Adaptive Partitioning for Large-Scale Dynamic Graphs

Vaquero, Luis M.; Cuadrado, Félix; Martella, Claudio; Logothetis, Dionysios

ISSN 2043-0167

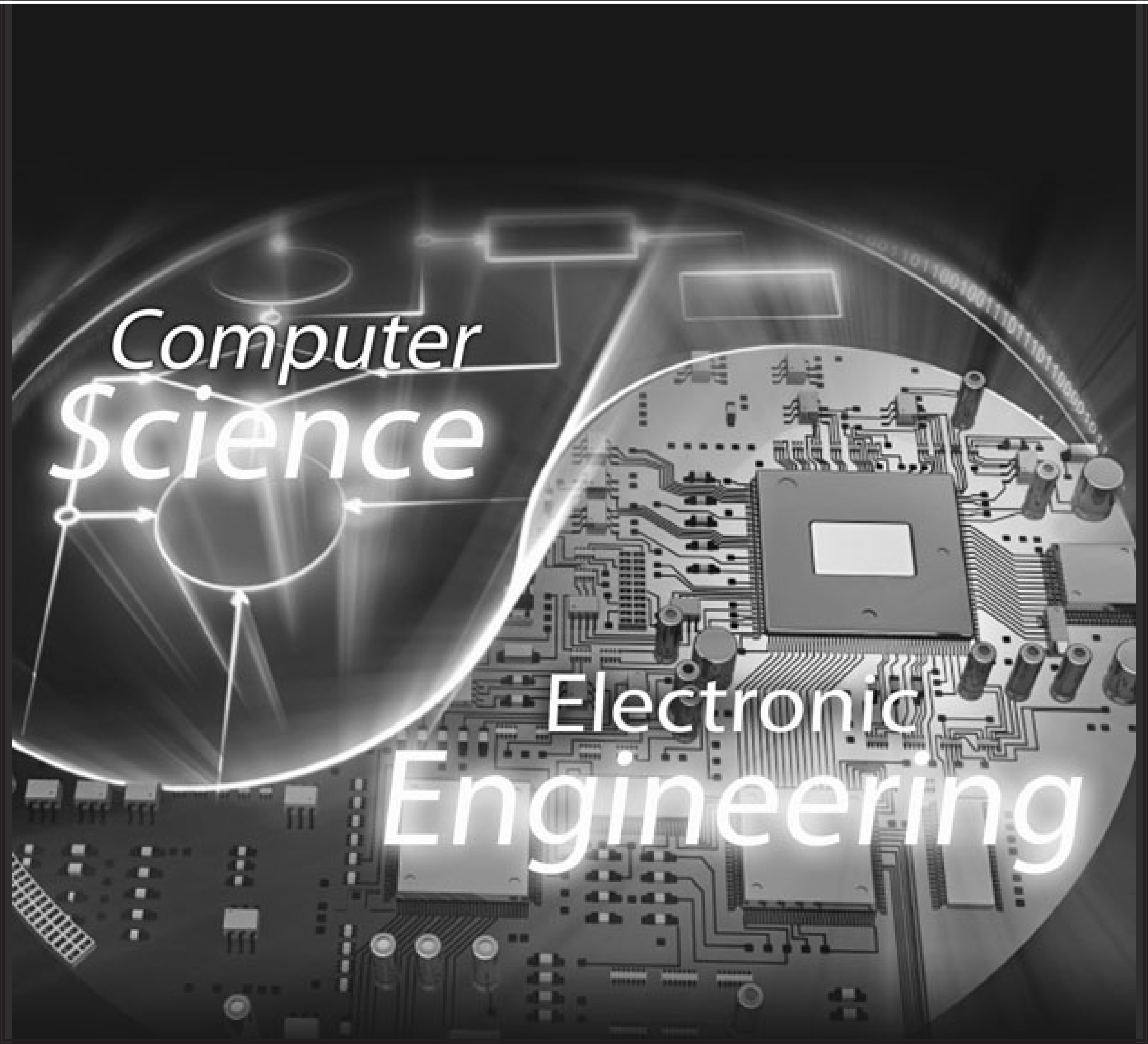
Adaptive Partitioning for Large-Scale Dynamic Graphs

