:

$$t_r = \underbrace{\frac{c(r)|P(p)|}{\chi_p}}_{\text{computing time}} + \underbrace{\lambda|\{j \neq p, \text{s. t. } P(j) \cap B(r,\ell) \neq \varnothing\}|}_{\text{information gathering time}}.$$

Observe that the computing time contribution depends on the size of $P(p)$ since the bigger the partition is, the more requests machine p will have to serve in parallel. The information gathering time also depends on $P(p)$ since the bigger $P(p)$ is, the higher the chances are that $B(v,\ell) \subset P(p)$, therefore reducing the information gathering time to 0.

Thus, we have here a first visible **tradeoff** the partitioning strategy has to solve in order to minimize request compute time:

- Computations over small partitions are processed faster, since the load on the machine holding the partition is low, at the cost of higher information gathering costs.
- Computation over big partitions are slower, but require on average less information fetching.

Now, considering that we have a fixed number of machines k, this tradeoff translates in: *shall we prefer to minimize the cut or to optimize the load balancing ?*

B. Graph-Related Tradeoff

With the aforementioned tradeoff in mind, consider the graph depicted figure 2. This graph consists in 4 fully

connected clusters of sizes N, N, n and n. Clusters of equal size are connected by C links, and two links connect one cluster of size N with one of size n. Assume that $N > n$ and $n > C > 1$. Two key observations are:

- Any exactly balanced bisection (*i.e.* two partitions G_1, G_2 such that $|G_1| = |G_2| = (N + n)$) of the graph cuts at least $2C$ links. Let P_{WB} such partition, symbolized by Cs on Figure 2.
- The graph is 2-connex. The minimal edge-cut is 2 and has a balancedness $\min(|G_1|, |G_2|)/\max(|G_1|, |G_2|)$ of n/N. Let P_{MC} such partition, symbolized by 1s on Figure 2.

Now let us compute the average request processing time $\mathbb{E}(t_r)$ centered on a node v. Since we have only two clusters, assuming $\ell \in \{0, 1\}$, computing the information fetch cost is easy. Let $\mathcal{B}$ be the boundary of each cluster, and $\phi = c(r)$.

Assume well balancedness is preferred:

$$\mathbb{E}(t_r|P_{WB}) = \frac{\phi(n+N)}{\chi} + \lambda\ell \Pr(v \in \mathcal{B}) \quad (1)$$

$$= \frac{\phi(n+N)}{\chi} + \lambda\ell\frac{2C}{n+N}. \quad (2)$$

Now assume cut minimization strategy is preferred:

$$\mathbb{E}(t_r|P_{MC}) = \Pr(v \in P_1)\frac{\phi|P_1|}{\chi} + \Pr(v \in P_2)\frac{\phi|P_2|}{\chi} + \quad (3)$$

$$\lambda\ell \Pr(v \in \mathcal{B}) \quad (4)$$

$$= \frac{N}{n+N}\frac{2N\phi}{\chi} + \frac{n}{n+N}\frac{2n\phi}{\chi} + \lambda\ell \Pr(v \in \mathcal{B}) \quad (5)$$

$$= \frac{2\phi(n^2+N^2)}{\chi(n+N)} + \lambda\ell\frac{2}{n+N}. \quad (6)$$

If we compare those two quantities we have:

$$\mathbb{E}(t_r|P_{WB}) \leq \mathbb{E}(t_r|P_{MC}) \quad \Leftrightarrow \quad (7)$$

$$\phi(n+N)^2 + 2\ell\lambda\chi C \leq 2\phi(n^2+N^2) + 2\ell\lambda\chi \quad \Leftrightarrow \quad (8)$$

$$2\ell\lambda\chi(C-1) \leq \phi(n^2+N^2-2nN) \quad \Leftrightarrow \quad (9)$$

$$2\ell\lambda\chi(C-1) \leq \phi(n-N)^2 \quad \Leftrightarrow \quad (10)$$

$$\frac{2\ell\lambda\chi}{\phi} \leq \frac{(n-N)^2}{C-1}. \quad (11)$$

Therefore, in such a setting, one can draw the following conclusions:

- if the problem is only local ($\ell = 0$), well balancedness is always faster.
- under this last form, the inequality's left hand side only contains application and hardware dependent variables, that are unlikely to change along the progress of the stream. The inequality's right hand side only contains graph dependent variables: these are likely to evolve along the streaming progress.

