# Application-aware adaptive partitioning for graph processing systems

Erwan Le Merrer, Gilles Trédan

# Application-aware adaptive partitioning for graph processing systems

Erwan Le Merrer
*Univ Rennes, Inria, CNRS, IRISA*
Rennes, France
erwan.le-merrer@inria.fr

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

***Abstract*—Modern online applications value real-time queries over fresh data models. This is the case for graph-based applications, such as social networking or recommender systems, running on front-end servers in production. A core problem in graph processing systems is the efficient partitioning of the input graph over multiple workers. Recent advances over Bulk Synchronous Parallel processing systems (BSP) enabled computations over partitions on those workers, independently of global synchronization supersteps. A good objective partitioning makes the understanding of the load balancing and communication trade-off mandatory for performance improvement. This short paper addresses this trade-off through the proposal of an optimization problem, that is to be solved continuously to avoid performance degradation over time. Our simulations show that the design of the software module we propose yields significant performance improvements over the BSP processing model.**

## I. Introduction

Graph-based applications such as social networks [10], search engines or recommender systems [2] have to deal with giant and constantly evolving networks of user or item interactions. Data-processing over those graphs include graph-oriented operations, such as PageRank or shortest paths [8]. More local vertex-oriented operations, where computations are centered on some particular vertices, are of particular interest for instance in the support of social networks. This includes users fetching their friends' profiles [10] (one hop retrieval), getting recommendations [2], or being assigned an influence metric such as TunkRank [16], [3]. Such operations are attractive for front-end Internet applications, as query latency is low as compared to global operations, and as the graph can be updated in real time to reflect user interactions on the platform. This paper specifically values those vertex-oriented computations.

In order to ensure scalability when running such applications on a set of workers, different execution models were considered by system designers. Due to it's widespread use in datacenters, the Bulk Synchronous Parallel (BSP) [15] paradigm is the de-facto execution model for actual state-of-the-art graph processing systems [8], [1]. Notably, Giraph++ [14] introduced an alternative execution model, named BSP-hybrid. In a nutshell, this paradigm is as follows: like in BSP, BSP-hybrid workers each hold a partition (*i.e.* a sub-graph) of the graph; they process the requests related to this partition. Both BSP and BSP-hybrid are organized is supersteps, and inter-worker communication can only be made at the end of a superstep. Yet, unlike BSP where intra-worker communications are also delayed until the end of the current superstep, BSP-hybrid systems allow intra-worker communication to be done within the current superstep. In other words, the improvement made by BSP-hybrid is to allow algorithm steps to proceed between vertices belonging to the same partition, without being blocked by the global synchronization superstep. Therefore, requests can be processed in only one superstep provided they do not require inter-worker communication: the partitioning of the graph becomes a salient feature for fast request processing.

The quality of the partitioning method is thus at the core of BSP-hybrid efficiency: well balanced partitions cause equivalent completion times for the different workers, preventing the last-reducer curse [13]. Simultaneously, avoiding too many graph edges in between workers allows most requests to be fulfilled locally by a worker within a superstep while reducing the network usage. These two objectives are commonly captured by the following metrics: load balancing (noted $LB$ hereafter) and edge cut (noted $C$), and formalized as follows: $LB = \frac{\min_{i\in[k]}(|P(i)|)}{\max_{i\in[k]}(|P(i)|)}$, $C = 1 - \frac{|\{(a,b)\in E \text{ s.t. } a\in P(i), b\in P(j), i\neq j\}|}{|E|}$, where workers host and process over a partition of the $G = (V, E)$ graph, $P(i)$ denotes the partition number $i$, and $|P(i)|$ its cardinality in terms of graph nodes.

The importance of a good partitioning method, for providing good values for $LB$ and $C$ metrics is highlighted in the paper introducing Giraph++ [14]. Yet, the relationship between those two metrics and the performances of the BSP-hybrid processing model is unexplored. Secondly, for vertex-oriented applications, graph persistence in the processing system is mandatory for avoiding constant graph reloading (see *e.g.* [3], [10]). This questions the possibility to migrate vertices between workers at runtime, in the context of a constantly evolving graph; the relationship between $LB$ and $C$ thus becomes particularly important.