Luis M. Vaquero, Félix Cuadrado, Claudio Martella and Dionysios Logothetis

Queen Mary University of London

EECSRR-13-02 June 2013

School of Electronic Engineering and Computer Science



Adaptive Partitioning for Large-Scale Dynamic Graphs

Luis M. Vaquero
Queen Mary University of London
luis.vaquero@ieee.org

Félix Cuadrado
Queen Mary University of London
felix@eecs.qmul.ac.uk

Dionysios Logothetis
Telefonica Research
dl@tid.es

Claudio Martella
VU University Amsterdam
c.martella@vu.nl

ABSTRACT

Mining large graphs is critical for several real-world systems such as social networks. Such graphs typically consist of hundreds of millions of vertices and edges and are highly dynamic, with their structure continuously evolving over time. The current data deluge is making this problem even harder: bigger graphs must be processed even faster than before. Appropriate graph partitioning is crucial to scale to large graphs and reduce processing time. However, partitioning large dynamic graphs is challenging: current techniques either do not scale well as they require global graph knowledge or do not handle changes in the graph.

In this work, we introduce a general purpose graph partitioning algorithm that adapts to dynamic structural changes in massive graphs. The algorithm works by iteratively adapting the partitions using only local information. We show how the application of our algorithm to three real-world scenarios can speed up the execution of graph analyses by over 50%, while adapting the partitioning to a high volume of changes to the graph.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Graph Theory, Graph Algorithms; H.3.4 [**Systems and Software**]: Distributed Systems

Keywords

Large scale optimisation, dynamic graphs, adaptive graph partitioning, distributed algorithm

1. INTRODUCTION

Graphs underlie many real-world systems, such as social and communication networks and even the stock market. Several critical analytical tasks in these systems, like ranking and recommending online content or discovering groups of correlated stocks, depend on the ability to mine huge graphs that may consist of millions of vertices and billions of edges. The importance of managing graphs of that scale is evidenced by the emergence of numerous distributed graph storage and graph processing systems in recent years [21, 9, 27, 1, 7, 33, 20, 16, 36].

Graph partitioning is crucial in attacking large-scale graph management problems. It impacts both the performance of graph mining algorithms and the scalability of the underlying storage and graph processing systems. For instance, popular algorithms for content ranking converge faster if initialised with a good graph partitioning [38]. Further, distributed graph processing systems benefit from intelligent graph partitioning that reduces communication and achieves load balancing across the system, eventually achieving scalability and efficiency [21]. This makes graph partitioning an indispensable step in the graph management lifecycle.

However, in accordance with trends observed in data analytics today, graph management faces a new challenge: data are becoming increasingly time-sensitive. Graphs are highly dynamic by nature and several real-world analytical applications now require near real-time response to graph changes. For instance, the Twitter graph may receive thousands of updates per second at peak rate [3] that can potentially indicate new trending topics. Topic recommendation analytics must reflect these changes within minutes, otherwise they become irrelevant. Similarly, telcos must detect frauds by mining the Call Detail Record graph in real-time [41].

As graph partitioning is an integral phase of these analytical pipelines, it is critical that it can keep up with the graph update rate. However, existing graph partitioning methods overlook this dynamic nature of real graphs. Graph dynamism may render the existing partitioning of a graph obsolete within minutes. Current approaches either allow the partitioning to diverge from the optimal as the graph changes, hurting processing time, or require re-partitioning of the graph every time, a costly process that effectively also increases processing time.

We believe that to enable graph mining applications in real-world environments, such as online social networks, we need scalable partitioning algorithms that take the dynamic nature of the graphs into consideration. Essentially, this requires the ability to quickly and efficiently adapt the partitioning as the graph changes.

1.1 Challenges

However, partitioning large dynamic graphs is a challenging task. First, ensuring good application performance, our ultimate goal, requires graph partitions that minimise communication and achieve load balancing. Minimising communication allows more parallelisation and improves system scalability, while load balancing can have a significant impact on overall processing time [21, 35]. However, these are often conflicting requirements. For instance, random partitioning has good load-balancing properties, but results in high communication overheads.

Second, the graph partitioning algorithm must scale itself to large graphs. Most existing partitioning methods do not scale because they require global information about the graph and involve complex coordination mechanisms. Additionally, they may incur high processing times, rendering any potential speed up of the graph mining task useless. Instead, a graph partitioning algorithm must afford a scalable and lightweight implementation. Simplicity as a design choice is, therefore, necessary for a real large-scale implementation, a requirement recognised in [35] as well.