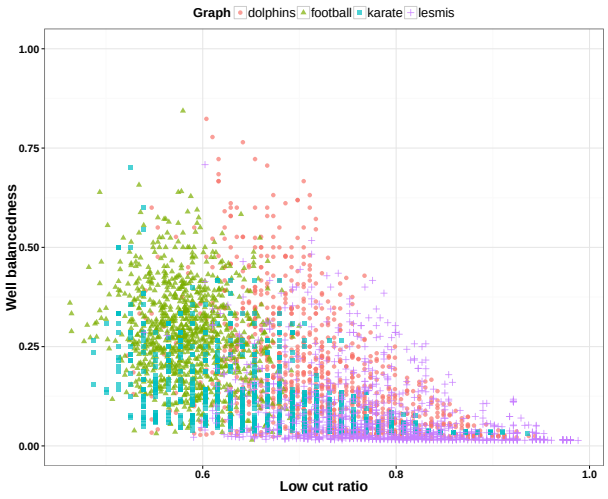


Figure 3. Reachable configurations while partitioning 4 different graphs, under the load balancing vs cut ratio tradeoff. Each point represents a particular configuration: each graph as it own particular set of "good" configurations (positions on the top-right envelope are the desirable ones).

This simple model illustrates a major problem of classical graph-partitioning approaches when dealing with the incremental nature of streamed graph: the fastest partitioning strategy depends **both** on the target **application** and on the target **graph**. As at least the graph is unknown, meaning that we cannot make strong assumptions on the evolution of its characteristics or on the order on which will be received particular updates, we therefore argue that an online exploration of this well balancedness/min-cut tradeoff is mandatory for top-application performance.

One might wonder, due to the artificial nature of the constructions used to illustrate this tradeoff, whether such situation do happen in practice. To answer this question, we took a set of standard small real-world topologies[1], and randomly partitioned them into 4 pieces, and measured the obtained well balancedness and cut size. We repeat the process $1,000$ times for each topology, therefore "sampling" the partitioning configuration possibilities. Figure 3 represents the obtained results. Each point represents a random partitioning, and although chances are low that optimal partitionings are represented on this figure, the point cloud associated with each topology represents the likely outcomes of partitionings. As we can see, they differ from one topology to the other: each topology allows its own tradeoff between well balancedness and low edge cut.

IV. (Re)partitioning from streamed graphs

We have seen in previous sections that the hardness of partitioning precludes a new partitioning iteration on the

[1]Topologies are available at http://www-personal.umich.edu/~mejn/netdata/. The authors would like to thank M.E.J. Newman for providing these topologies.

whole graph at each new edge arrival. We have established that the partitioning achieving the lowest request processing time has to find an optimum between a good load balancing, and a low edge cut. Moreover, we have seen that such optimum not only depends on the application, but also on the characteristics of the streamed graph.

We now propose a greedy solution for partitioning and reconfiguration, that handles increments as graphs are streamed.

A. Global Framework Overview

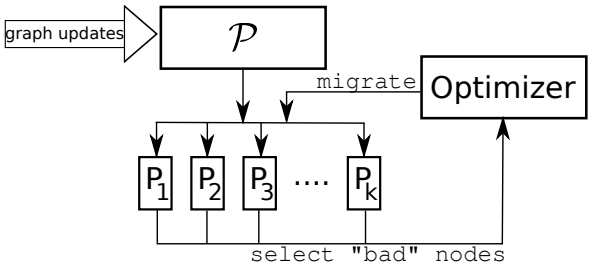


Figure 4. Overall system overview

The framework we propose is the following one, as shown on Figure 4. The stream of incoming graph updates (edge additions) is dispatched to the partitions by $\mathcal{P}$. Since the optimal balance between load balanced partitions and low cut ratio depends on the graph (which is constantly evolving) and the application (whose sweet spot for performance is not necessarily known), and since the continuous stream of updates has to be handled as quickly a possible, the partitioner decides which partition to assign new edges to in a greedy fashion. A logical optimizer monitors the state of the machines, and periodically optimizes this greedy layout by choosing nodes in partitions, and by migrating them. The choice of which nodes to move, and of where to migrate them is driven by an optimization procedure following two strategies: it tries to improve either the edge cut, either the load balancing. The precise role of each component is described afterwards.