This paper addresses this unexplored relationship between $LB$ and $C$ in the context of a BSP-hybrid processing model, and proposes the design of a software module for further processing time improvement. Our contributions are: *(i)* to exhibit the graph-related tradeoff between $LB$ and $C$ in such a processing model. This tradeoff directly translates into the performance of applications exploiting that graph; we illustrate this principle through analysis in Section II. *(ii)*

Given the previous observation, to consider the problem of an adaptive partitioning method as an optimization problem, in Section III. In this problem, the optimization choices are either $LB$ or $C$, and the metric to optimize is the application performance, measured by the *average request processing time*. *(iii)* Finally, to propose a *blind hill climbing* optimizer for adaptive partitioning, in sub-Section III-B. This module implements arguably the simplest heuristic solution to the aforementioned optimization problem. It monitors the average processing time, and leads partitioning towards $LB$ or $C$ improvement strategies. We finally show through simulations, in Section IV, that such module significantly improves the performance of a BSP-hybrid system.

## II. Impact of Partitioning in the BSP-hybrid Model

In BSP systems, any vertex-to-vertex communication will be delayed until the end of the current superstep, be it intra-worker or not. As a superstep ends when the last worker is done, a high stress is put on $LB$; $C$ is thus taking a back seat as a network congestion consideration. In the BSP-hybrid model, dense sub-graphs on workers allow for in-worker computation between supersteps, and even partition sizes imply similar completion times and no straggler workers: both $C$ and $LB$ are simultaneously valuable. Yet, we show in this section that $C$ and $LB$ often cannot be improved simultaneously.

### A. Load/Cut Trade-off Impact on Performance

We now propose a simple model, intended to capture the relationship between aforementioned concurrent objectives.

*1) System Modeling:* As a BSP distributed application, we assume $k$ workers connected through synchronous equal links dedicated to an application running on top of a partitioned graph $G$. We hereafter coarsely model its environment. To fulfill a request $r$, we consider that this application only consumes two quantities: data $m(r)$ and CPU cycles $c(r)$. Every worker is able to provide CPU cycles at rate $\chi$ per time unit. If $r$ is the only request on a single worker ($k = 1$), all the required data is available locally and instantly, therefore $t_r = \frac{c(r)}{\chi}$.

As a simple model for congestion, assume that every worker evenly splits its processor time to all the requests it has to handle (side effects such as context switches are neglected), and that the number of requests arriving at worker $i$ is proportional to $|P(i)|$: a worker $i$ provides CPU cycles to each request at rate $\frac{\chi_i}{|P(i)|}$.

The data requirements of a request $r$ are modeled as the $\ell$-hop neighborhood of a node (*i.e.* a closed metric ball $B$): $m(r) \simeq B(v, \ell) \subset G$, where $v$ is the center of request $r$. We define $\ell$ as a measure of requests' *locality*. Let us illustrate this concept: the request "get $v$'s degree" [10] has a locality of 0 (every node knows its adjacency list), whereas the request "get $v$'s graph *eccentricity*" (*i.e.* its greatest distance to any other vertex) has a locality of $D$, the graph diameter. A damped random walk (jump with a probability $\alpha < 1$) can be modeled by an "expected" locality (*e.g.* $\ell = \lceil \frac{-1}{\log(\alpha)} \rceil$).

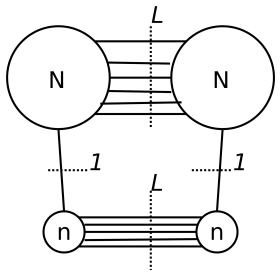



Fig. 1. Illustration of the load balancing ($LB$) versus edge cut ($C$) trade-off: contradictory decisions can be made, favoring one over the other.

Let $p$ be the worker holding vertex $v$, center of a request $r$ of locality $\ell$. 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 CPU resources available on $p$: $t_r = \frac{c(r)|P(p)|}{\chi_p}$. However, if $\exists q, B(v, \ell) \cap P(q) \neq \varnothing$, then information will have to be fetched from worker $q$, and the duration of this fetch will add up to the request processing time. Let $\lambda$ be the network latency induced by such a fetching operation. Remote fetches cannot be made parallel, mostly because the $\ell$ hop neighbors (and therefore the partitions holding them) are not known in advance beside direct neighbors. Therefore we model the processing time of request $r$ when processed by $p$ as:

$$t_r = \underbrace{\frac{c(r)|P(p)|}{\chi_p}}_{\text{computing time}} + \underbrace{\lambda|\{j \neq p, \text{s. t. } P(j) \cap B(r, \ell) \neq \varnothing\}|}_{\text{information fetching time}}. \quad (1)$$

Observe that the computing time contribution depends on $|P(p)|$ since the bigger the partition is, the more requests worker $p$ will have to serve in parallel. The information fetching 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 fetching time to 0. This is the **trade-off** that the partitioning strategy has to solve in order to minimize request processing 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 fetching costs. On the other hand, computations over big partitions are slower, but requires on average less fetching. We now illustrate how the data requirements of the application impact this trade-off.