Third, real-world graphs evolve rapidly. To maintain good application performance as the graph changes, the existing partitioning must be updated fast even in the face of high update rates. Otherwise, the time spent adapting the partitioning may render the potential savings useless. Adaptive partitioning must incur minimum costs, making efficient use of available resources, such as the network.

1.2 Our Approach

To address the challenges in processing massive dynamic graphs, we propose a scalable graph partitioning algorithm that reacts to graph topology changes with minimum overhead. Our algorithm is based on decentralised, iterative vertex migration. Starting from any initial partitioning, the algorithm migrates vertices between partitions trying to minimise the number of cut edges, while at the same time keeping partitions balanced, until convergence. The migration heuristic is based on local per-vertex information and requires no global coordination, affording a scalable distributed implementation. Furthermore, our approach naturally supports dynamic graph changes. Updates in the graph topology trigger the vertex migration process, adapting the partitioning to the new topology.

This paper makes the following contributions: 1) The first heuristic capable of adapting massive graph partitions to dynamic changes in the topology of the graph relying only on local information which boots the scaling potential and adaptation rate. 2) The extension of the core heuristic and its integration into a real processing system based on Pregel [21]. We show that implementing our algorithm inside such a system can speed up processing through partitioning optimisation. 3) An extensive evaluation of our technique through both simulations and system deployments, using synthetic and real datasets.

The rest of the paper is organised as follows. In Section 2, we describe our algorithm in more detail. Section 3 describes some relevant pitfalls that need to be overcome when realising the algorithm into a real system. The algorithm is then tested at scale in a series of lab experiments and real world use cases in Section 4. Section 5 contextualises our approach by presenting the related work. We present the main conclusions and discussion in Section 6.

2. ADAPTIVE ITERATIVE PARTITIONING

In this section, we present an iterative algorithm that produces balanced partitions (avoids concentrating too many vertices in a few partitions). Note that the iterative nature of the algorithm guarantees adaptation to changes in the graph topology. We have designed the algorithm relying on local information only, so that it I can be efficiently computed, and enables a scalable implementation. We start by presenting the algorithm at a logical level. In Section 3, we show how it can be naturally implemented in a distributed scalable fashion. Before we present the algorithm, let us start with a few definitions.

Definition *(Graph Partitioning).* Given a graph $G = (V, E)$, let P^t be the set of partitions on V at time t and $P^t(i)$ the individual partition i, with $|P^t| = k$. These partitions will be such that $\bigcup_{i,t}^{k} P^t(i) = V$ and $P^t(i) \cap P^t(j) = \emptyset$ for any $i \neq j$. The edge cut set $E_c \subseteq E$ is the set of edges which endpoint vertices belong to different partitions.

At a high level, the algorithm starts with an initial partitioning: the graph is loaded on the different partitions. The most commonly used strategy in large scale graph processing systems is *hash partitioning.* Given a hashing function $H(v)$, a vertex is assigned to partition $P^0(i)$ if $H(v)\ mod\ k = i$. This strategy is effective as it is lightweight, it does not require a global lookup table, and, depending on the characteristics of the vertex I'dDs, it can scatter the vertices uniformly across the partitions. Unfortunately, it introduces many cut edges. In addition, this method does not guarantee adaptation to changes in the topology of the graph, since its initial partitioning is never updated.

2.1 Greedy Vertex Migration

Additional processing is required for the partitions to keep track with the changes in the structure of the graph. On every iteration after the initial partitioning, each vertex will make a decision to either remain in the current partition, or to migrate to a different one. Migration decisions are only based on local information available to the vertex, where the goal is to "get neighbours together" in order to minimise the number of cut edges $|E_c|$.

For this individual decision we evaluated multiple heuristics based on local information [35, 28]. We chose a simple greedy heuristic that had the strongest performance and lowest computational cost. The heuristic works as follows. At each iteration, a vertex will decide to migrate to the partition where the highest number of its neighbouring vertices are. With this premise, the candidate partitions for each vertex are those where the highest number of its neighbours are located. Formally, for a vertex v, the list of candidate partitions is derived as follows $cand(v,t) = \{P^t(i) \in P^t, \exists$ w, w $\in (P^t(i) \cap \Gamma(v,t))\}$, where $\Gamma(v,t)$ is the set of v plus its neighbours at iteration t[1]. Since migrating a vertex potentially introduces an overhead, the heuristic will preferentially choose to stay in the current partition if it is one of the candidates.