B. Stream-greedy: A Simple Greedy Partitioner

We first introduce the two related work competitors for efficient online partitioning.

1) Related Approaches: The common approach, due to its simplicity, is to partition at random. Once a node arrives at the datacenter, a partition is selected according to a modulo operation over the node identifier (itself being pseudo-random), see *e.g.* Pregel [6]. This often called *hashing*.

More advanced heuristics for partitioning a graph that is read from disk as a stream of nodes, are presented in [7]. They all outperform hashing. The best performing heuristic is called *weighted deterministic greedy*, and is node driven. It consists of placing an incoming node v to the partition

Initialize k partitions with Capacity C;
For each incoming edge e_{ij}:
if $\exists i \; and \; \exists j$ **then**
  $addLink(i, j)$
else if $\exists i \; and \; \nexists j$ **then**
  Place j in $P(i)$ if not full. Otherwise, place j at the least occupied partition. $addLink(i, j)$
else
  //i.e. $\nexists i \; and \; \nexists j$
  Place both i and j at the least occupied partition.
  $addLink(i, j)$

Figure 5. **stream-greedy** heuristic for streamed graph partitioning.

j where it has the most edges, weighting this by a linear penalty function based on the capacity of the partition:

$$j = \arg\max_{i \in [k]} (|P(i) \cap \Gamma(v)|(1 - \frac{|P(i)|}{C})). \qquad (12)$$

All methods stream nodes in random order or in a breadth/depth first search fashion. The major assumption of this technique is that each streamed node comes with its complete (non-yet seen) neighbor-list. This make this study not applicable to the stream processing model where edges are received as events at a datacenter along the application life (as opposed to nodes read from a disk containing the whole graph).

We pick this deterministic greedy heuristic as the **baseline** technique to compare to in the rest of this paper.

2) stream-greedy partitioner: As we are bound to a restrictive model where edges are received arbitrarily following the application logic, and because a crucial point for system scalability is to propose a fast and lightweight partitioning method at $\mathcal{P}$, we detail a simple and intuitive greedy partitioner: when e_{ij} arrives at the datacenter, placement decision is made following the pseudo-code provided on Figure 5. $addLink(i, j)$ is a function triggered by $\mathcal{P}$, that informs machines hosting i and j (they could be one single machine) to link those two nodes via and edge in their respective adjacency lists.

Computationally, this heuristic simply require $\mathcal{P}$ to perform membership tests and cardinality operations over the mapping table (for finding the least represented partition).

C. Improving Current Partitioning

To improve the current partitioning, the optimizer relies on two heuristics. The first targets an improvement on the load balancing criterion, while the second targets an improvement on the cut criterion. The simulation results presented Section IV-D show that both heuristics are efficient most of the time: each one improves one metric without significant degradation of the other.

Both heuristics select nodes of machines based on their "badness", which measures each node's individual contribution to the size of the cut. Rationale behind using the same selection being that even if load balancing is not concerned by cut ratio, selecting bad nodes instead of random ones does not degrade intentionally the second metric. For each node v in a partition i, it is formally defined as:

$$badness(v) = \frac{|\Gamma(v) \cap P(i))|}{|\Gamma(v)|}, \tag{13}$$

(the lower the worse). However, while improvement towards cut selects the same amount of nodes in each machine, the load-balancing improvement heuristic only selects nodes on the most loaded machines.

The selected nodes are then migrated to a different machine. Again, heuristic improving cut proceeds while ignoring the machine loads: it migrates each selected node to the partition j containing the most of its neighbors:

$$j = \underset{i \in [k]}{\arg\max}(|P(i) \cap \Gamma(v)|). \tag{14}$$

In contrast, the load balancing heuristic takes each machine load into account when selecting a new partition for each node: for this it relies on the **baseline** strategy (12). Note that we can apply **baseline**, as when this heuristic is executed, the system deals with already received edges and nodes for reconfiguration.

Afterwards, both heuristics proceed the same way. They send the node to the selected partition, and update $\mathcal{P}$ regarding the node's new location.