*2) Trade-off Analysis Using a Pathological Graph:* With the aforementioned tradeoff in mind, consider the graph depicted in Figure 1. This graph consists in four fully connected clusters of sizes $N, N, n$ and $n$. Clusters of equal size are connected by $L$ links, and two links connect one cluster of size $N$ with one of size $n$. Assume that $N > n$ and $n > L > 1$. Two key observations are: *(i)* 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 $2L$ links. Let $P_{LB}$ such bisection, symbolized by $L$s on Figure 1. *(ii)* The graph is 2-connex. The minimal cut is 2 and has a load balance $\frac{\min(|G_1|,|G_2|)}{\max(|G_1|,|G_2|)}$ of $\frac{n}{N}$. Let $P_L$ such bisection, symbolized by 1s on Figure 1.

Now let us compute the average processing time $\mathbb{E}(t_r)$ of a request centered on a node $v$. Since we have only two clusters,

assuming $\ell \in \{0,1\}$, computing the information fetching cost is easy. Let $\mathcal{B}$ be the boundary of each cluster, and $\phi = c(r)$. Assume $LB$ is preferred:

$$\mathbb{E}(t_r|P_{LB}) = \frac{\phi(n+N)}{\chi} + \lambda\ell \Pr(v \in \mathcal{B}) = \frac{\phi(n+N)}{\chi} + \lambda\ell\frac{2C}{n+N}. \tag{2}$$

Now assume $C$ is preferred:

$$\mathbb{E}(t_r|P_C) = \Pr(v \in P_1)\frac{\phi|P_1|}{\chi} + \Pr(v \in P_2)\frac{\phi|P_2|}{\chi} + \lambda\ell \Pr(v \in \mathcal{B}) \tag{3}$$

$$= \frac{2\phi(n^2+N^2)}{\chi(n+N)} + \lambda\ell\frac{2}{n+N}. \tag{4}$$

If we compare those two quantities we have:

$$\mathbb{E}(t_r|P_{LB}) \leq \mathbb{E}(t_r|P_C) \Leftrightarrow \tag{5}$$

$$\phi(n+N)^2 + 2\ell\lambda\chi C \leq 2\phi(n^2+N^2) + 2\ell\lambda\chi \Leftrightarrow \frac{2\ell\lambda\chi}{\phi} \leq \frac{(n-N)^2}{C-1}. \tag{6}$$

Therefore, in such a setting, one can draw two observations: the confirmation that if the problem is only local ($\ell = 0$), a partitioning providing a good $LB$ is always faster (which corresponds to embarrassingly-parallel problems, as targeted by use-cases of the BSP model). On the contrary, a less local problem (*e.g.* users getting their influence rank) will benefit a lot from a low $C$. Second, final inequality (6)'s left hand side only contains application and hardware dependent variables, that are unlikely to change at runtime, even after graph updates. The inequality's right hand side only contains graph dependent variables: these are likely to evolve at runtime, as the graph evolves. A static partitioning strategy, that does not improve results of its past decisions, is thus likely to be sub-optimal. This calls for the design of an adaptive partitioning module, that we now present.

## III. An Optimization Module for Adaptive Partitioning

### A. Optimization Problem Statement

For navigating in this trade-off at runtime, a processing system has to employ a well defined function in order to choose one of the two concurrent directions for optimization ($LB$ or $C$). Let $\mathbb{E}^G$ be the *general* average processing time, *i.e.* for all requests centered on all nodes, during an arbitrary observation period. The optimization problem the graph processing system has to solve to maximize throughput in this model is therefore the minimization of:

$$\begin{aligned} \text{min.} \quad & \mathbb{E}^G\left(1/\chi \sum_{r\in R}\sum_{r'\in R} c(r)\delta_{q_{r'},q_r} + \lambda \sum_{r\in R}\sum_{j\in V} m_{rj}(1-\delta_{q_r,p_j})\right), \\ \text{with} \quad & 1 \leq p_i \leq k, \forall i \in V, \\ & 1 \leq q_r \leq k, \forall r \in R, \end{aligned} \tag{7}$$