The heuristic is compatible with the approach described in Section 1.2: relying on local information. A vertex v is only aware of its own neighbours and it uses this information

[1]Note that we measure time in number of iterations, decoupling the heuristic from implementation considerations. The actual time taken by an iteration to complete will depend on the system and the specific load of the system at that iteration.

to choose its destination partition. This restriction fosters a lightweight heuristic, avoiding the need of coordination mechanisms and greatly boosting scalability .

2.2 Maintaining Balanced Partitions

The greedy nature of the presented heuristic will naturally cause higher concentration of vertices in some partitions. We refer to this phenomenon as node densification. As our goal is to obtain a balanced partitioning, a capacity limit must be introduced for every partition. The approach does not guarantee an optimal balanced k-way partitioning, but it limits the unbalance of vertices distribution across the partitions.

Definition *(Partition Capacity).* Let $C(i)$ be the capacity constraint on each partition. At all times t, for each partition i, $|P^t(i)| \leq C(i)$.

In order to control node densification, vertices need to be aware of the maximum partition capacities capacity $C(i)$. The remaining capacity of each partition i at iteration t is $C^t(i) = C(i) - |P^t(i)|$.

The local and independent nature of migration decisions make these capacity limits difficult to enforce. At iteration t the decision of a vertex to migrate can only be based on the capacities $C^t(i)$ computed at the beginning of the iteration. These capacities will not be updated during the iteration, which implies that without further restrictions all vertices will be allowed to migrate to the same destination, potentially exceeding the capacity limit.

The only way to ensure that the capacity of each partition will not be surpassed, with only the information available to the vertex, is to work on a worst case basis. We split the available capacity for each partition equally and we use these splits as quotas for the other partitions. Hence, the maximum number of vertices that can migrate from partition i to partition j over an iteration t is defined as: $Q^t(i,j) = \frac{C^t(j)}{|P^t|-1}$; $j \neq i$. See Section 3 for details on how this was implemented.

This strategy to manage partition capacities introduces minimum coordination overhead. Vertices base their decision on the location of their neighbours, and the partition-level current capacity information, which must be available locally to every node. Propagating capacity information is scalable, as it is proportional to the total number of partitions k.

2.3 Ensuring Convergence

The independent nature of the migration decisions endangers convergence of the heuristic. Local symmetries in the graph may cause pairs (or higher cardinality sets) of neighbour vertices independently decide to "chase each other" in the same iteration, as the best option is to join its neighbour.

We have addressed these issues by introducing a random factor to the migration decisions. At each iteration, each vertex will decide whether to migrate with probability s, $0 < s < 1$. A value of $s = 0$ causes no migration whatsoever, while $s = 1$ allows the vertices to migrate on every iteration they attempt to. Intermediate values in the range address the chasing effect, but lower values also affect the overall convergence time.

We explored the effect of different values of s with an extensive set of experiments on different graphs, assessing convergence time and node densification. Details about the

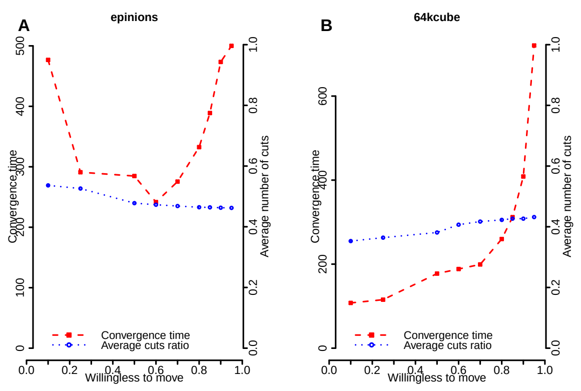


Figure 1: Effect of s into Convergence and Number of Cuts (normalised to the total number of edges in the graph). Average of 10 experiments performed over two graphs: 64kcube (A) and Epinions(B) from Table 1, partitioning over 9 nodes.

selected graphs are provided in Section 4.1. We assumed full convergence when the number of vertex migrations was zero for more than 30 consecutive iterations, although the number of migrations decreases exponentially with the number of iterations. Figure 1 shows the effect of s on convergence time and normalised number of cuts for two different graphs. In both cases, there was no statistical difference in the number of cuts achieved by the heuristic, regardless of the value of s. Similar results were obtained for the remaining graphs used in our study, shown in Table 1.