Given the state of the partitioning at a given time, improving on one criterion means reconfiguring the system, by moving a small part of system nodes from one partition to another one. The migration rate must be small, in order not to impact the application running on top of the partitioning. The optimizer measures the application performance at runtime, decides when to reconfigure, chooses a reconfiguration strategy, and commits it if it considers this reconfiguration was successful. In order not to force the developers to over instrument the application, and for overall simplicity/reusability, we built a system that self-tunes solely based on one information given by the application at runtime. This information takes the form of the average computation time. From this value, the optimizer decides when to trigger a reconfiguration, and measures after a reconfiguration if the average runtime is lower or not than before a reconfiguration.

The optimization problem is then to reach a configuration, *i.e.* a certain graph partitioning, that minimizes request execution time at the application. As finding a particular graph partitioning is NP-complete (*e.g.* bisecting static graphs [8]), we have to rely on local search optimization. Considering our computing time feedback, and two improvement criterions, a natural optimization framework is **hill climbing**. The difference of our setup with canonical hill climbing is that we cannot instantly evaluate both neighbors of current configuration, *i.e.* the new configuration after a step on load balancing and after a step on cut ratio. We thus make a random choice towards one criterion, and act as a function of resulting compute time. We call this variant **blind hill climbing**.

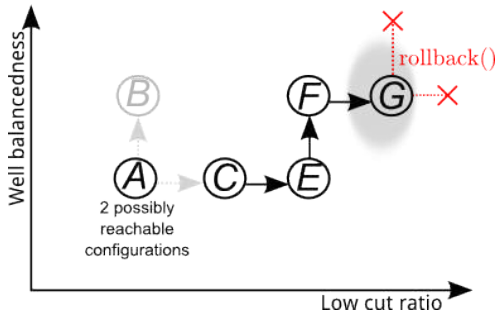


Figure 6. Blind hill climbing optimization over current partitioning. **stream-greedy** leaves the system in configuration A. Ideal configuration is top-right (shaded area). From A, running a heuristic for improvement would lead to configuration B or C. Random coin flip selects heuristic to improve on cut ratio. Process is iterated (going from C to E, F and G), until none of the two heuristics can improve upon G. Rollbacks are operated for return to G, waiting for substantial graph growth before new optimization trials.

This approach is summed-up by Figure 6; periodic optimizations are conduced under trial and error. When no progress is possible in any of the two directions, the configuration has reached a sweet spot for the application performances. Note that as nodes and edges arrive continuously at the datacenter, this spot moves after each addition. Each optimization step then performs in best effort fashion voluntarily considering current graph as static.

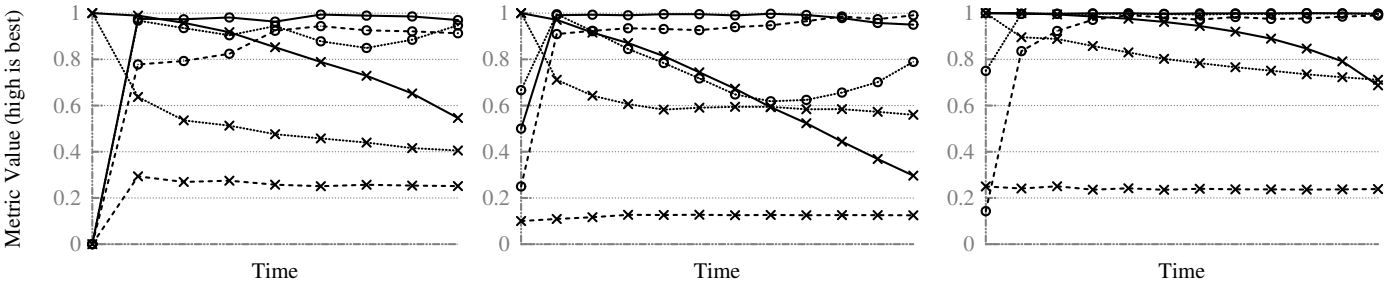
Finally, note that we presented the optimizer as an independent centralized entity for the sake of clarity. In practice, the optimizer task is distributed on each machine.

D. Simulation Results

1) Simulation Setup: For the sake of comparison with simulation results conduced in [7], we use the same three graphs depicted as representative from the major three types of network structures. The first one, PL1000, is a synthetic graph of $1,000$ nodes with clustered power-law characteristics[2]. The social network, Marvel, consists in $6,486$ characters, having $427,018$ interactions in comics. The last graph, 4elt, of $15,606$ nodes and $45,878$ edges, is a FEM graph from the NASA.