where $R$ is the set of requests, $M$ is the matrix of data requirements for the requests $R$: it is a $|R| \times n$ matrix where $m_{ri} = 1$ if request $r$ needs $i$'s data to be processed, and 0 otherwise. Finally, $c(r)$ if the computing cost of request $r$, $\chi$ is the power of each worker (in computing units per timestep), and $\lambda$ is the network access latency (in timesteps), and $\delta$ is the Kronecker delta function. Vectors $(p_i)_{i\in V}$ and $(q_r)_{r\in R}$ respectively represent how the system allocates nodes and requests to workers. In practical contexts ($c(r)$ is not too big) a good strategy is to allocate requests at their center node, leaving node allocations the only input of this problem.

In this notation, one can express the cut as:

$$C = 1 - 1/m \sum_{i,j\in V^2} a_{ij}(1-\delta_{p_i,p_j}),$$

with $a_{ij}$ being the values of the adjacency matrix associated to the graph $G$ and $m$ the total number of edges of the graph. Load balancing can be expressed as:

$$LB = \frac{\min_{i\in 1..k}\sum_{r\in R}\delta_{i,q_r}}{\max_{i\in 1..k}\sum_{r\in R}\delta_{i,q_r}}.$$

With arbitrary input graphs and workloads, the impact of any partitioning algorithm is complex and does not permit to derive $\mathbb{E}^G(t_r)$ from a closed-form formula. This could be observed from our example model with Eq 1, where $LB$ and $C$ do not appear directly in the equation. We thus have to observe the resulting evolution of this metric based on the changes of both $LB$ and $C$ with time. Specifically, one can observe that $LB$ and $C$ respectively contribute to the left and right parts of the function to minimize (7); they therefore constitute natural optimization directions. In practice, as future requests cannot be predicted, an optimizer can 1) monitor $\mathbb{E}^G$ and 2) change the node allocations $(p_i)_{i\in V}$: migrate the nodes. The questions that remain are **which nodes must be migrated**, and **where to migrate them**. Heuristics exist for the second question [12], but are until now agnostic of current $\mathbb{E}^G$; we hereafter present a simple heuristic to answer both questions at runtime.

### B. An Optimizer Using Blind Hill Climbing

Given the state of the partitioning at a given time on workers, improving on either $LB$ or $C$ means re-configuring the system, by moving some nodes onto different workers. As finding a particular graph partitioning is a NP-complete problem (*e.g.* bisecting static graphs [5]), we have to rely on local search optimization. Considering our computing time feedback $\mathbb{E}^G$, and two improvement criterions, we seek a configuration on the *Pareto frontier* for the optimal choice between $LB$ and $C$. A classic optimization framework is *hill climbing*.

Among all possible optimization approaches, we choose to use hill climbing for two reasons. First, it is simple and provides a baseline for the approach: more elaborated approaches are likely to perform better for instance by leveraging previous runs to select the improvement direction. Secondly, the continuous updates of the graph perpetually prevent the system from converging to a stable state, therefore removing the local minima drawback of hill climbing.

Algorithm 1 presents the pseudo-code. The difference of our setup with a canonical hill climbing is that we cannot

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

**Algorithm 1:** Pseudo-code for the BHC adaptive *Optimizer*.

instantly evaluate both neighbors of current configuration, *i.e.* the new configuration after a step on $LB$ and after a step on $C$. We thus make a random choice towards one criterion, and act as a function of resulting computing time, by committing or rollbacking changes made (an optional parameter $\epsilon$ can enforce improvement over previous configuration to be $\epsilon$ times better). We name this algorithm *Blind Hill Climbing* or BHC in the sequel. Methods `OptimizeOnCut()` and `OptimizeOnBalancing()` are for instance picking "bad" nodes having a large contribution to $C$; `OptimizeOnCut()` selects worst nodes on all workers, while `OptimizeOnBalancing()` only on the more unbalanced workers. They then allocate the selected nodes to the partitions containing the most of their neighbors (`OptimizeOnCut()`), or using the best heuristic from [12] (*i.e.* the *weighted deterministic greedy* strategy for `OptimizeOnBalancing()`). Method `getComputeTime()` returns current $\mathbb{E}^G$. Finally, method `buffer()` records all incoming events (vertices, requests) in a message queue, while `flushBuffer()` consumes those buffered events. This encapsulation ensures that no graph modification affects the system while optimization is being performed.