However, s can have a significant impact on convergence time. Low values os s limit the number of migrations executed per iteration, potentially increasing the time required for convergence. On the other side, high values fail to fully compensate the neighbour chasing effect, introducing wasted migrations per iteration that delay convergence and increase computation time. This is particularly evident in Figure 1 (B). From our experience, a constant intermediate value ($s = 0.5$) will have adequate performance over a variety of graphs: the reduced message overhead makes processing differences (due to variations in s) negligible. This is specially true in the context of long running (continuous) processing systems.

3. LARGE SCALE IMPLEMENTATION

To validate our approach, we implement a distributed version of our algorithm and integrate it into a large-scale graph processing system inspired by Pregel. Here, we describe the details of our implementation.

Our iterative graph algorithm runs as a background application fully compatible with Pregel's computational model. Figure 2 shows the layered architecture of the implemented system. The implementation differs from Pregel in two aspects: 1) once the graph has been loaded into memory, computation is run continuously; 2) vertices/edges can be injected/removed from the graph during the computation from a stream. These two aspects motivate the implementation of an adaptive graph partitioning algorithm that runs in the background of the system, while the user applications process the graph.

Having a well-defined heuristic is not enough to come up with a usable implementation at large scale. Here, we

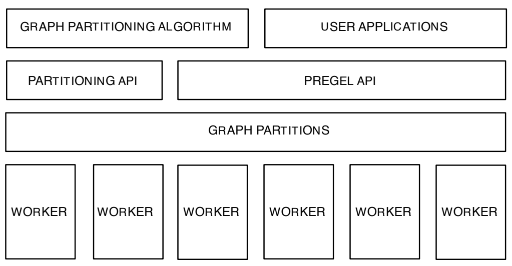


Figure 2: Layered architecture of the implemented system. Both the graph partitioning algorithm and the user applications make use of the Pregel API to process the graph. The partitioning algorithm uses an extension of the API to migrate the vertices and access capacities.

present a short roadmap to overcome the main implementation pitfalls we faced while implementing the algorithm for large-scale graph processing systems.

Deferred Vertex Migration.
At any iteration t, neighbours do not know where any of their (potentially) migrated neighbouring vertices will be at iteration $t+1$, when messages will actually be delivered. Migrating a vertex instantly after its decision would require one of the following adaptations to avoid losing messages(see Figure 3 (top)): either forwarding the incoming messages to the new destination of the vertex, or update the messages in the outgoing queues of the other workers with the updated destination. However, these solutions require additional synchronisation and coordination capabilities that would greatly limit the scalability of the algorithm.

We solve this coordination problem by forcing vertices to wait for one iteration before they migrate. At the end of iteration t, at which the vertex requested the migration, the worker notifies the other workers about the upcoming migration, so that they will have been notified at the start of the following iteration $t+1$, and the new messages produced during iteration $t+1$ can be sent directly to the new destination (see Figure 3 (bottom)). This way the computation is not directly affected by the migrations.

Worker to Worker Capacity Messaging.
Each worker must send a message notifying the $|C^i(t)|$ of its partitions to the other workers. Scalable remote messaging has to follow the one iteration delay enforced by Pregel. Therefore, workers must actually send information about their capacity at iteration $t+1$, ensuring partial freshness. The predicted capacity will be $C^{t+1}(i) = C^t(i) - V_o^{t+1}(i) + V_i^{t+1}(i)$, where $V_i^t(i) \subset V$ are the vertices migrating to i in $t+1$, and $V_o^t(i) \subset V$ are the vertices migrating from i to partition j in $t+1$. Both $V_o^{t+1}(i)$ and $V_i^{t+1}(i)$ are known by the worker at iteration $t+1$: the migration delay for incoming vertices ensures that the workers will be aware of this value, and outgoing vertices are notified locally, overcoming any communication restrictions.

4. EVALUATION

In this section, we present the evaluation of our algorithm on different datasets. First, we evaluate the quality of the partitioning with respect to minimising the number of cut edges. In particular, we establish the quality of the partitioning along with its dependence on the strategy used for the initial partitioning, the type and size of the graphs. Second, we evaluate the impact of running our adaptive graph partitioning algorithm on a real large-scale graph processing system. Here, we assess the impact of the graph partitioning on the actual computation time of the processing system.