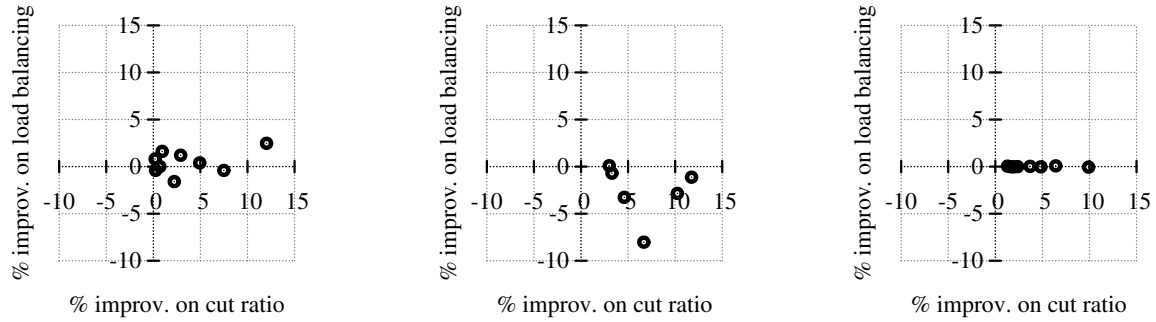
For all three datasets, as the graph is streamed in a random order from the first edge to the last one. The number of partitions is set to 4 for PL1000, 8 for Marvel and 4 for 4elt, as in [7] for matters of reproducibility. Our greedy heuristic, as well as random and **baseline** results are exposed. Please note that as **baseline** assumes that when a node arrives at the

[2]generated with NetworkX (http://networkx.github.io/)



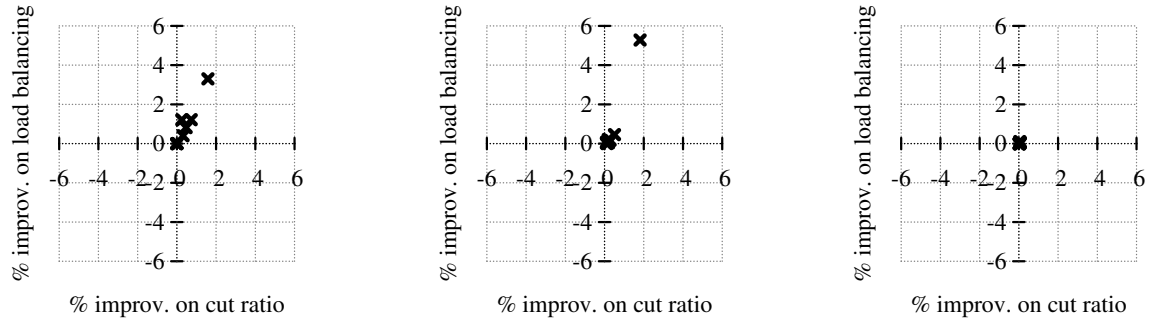
(a) PL1000 graph, $k = 4$. (b) marvel graph, $k = 8$. (c) 4elt graph, $k = 4$.

Figure 7. **stream-greedy** partitioner (solid line), random (dashed line) and **baseline** (dotted line) competitors, on 3 representative graphs. Load balancing (circles) and cut ratio (crosses) are plotted (the higher the better on the y-axis) for each approach, as the graph is streamed from its first edge to its last one (x-axis).



(a) PL1000 (b) marvel (c) 4elt

Figure 8. Improvement on cut ratio and load balancing on greedily partitioned graphs, produced by successive runs seeking to improve **cut ratio**. Each point represents the percentage of improvement obtained considering current configuration and then executing heuristic to improve on cut. Ideally, points should follow the positive y-axis. The rightmost point on (a) means that a single call to heuristic improved previous cut by 12% and also improving load balancing 2.6% as a side effect.



(a) PL1000 (b) marvel (c) 4elt

Figure 9. Improvement on min cut and load balancing, produced by successive runs seeking to improve **load balancing** Ideally, points should follow the positive x-axis (setting similar to Figure 8).

partitioner, all its (future) neighbor nodes are also known, we implement such assumption in our simulation. This clearly gives it an advantage over the two other approaches, but also serves as an indicator of the gap between greedy methods and the one with full knowledge.

2) stream-greedy performance: We now assess the performances of random, **baseline** and **stream-greedy** heuristics by simulation, in a streamed graph setup.