## IV. Performance Evaluation by Simulation

To validate our approach, we evaluate the performance of the BHC module through simulations on real-world network topologies for a panel of typical vertex-oriented graph applications. The worker model is patterned after Section II, and the system processes real graphs that are presented hereafter.

### A. System Setup

We consider four applications, that are representative of different locality/computation trade-offs. Each operation is vertex-oriented, and run on the current graph partitioning:
`FETCH`: get one-hop neighbors, to fetch and display information (*e.g.* on user's wall) [10].
`RECOM`: get a recommendation, for instance to propose new friends to connect to [2]. This implementation uses random walks from the node to propose friends to.
`CENTRAL`: get $\frac{diameter}{4}$-hop neighbors, and compute centrality over this ego network, to identify key locations for data dissemination [4].
`MIX`: execute an equal load of the three previous operation.

The computational cost of the requests `FETCH`, `RECOM`, `CENTRAL` and `MIX` are respectively $c(r) = 1$, 2, 3 and 2. We use different values of $\lambda$ to simulate different network instantiations: one very slow with $\lambda = 1000$, one medium[1] with $\lambda = 100$ and one fast with $\lambda = 10$. The processing throughput $\chi$ of a machine is set to one, and $\epsilon$ to zero (*i.e.* no degradation allowed). Requests on the system are simulated as follows: 100 times per run, $1\%$ of randomly selected nodes are the center of a request (either `FETCH`, `RECOM`, `CENTRAL` or `MIX` accordingly). We simulate various number of workers $k \in \{4, 8, 32\}$ and migrations are up to $10\%$ of workers' partition sizes at time of optimization. `getComputeTime()` is estimated by executing Eq 1 over $\frac{n_t}{100}$ requests (with $n_t$ the size of the graph $G$ at time $t$), randomly picking the start node. Finally, each experiment is reported as the average of 15 independent runs.

We simulate requests over growing graphs, that are real world interaction graphs available online[2]. As the information about the real growth of those graphs is not available (exact order of arrival of their vertices), we stream them by picking vertices in a random order (15 runs and then random seeds for each graph and each application). We run simulations on the following graphs: 4elt ($n = 15,606$, $m = 91,756$), Brightkite ($n = 58,228$, $m = 214,078$), Digg ($n = 30,398$, $m = 87,627$), escorts ($n = 16,730$, $m = 50,632$), Facebook ($n = 63,731$, $m = 1,269,502$), Gnutella ($n = 62,586$, $m = 147,892$), Gowalla ($n = 196,591$, $m = 950,327$), pgp ($n = 10,680$, $m = 24,316$), Slashdot ($n = 79,120$, $m = 515,581$) and Twitter ($n = 465,017$, $m = 834,797$).

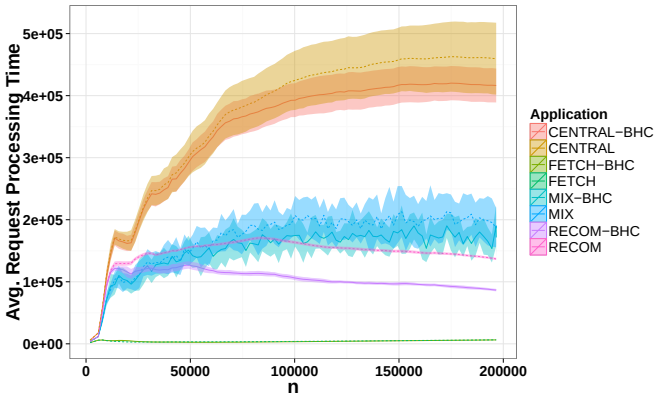
### B. Simulation Results

Figure 2 details a run of our system. In 2(a), curves represent the evolution of the request processing time $\mathbb{E}^G$, while the Gowalla graph is growing over time. Dashed curves represent $\mathbb{E}^G$'s evolution in a system with no optimization (*i.e.* no vertex migration), while plain curves a system implementing the BHC module. The most striking benefits of BHC are witnessed on less local applications such as `CENTRAL` and `RECOM`, *e.g.* with around $50\%$ improvement for `RECOM`. On `FETCH` requests, no noticeable improvement occurs, as the state of the art algorithm from paper [12] achieves a very good static partitioning.