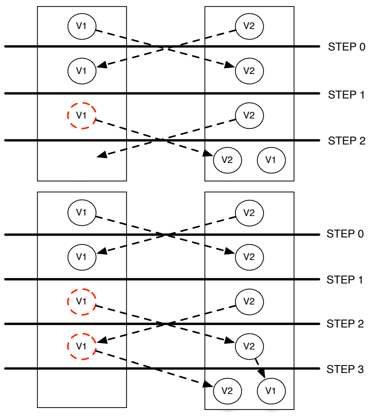


Figure 3: Deferred Vertex Migration to Ensure Message Delivery. *Top:* Failed message delivery due to incorrect synchronisation. *Bottom:* Correct delivery. The dashed-red circle indicates when the vertex is in a "migrating" state waiting for one iteration (step) before actually migrating.

4.1 Datasets

We have selected a representative collection of graphs (see Table 1), including synthetic graphs and real world graphs, of multiple sizes (up to 300 million edges) and edge distributions: homogeneous, like in finite-element meshes (FEM) and power-law degree distribution.

Regarding the synthetic graphs, the synthetic meshes have a 3d regular cubic structure, modelling the electric connections between heart cells [37]. Power law synthetic graphs have been generated with networkX, using its power law degree distribution and approximate average clustering [13]; the intended average degree is $D = log(|V|)$, with rewiring probability $p = 0.1$. These are static graphs, to mimic dynamic changes we employed a "forest fire" model and created a synthetic extension [18]. This extension was injected following different time distributions, here we present the worst case: simultaneous creation of all the new vertices (see Section 4.3).

In addition to these graphs, we also used two real world sources of dynamic data: 1) We processed tweets from Twitter's streaming API access level in real time; 2) We processed one-month data of anonymised calls in a mobile European operator, consisting of 21 million vertices, 132 million reciprocated social ties, with a mean geodesic distance of 9.4, an average degree of 10.1 network neighbours, and a giant component containing 99.1% of all vertices was analysed. We fed these data chronologically, building a dynamic graph of call interactions, and calculated the maximal clique at any time.

4.2 Quality of the Partitioning

Obtaining a high quality partitioning for large scale graphs is one of the main goals of the adaptive algorithm. We adopted the *cut ratio*, i.e. the number of cut edges normalised to the total number of edges in the graph, as gold standard for assessing the quality of the partitioning. All the experiments shown below are the mean of $n = 10$ repetitions. Errors are reported in the form of estimated error in the mean.

4.2.1 Sensitivity to Initial Partitioning

As a first step, we wanted to rule out variations on performance arising from using a different initial partitioning. After initialising the partitions with one of four different initial strategies, we ran our adaptive iterative algorithm until no vertex migration was requested for 30 iterations. The initial partitioning strategies we tested were: 1) *Hash Partitioning (HSH)*: the destination partition is computed for each vertex as described in Section 2; 2) *Pseudorandom Partitioning (RND)*: vertices were assigned to partitions through a pseudorandom generator, still ensuring balanced partitions; 3) *Deterministic Greedy (DGR)*: stream-based "linear deterministic greedy" as presented in [35]; 4) *Minimum Number of Neighbours (MNN)*: applies the same stream-based approach to the "minimum number of neighbours" heuristic presented in [28].

The results show that the iterative partitioning significantly improves the cut ratio (by 0.2 to 0.4) for FEM (Figure 4A) and power law (Figure 4B) graphs for three out of four initial partition strategies. It only slightly improves the cut ratio when starting from the DGR heuristic since the heuristics have a similar nature, and the results are already close to the benchmark provided by partitioning using METIS [14], a state-of-the-art centralised graph partitioning algorithm. In other words, our heuristic seems to improve widely used alternatives towards this lower limit. It is worth noting that DGR depends on full graph knowledge (destinations of already allocated vertices), which poses limits to its scalability and its applicability to real deployments.

Video 1[2] shows how partitioning evolves in real time in a 2d slice of a 3d cube of a 1000000 mesh graph, where every vertex is physically surrounded by its neighbours. As can be observed, the initial hash partitioning across 9 partitions (represented with a different colour each) is improved by increasing the number of neighbours placed together.

4.2.2 Dependence on the Type of Graph

Ruling side effects due to differences in initial partitioning out is as important as proving our heuristic behaves the

[2]https://dl.dropbox.com/u/5262310/reducedCuts.avi

Table 1: Summary of the datasets employed in this work.