Figure 7 present results for well balancedness (lines with rounds) and cut (lines with crosses) metrics on the y-axis,

the higher the better. The x-axis represents time, from $t = 0$ where first edge is dispatched, to t_{end} where the whole graph has been streamed (then having G_∞ partitioned).

Regarding load balancing curves, results for all three graphs and all three competitors are consistent and close to perfect balancing at t_{end}, and all three are comparable. Now regarding cut, we first remark that **baseline** results are consistent with results produced in [7] (where only cut ratio is considered), as operating under the same assumptions. Produced results for random partitioning are also consistent with theory, as random placement is awaited to cut a fraction of $3/4$ of edges for $k = 4$ and $7/8$ for $k = 8$. The first learning is that baseline always beats random partitioning (as seen in [7]); this is also the case for our greedy approach. Surprisingly, despite the fact that **baseline** beats **stream-greedy** on Marvel and slightly on 4elt graphs, **stream-greedy** outperforms the deterministic greedy approach on PL1000. This shows that even with an increased amount of information, **baseline** does not perform better, as it could be beaten by **stream-greedy**, even operating on less information. We thus learn that operating under reduced assumptions still make possible a very competitive first partitioning step, using an intuitive partitioner such as **stream-greedy**.

We have presented and compared a simple one pass greedy stream partitioner; as a consequence, edges placed by that heuristic are never moved from one partition to another one. Next section shows that we can periodically improve on one criterion or the other at runtime.

3) Improving on Cut Ratio: Figure 8 presents improvement percentages for the cut criterion. Ideally, running a heuristic on one criterion (*e.g.* on cut ratio) should of course improve it, but also avoid degrading the second one (*e.g.* not degrade load balancing). Simulations show that, on the same three graphs at t_{end}, there is room for improvement. With a typical value for top-K worst nodes of 10%, up to 12% improvement is achieved at each procedure call, after what next calls produce more slight changes. We also see that except for few percent on the Marvel graph, improving on cut ratio does not degrade current load balancing of graphs.

4) Improving on Load Balancing: Figure 9 present results. Improvement on load balancing at each call is positive but less important that for improvement on the cut ratio criterion. This is explained by the fact that at t_{end}, as seen on Figure 7, load balancing is already close to perfect. Those calls does not degrade the other criterion either.

V. Illustration: graph-based based instant recommender

We now present a direct application of this optimization framework in a system leveraging streamed graphs, detail implementation needs, and show performance indicators.

A. System Implementation of Blind Hill Climbing

Regardless of the application relying on the partitioning process, few system primitives are required to implement

```
while True do
    c_before ← getComputeTime();
    snapshot();
    buffer();
    if Random(cut,balancing) == cut then
        OptimizeOnCut()
    else
        OptimizeOnBalancing()
    c_after ← getComputeTime();
    if c_after > c_before + ε * c_before then
        rollback();
    else
        commit();
    flushBuffer();
```

Figure 10. Blind hill climbing optimization, a generic approach for improving computation time in stream-enabled applications.

the optimization framework, here is their detail:

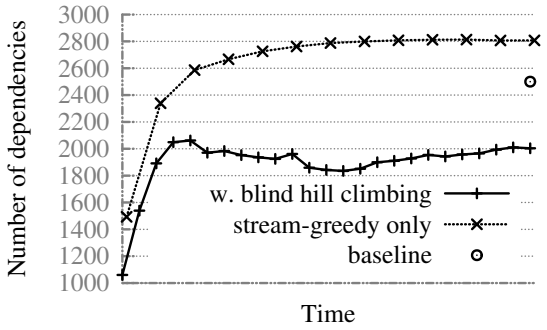
- `snapshot()`: atomically records node/edge repartition over machines
- `getComputeTime()`: top application returns current average compute time
- `commit()`: remain in current configuration, and free structure used for snapshoting
- `rollback()`: return to previous configuration (snapshoted earlier)
- `buffer()`: record all incoming events (edges, requests) in a message queue. `flushBuffer()` consumes those buffered events.

From those primitives, we propose the following heuristic, that pursue a hill climbing optimization based on application feedback (see Figure 10).