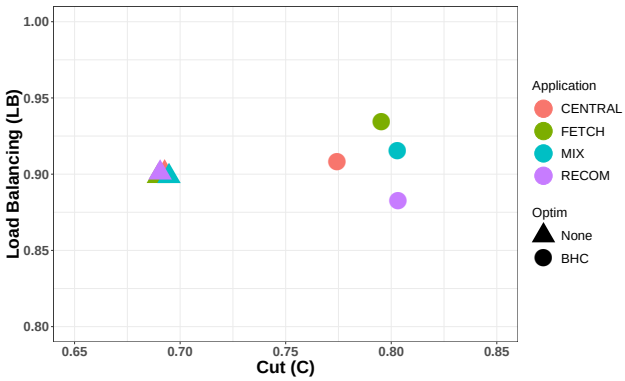
Figure 2(b) provides a more aggregate perspective of the results. It plots the endpoints (graphs fully streamed) with and without BHC optimization, averaged for all 45 different runs (15 for each cost ratio) on each graph, and for $k = 4$. First, one can observe that non-optimized results (triangles) have a worse $C$ and $LB$ than the results using BHC (except for `RECOM`, having a worse $LB$). Moreover, one can notice that all the non-optimized results are application-oblivious:

[1]There is typically a factor 100 between a main memory based operation, and a network operation on a 1Gbps network (please refer to http://norvig.com/21-days.html).

[2]http://konect.uni-koblenz.de/networks/




(a) Runtime of the different applications over the Gowalla graph ($\lambda = 100$, and $k = 4$) along its streaming process, as a function of the number $n$ of streamed vertices (emulating time). Mean and standard deviation computed over 15 independent runs.




(b) Aggregate load balancing (y-axis) and edge cut (x-axis) values produced with and without BHC optimization. Aggregation is made over the 10 datasets, for 15 independent runs. $k = 4$.

Fig. 2. Simulation of a graph processing system, with and without the BHC optimization module: results plotting evolution of processing time and improvement of load balancing and edge cut.