Name	$\|V\|$	$\|E\|$	Type	Source
1e4	10000	27900	FEM	synth
64kcube	64000	187200	FEM	synth
1e6	1000000	2970000	FEM	synth
1e8	10^8	$2.97 * 10^8$	FEM	synth
3elt	4720	13722	FEM	[34]
4elt	15606	45878	FEM	[34]
plc1000	1000	9879	pwlaw	synth
plc10000	10000	129774	pwlaw	synth
plc50000	50000	1249061	pwlaw	synth
wikivote	7115	103689	pwlaw	[19]
epinion	75879	508837	pwlaw	[30]
uk-2007-05-u	1000000	41247159	pwlaw	[2]

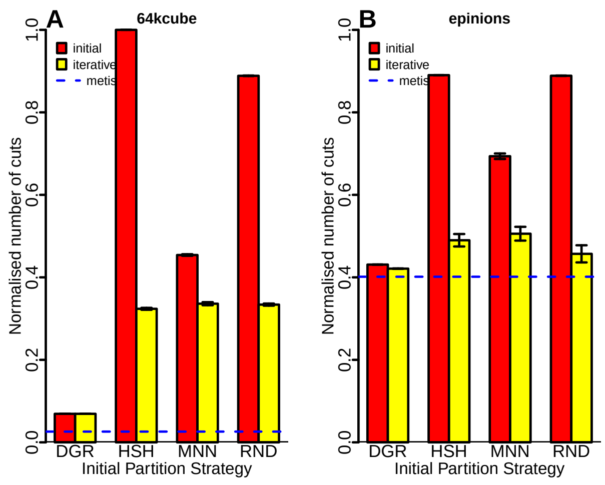


Figure 4: Normalised number of cut edges after applying the iterative algorithm, starting from four initial partitioning strategies. 9 partitions, with maximum capacity equal to 110% of the balanced load. The horizontal dashed line represents the results obtained using METIS.

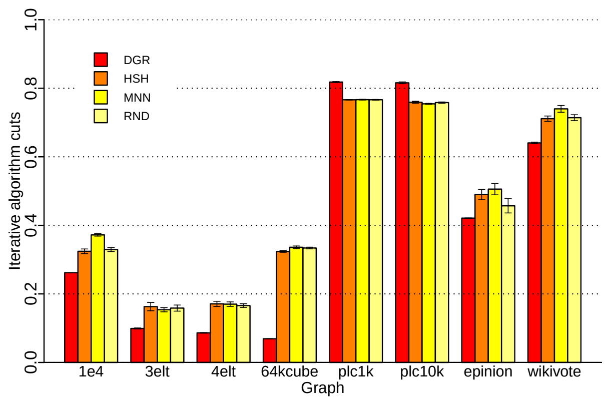


Figure 5: Average cuts for each graph after running the iterative heuristic over four different initial partitioning strategies.

same for a variety of graphs. Figure 5 shows the final cut ratio values for a few graphs in our dataset. The analysed FEMs generally get better results, while synthetic power law graphs with high average degree perform noticeably worse. The difficulties encountered also by DGR and METIS, show that these graphs are very difficult to partition.

These results point out that the proposed heuristic is compatible with different initial partitioning strategies and can improve the partitioning quality of a wide range of graphs.

4.2.3 Scalability

To test the scalability of our algorithm we generated six homogeneous FEMs with a number of vertices ranging from 1000 to 300000 vertices. Figure 6 shows the performance of the algorithm with respect to both cut ratio and convergence time (measured in number of iterations). Convergence time increases with the size of the graph, with a $O(log (N))$ growth rate, therefore pointing to feasible application in large-scale scenarios. The quality of the obtained partitioning slightly improves with the increase in the problem size, which can be derived from the constant ratio between

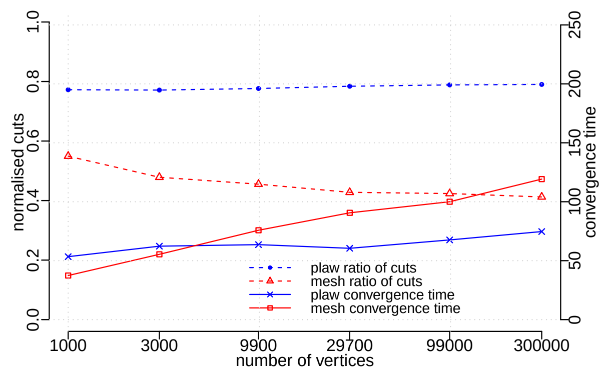


Figure 6: Evolution of cut ratio and convergence time for a family of meshes (red) and power law graphs (blue) ranging from 10000 vertices to 300000. 9 partitions, with $s = 0.5$

average vertex degree and number of partitions, while the total graph size increases.