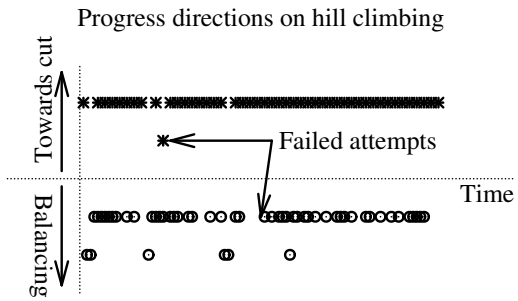
As configuration switch has practical costs, system parameter ϵ denotes the degradation threshold over which it is profitable to rollback to previous configuration. This threshold also masks the slight deviation in average compute time due to particular request patterns, or due to the arrival of nodes and edges in between two optimizations.

B. A System for Instant Recommendation

A typical example of a latency critical service is recommendation. The reactivity capabilities of the application to process fresh data is key for successful services, as accurate and instant recommendation based on previous clicks for instance [9]. We take the scenario of movie recommendation for the rest of this paper. In this framework, nodes are users/movies and an edge is present if one user has rated a particular movie. We are interested in treating ratings as a stream, as their direct incorporation into the system for instant recommendation can for instance solve the cold start problem at user registration (*i.e.* before waiting for the platform to compute offline for later personalized recommendations).



(a) Actual improvement of blind hill climbing optimization over a real dataset (MovieLens, $k = 8$).



(b) Success of blind hill climbing optimization: each point on the two outer lines are successes, while inner lines represent failed optimization steps.

Figure 11. Simulation of the blind hill climbing optimization framework, over the streamed MovieLens dataset.

Bahmani et al. [2] show that a variant of Pagerank, based on random walks, provides fast and personalized recommendations. On a user/item graph, it essentially consists in launching multiple (damping) random walks from the node (user) that needs a recommendation; most visited nodes (items) are extracted through higher Pagerank values and ranked. On such a graph, top-nodes (items) are good recommendations for that user.

As random walks are locality critical in the context of parallel computing, we expect partitioning to be crucial for the recommender performances. With a good partitioning (high density of links within machines and low edge cut), launched damping random walks stay on the same machine[3]. If not, then many dependencies arise from inter-machines communications.

C. Simulation Over the MovieLens Dataset

We ran personalized Pagerank for recommendation over the MovieLens dataset [10], a user/movie graph consisting of $100,000$ ratings from $1,000$ users on $1,700$ movies.

The blind hill climbing optimization, described on Figure 10, is implemented by a logical optimizer in the following way. $\mathcal{P}$ controls the frequency of optimizations. When $\mathcal{P}$ decides to perform an optimization step, it flips a coin to choose on which criterion to try to improve. It then stops consuming events by calling the `buffer()` primitive (architecturally this correspond in practice to let the message queue store events, *e.g.* [11], and let it grow for an instant). Order to call improvement heuristic is broadcasted to the k machines. Bad nodes are computed (according to metric (13)) on each machine, in order not to add computational burden onto the partitioner. On this simulation 10% of the worst nodes of each partition are migrated in each optimization cycle. Once optimization has completed on all machines, a generic batch of requests is artificially executed on the system. If `getComputeTime()` returns a degraded average completion time, the partitioner broadcasts a rollback order (a commit one otherwise). The rollback to previous partitioning configuration is executed in a lockstep at each machine, by a simple memory overwrite from previous `snapshot()` result. $\mathcal{P}$ then return in service mode by called `flushBuffer()`. This simple implementation constitutes a coarse grain handling of system states; we leave a finer grain handling of consistency for future work.

The primary metric to assess performance of the application related to a given partitioning is the number of dependencies. A dependency of the application occurs when a random walk (launched at any machine) has to get onto another machine to pursue process. It is a direct indicator of the latency at the application level, as network costs clearly overcome local CPU computation.

Each simulated request triggers $1,000$ random walks (with a min-hop of 3 away from the querying user, and a damping factor of $\alpha = 0.9$). To simulate the increase of application-usage as the graph grows, we ran a number of user requests of 1% times the current graph size, after each increase of the number of edges by 5%. With an optimization step being triggered when graph size increases by 5%, $\epsilon = 1\%$, and $k = 4$, dependencies are plotted on Figure 11(a), as they evolve while the graph is streamed. The hill climbing optimization clearly maintains dependencies at a lower level that **stream-greedy** (close to 30% less). The static partitioning resulting from **baseline** performs again slightly better than **stream-greedy**, but as its operation is not periodically optimized, it cannot compete with our blind hill climbing approach.