| | | CENTRAL | | | FETCH | | | MIX | | | RECOM | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dataset | l=10 | 100 | 1000 | l=10 | 100 | 1000 | l=10 | 100 | 1000 | l=10 | 100 | 1000 |
| **w=4** | | | | | | | | | | | | | |
| 1 | 4elt | 0.84 | 2.32 | 11.94 | 0.79 | 0.93 | 4.27 | 4.48 | 21.92 | 38.71 | 7.44 | 31.38 | 39.14 |
| 2 | brightkite | 2.53 | 9.53 | 16.25 | 0.56 | 0.72 | 3.19 | 8.02 | 26.86 | 33.74 | 16.01 | 40.77 | 49.66 |
| 3 | digg | 0.16 | 3.58 | 7.94 | 0.22 | 0.21 | 0.97 | 1.15 | 3.43 | 3.58 | 3.09 | 6.16 | 6.91 |
| 4 | escorts | 0.01 | 0.43 | 4.40 | 0.41 | 2.11 | 15.22 | 23.69 | 57.05 | 68.04 | 43.63 | 67.45 | 71.31 |
| 5 | facebook | 2.50 | 10.03 | 12.60 | 0.79 | 2.74 | 18.31 | 8.87 | 14.35 | 17.39 | 14.43 | 38.24 | 47.46 |
| 6 | gnutella | 0.00 | 0.00 | 0.43 | 0.00 | 0.07 | 0.14 | 0.13 | 5.09 | 4.80 | 1.87 | 5.71 | 7.76 |
| 7 | gowalla | -2.99 | 9.25 | 7.83 | 0.38 | 0.53 | 1.95 | 3.64 | 14.81 | 33.32 | 7.13 | 35.51 | 60.55 |
| 8 | pgp | -0.60 | 13.55 | 8.74 | 0.07 | 0.85 | 5.94 | 7.59 | 32.43 | 28.63 | 17.50 | 37.85 | 39.81 |
| 9 | slashdot | 0.03 | 4.46 | 1.04 | 0.31 | 0.70 | 4.51 | 4.75 | 6.62 | 5.80 | 8.15 | 23.90 | 28.29 |
| 10 | twitter | 0.33 | 0.86 | 1.23 | 0.34 | 0.34 | 1.85 | 4.30 | 18.45 | 34.51 | 14.29 | 38.65 | 49.77 |
| **w=8** | | | | | | | | | | | | | |
| 11 | 4elt | 1.82 | 4.32 | 16.69 | 1.41 | 2.04 | 3.48 | 6.77 | 29.25 | 34.24 | 12.38 | 37.79 | 33.89 |
| 12 | brightkite | 5.94 | 8.64 | 15.79 | 0.34 | 1.01 | 7.85 | 10.75 | 21.24 | 30.81 | 22.79 | 43.44 | 47.61 |
| 13 | digg | -0.31 | 5.20 | 9.00 | 0.20 | 0.37 | 2.56 | 3.29 | 2.77 | 5.21 | 4.32 | 4.83 | 4.83 |
| 14 | escorts | 0.39 | 0.86 | 7.84 | 0.45 | 4.27 | 21.37 | 34.25 | 57.55 | 63.09 | 49.97 | 61.73 | 64.00 |
| 15 | facebook | 9.78 | 11.90 | 5.26 | 1.29 | 5.49 | 27.27 | 13.53 | 19.14 | 21.05 | 20.97 | 39.43 | 42.24 |
| 16 | gnutella | 0.01 | 0.22 | 0.52 | 0.01 | 0.07 | 1.36 | 0.70 | 1.28 | 0.12 | 2.38 | 5.14 | 3.83 |
| 17 | gowalla | 6.20 | 15.22 | 18.49 | 0.12 | 0.41 | 3.52 | 2.24 | 24.29 | 24.71 | 10.99 | 38.59 | 51.69 |
| 18 | pgp | 0.31 | 24.85 | 3.85 | 0.30 | 2.34 | 9.58 | 16.02 | 26.80 | 33.72 | 26.43 | 38.63 | 42.91 |
| 19 | slashdot | 0.98 | 2.19 | -4.12 | 0.40 | 1.14 | 3.28 | 4.79 | 10.60 | 2.18 | 11.08 | 20.19 | 23.06 |
| 20 | twitter | 0.45 | 1.93 | 4.98 | 0.42 | 0.76 | 6.37 | 5.16 | 26.63 | 32.54 | 21.02 | 42.73 | 46.42 |
| **w=32** | | | | | | | | | | | | | |
| 21 | 4elt | 2.50 | 10.03 | 25.73 | 1.71 | 3.48 | 16.61 | 13.04 | 22.56 | 18.67 | 22.39 | 35.39 | 32.42 |
| 22 | brightkite | 2.37 | 7.41 | 5.75 | 0.48 | 1.67 | 14.56 | 18.71 | 24.17 | 23.65 | 27.95 | 33.97 | 35.34 |
| 23 | digg | -0.63 | 8.12 | -1.76 | 0.13 | 0.60 | 5.46 | 1.59 | 5.08 | 8.02 | 6.45 | 7.13 | 7.46 |
| 24 | escorts | 0.68 | 3.81 | 16.56 | 0.93 | 11.41 | 38.83 | 35.81 | 48.19 | 46.24 | 46.47 | 46.86 | 47.59 |
| 25 | facebook | 6.31 | 9.90 | 6.86 | 2.94 | 13.14 | 33.35 | 12.17 | 11.77 | 11.88 | 28.13 | 33.11 | 33.65 |
| 26 | gnutella | -0.12 | 0.65 | 3.25 | 0.03 | 0.14 | 1.58 | 2.56 | 3.19 | 5.67 | 2.31 | 3.31 | 3.87 |
| 27 | gowalla | 4.34 | 9.41 | 9.30 | 0.28 | 1.41 | 4.38 | 8.89 | 10.23 | 13.05 | 20.85 | 36.89 | 40.06 |
| 28 | pgp | -7.45 | 10.89 | 17.67 | 0.94 | 5.57 | 24.45 | 29.17 | 36.76 | 38.70 | 37.58 | 39.59 | 40.43 |
| 29 | slashdot | 8.97 | 0.26 | 5.70 | 0.29 | 0.08 | 5.64 | 1.65 | 1.83 | 11.96 | 14.42 | 17.48 | 16.81 |
| 30 | twitter | 0.77 | 4.87 | 23.11 | 0.83 | 3.49 | 17.20 | 30.34 | 43.31 | 46.47 | 42.29 | 51.37 | 50.83 |
| **Average** | | | | | | | | | | | | | |
| 31 | | 1.53 | 6.48 | 8.76 | 0.57 | 2.26 | 10.16 | 10.6 | 20.92 | 24.68 | 18.82 | 32.1 | 35.65 |

Fig. 3. Relative request processing time improvement (in %) of BHC for a variety of datasets, applications and system parameters. Improvements of processing time over 10% are in highlighted green, while degradations are in red.

all applications end up grouped at the same position. This is because the original BSP-hybrid systems does not target partitioning improvement based on the application feedback. On the contrary, results with BHC are clearly separated: optimizations are driven by $\mathbb{E}^G$, that lead the processing system in the best configuration for the current application. Intuitively, `FETCH` (green) favors $LB$ (locality prevails), while `RECOM` favors $C$ (more information fetching expected to occur). `CENTRAL` sits in the middle.

Figure finally 3 presents another aggregate results for all graphs: the relative runtime improvement (that is $\frac{\mathbb{E}^G - \mathbb{E}^G_{BHC}}{\mathbb{E}^G}$, where $\mathbb{E}^G_{BHC}$ and $\mathbb{E}^G$ are respectively the optimized and non-optimized average request processing times). As observed before, since the original vertex assignment heuristic [12] optimizes cut, local applications like `FETCH` do not benefit much from BHC. Yet, all other less local applications see their

runtime improved. Similarly, improvements are greater when the network latency is higher, with up to of $71\%$ of improvement (RECOM on the escorts network, with $\lambda = 1000$). Note that degradations (red values) are possible in this simulation, as the reported numbers are averages of independent runs, where result variability may occur due to the random nature of vertex stream order for instance. On average (as seen line 31 of Figure 3), and more notably for less local applications (CENTRAL, RECOM and thus also MIX) in contexts of medium to slow networks, significant improvement results from the use of BHC. Results range from around $10\%$, up to $35\%$ improvement.

As a conclusion of this evaluation Section, it appears that there is a clear gain in optimizing the processing system, based on the application runtime feedback. This is achieved by selecting and migrating vertices based on the direction ($LB$ or $C$) privileged by the applications running on the system, at runtime.

## V. Related work

Most of the distributed graph processing systems follow the Bulk Synchronous Parallel (BSP) model introduced by Pregel [8], allowing iterative graph computations on distributed partitions of the graph. This includes systems as Giraph [1], Mizan [7], Powergraph [6], GPS [11] and Kineograph [3]. Only recently, Giraph++ [14] has shown significant improvement over this BSP model, by keeping global synchronization supersteps and by additionally allowing in-worker computations between two of those supersteps. This is achieved by considering the vertex partitions on workers as sub-graphs. Good partitioning in Giraph++ can benefit from vertex locality. We precisely address this complex problem in this paper, to propose an additional software module for further performance improvement of BSP-hybrid systems.

Some previous works took specific steps for performance improvements, that can be classified in two categories. *(i)* Related works also stated that the input graph characteristics are impacting the system processing time. Powergraph [6] and GPS [11] improve the handling of graph with power law degree distributions, by distributing computations for highest degree vertices over the whole set of workers. *(ii)* Vaquero et al. [16] considered the the migration of vertices at runtime, to adapt to graph updates and preserve satisfying performances for the processing system. Some works target extreme graph scales of billions of edges [9]. Yet those adaptations are application agnostic or non-reactive, while our proposal is driven by average processing time of the applications. Those two classes of works only apply to BSP-systems; this paper addresses the adaptive graph improvement in BSP-hybrid systems.

## VI. Conclusion

Leveraging the recent advance proposed by Giraph++ over bulk synchronous parallel systems, this paper has studied the impact of the partitioning on processing performances of this new model. An optimization problem has been proposed to address the complex load balancing versus edge cut trade-off. A solver for this problem, available as a generic software module for BSP-hybrid systems, has been detailed and simulated. It improves by an average of up to $35\%$ the standard BSP-hybrid performances, by adapting the graph to the actual request workload on the system. Having exposed this partitioning trade-off and the benefits of considering it, future-work includes the deployment and testing of the optimization module in a tailored BSP-hybrid production system.

## References


[1] Apache Giraph, 2014.

[2] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. In *VLDB*, 2010.

[3] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EUROSYS*, 2012.

[4] Wei Gao and Guohong Cao. User-centric data dissemination in disruption tolerant networks. In *INFOCOM*, 2011.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[6] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[7] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EUROSYS*, 2013.

[8] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[9] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. In *ICDE*, 2017.

[10] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikolaos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. *IEEE/ACM Trans. Netw.*, 20(4):1162–1175, August 2012.

[11] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *SSDBM*, 2013.

[12] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, 2012.

[13] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 2011.

[14] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *VLDB*, 2013.

[15] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[16] L.M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *ICDCS*, 2014.