We now plot the on Figure 11(b) the success/failure of

[3]We prototyped a Storm-based system [4] implementing this recommendation engine, and answering to PUT/GET queries on the MovieLens graph. While adding and edge is made instantly, average computing time for single sequential GET recommendations is around 150ms (using as much as $50,000$ random walks per recommendation), over a Xeon E5-2603 CPU (DELL T5600). This motivates the will to keep computation local to a machine for both speed and capability to handle many concurrent requests.

calls to improvement heuristics. For 100 optimizations on a random criterion, 93 are successful on cut ratio criterion, and 6 on load balancing. There is only 1 rollback on cut, but 47 on load balancing. The more important failure rate over load balancing is due to the already very goad balance achieved without optimization, as seen on Figure 7 for other graphs with **stream-greedy**. A solution to decrease this failure rate is to bias the random choice towards the more successful of the two criterion (*i.e.* learn and call the most successful one with a higher probability).

In conclusion, there is a clear advantage in periodically reconfiguring current partitioning by calling lightweight optimization procedures, for correcting past greedy decisions in the context of streamed graphs. The framework we propose only takes as input the application feedback and allows self-tuning in an efficient way.

VI. Related Work

Static partitioning approaches take a graph as input and propose a bisection as an output (know as the minimum bisection problem). Reaching a configuration with minimal number of inter-machine edges while balancing the output is well known to be NP-complete [8]. This as been extended to k-partitioning, where the input graph is partitioned into k pieces [5]. Methods for partitioning largely depends on the application using the produced partitions, as computing while partitions fit network architecture [12], minimizing interactions between storage servers [1], or computing over embarrassingly parallel datasets [6] for instance.

Approaches like GraphChi [13] or TurboGraph [14] aim at computing metrics over large graphs on a centralized setting; they differ from stream-based approaches as they compute offline over the dataset and do not consider graph updates for low latency operation and online request handling.

Staton et al. [7] are the first to consider a stream of nodes to be placed onto partitions on the fly. Many heuristics are proposed and tested, from intuitive ones (considering balance) to more advanced ones (considering clustering coefficient); we re-implement the best performing one in this paper to compare it to our proposal. All approaches are one pass; a placed vertex is never moved afterward. Their paper assumes a full knowledge model, where the graph to be streamed has to be present on one machine prior to the execution of the proposed heuristics.

VII. Conclusion

This paper has exposed the hardness of partitioning a streamed graph not already present on a computing device. A greedy partitioner taking as input stream of edges has been proposed, that can compete with a state of the art heuristic for partitioning under full knowledge. While its operation is satisfying, we show that it is of interest to periodically call an optimization procedure to improve upon current partitioning, on the edge cut ratio or on the load balancing criterions. This building blocks form a general optimization framework allowing for application self-tuning, based solely on feedback on compute time. An interesting question for future work is to formally ask if there exists a greedy algorithm having provable bounds for its efficiency to partition a streamed graph, in the classic stream processing model.

References

[1] Pujol, J.M., Erramilli, V., Siganos, G., Yang, X., Laoutaris, N., Chhabra, P., Rodriguez, P.: The little engine(s) that could: scaling online social networks. IEEE/ACM Trans. Netw. **20** (2012) 1162–1175

[2] Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. In: VLDB. (2010)

[3] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51** (2008) 107–113

[4] Marz, N.: Storm Project. http://storm-project.net/ (2012)

[5] Andreev, K., Räcke, H.: Balanced graph partitioning. In: SPAA. (2004)

[6] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD. (2010)

[7] Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: KDD. (2012)

[8] Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)

[9] Linden, G., Smith, B., York, J.: Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Computing **7** (2003) 76–80

[10] Herlocker, J.L., Konstan, J.A., Borchers, A., Riedl, J.: An algorithmic framework for performing collaborative filtering. In: SIGIR. (1999)

[11] Cook, B.: Kestrel distributed message queue. https://github.com/robey/kestrel/ (2013)

[12] Ajwani, D., Ali, S., Morrison, J.: Graph partitioning for reconfigurable topology. In: IPDPS. (2012)

[13] Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: large-scale graph computation on just a pc. In: OSDI. (2012)

[14] Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In: KDD. (2013)