We repeated the experiment with a family of power law graphs of equal number of vertices, generated with the same parameters for power law graphs described above. In this case the average convergence time grows at a much slower rate than in the case of meshes and the cut ratio remains almost constant, slightly degrading with the increase in graph size. These results are in line with what was reported in [35]. These authors worked on optimising the initial partitioning of a graph and did not present any adaptation mechanism for massive graphs.

4.3 Real Word Use Cases

We validated our algorithm in the wild, integrating it into a Pregel-inspired system, and testing its effect over three real-word use cases. We assessed three aspects of the algorithm in these experiments: the ability to partition large scale graphs in realistic scenarios, the ability to cope with dynamic changes in the graph structure, and the performance impact of improving the quality of the partitioning. We believe that the diversity in the workloads of each application helps to support the general validity of our heuristic.

Adaptation in Biomedical Simulations.

This scenario assesses the impact of the partitioning algorithm in large scale biomedical simulations, where FEMs are common place. The main goal was to observe the behaviour of the adaptive algorithm implementation under extreme scaling conditions and prove that our heuristic rapidly reduces the number of edge cuts and, therefore, the communication cost without introducing a huge overhead due to vertex migration.

The input graph was a 100 million vertex/300 million edges FEM representing the cellular structure of a section of the heart. Each vertex computes more than 32 differential equations on one hundred variables representing the way cardiac cells are excited to produce a synchronised heart contraction and blood pumping [37]. Using a static partitioning(without the adaptive algorithm), simulation time is dominated by the exchange of messages (more than 80% of the time), even though CPU time is not negligible (more than 17%).

Figure 7 represents the evolution of quality of the partitioning (cuts), overhead (migrations) and performance (time per iteration) for two scenarios: 1) initial optimisation of the poorly performing hash partitioning that is common practice in most large scale graph processing systems; 2) absoprtion of a huge increase in the number of new vertices and edges, see Figure 7(b).

When the graph is initially loaded into memory with plain hash partitioning, the number of cuts is very high. Our heuristic dramatically reduces the number of cuts at the expense of moving vertices around for a better placement, see Figure 7(a). The time it takes to compute a whole iteration (represented by the green dotted triangles and right hand side Y axis) has been normalised to the one obtained with static hash partitioning, which is a direct comparison to most other large scale processing systems. This time increases drastically (21 times) due to the massive amount of vertices being migrated (16 million) and quickly starts to decay exponentially. With our partitioning an iteration is computed two times faster (compare first and last green triangle in the series).

After this initial optimisation, we modified the graph structure dynamically and tested the ability of the algorithm to adapt to the changes. To this end, we injected a forest fire expansion of 10 % of the size of the graph (10 new million vertices and 30 new million edges). This triggers a moderate increase in the number of cuts (30 million new edges), movements (1.5 million vertices being moved) and time per iteration (4.6 times bigger than with plain hash partitioning). The peak is rapidly absorbed by our heuristic getting back to similar levels of cuts and performance, but having 10 % more vertices/edges.

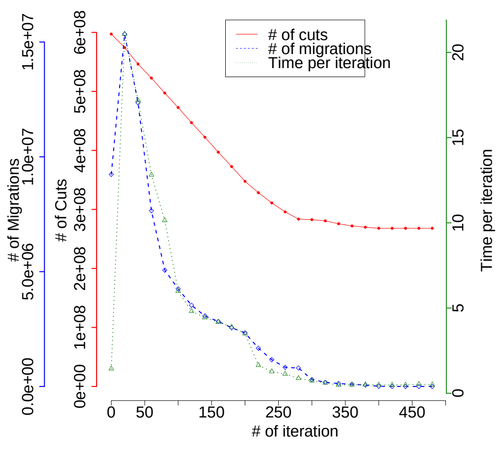
This test proves that our algorithm is highly scalable and responds nicely to extreme conditions and peaks in load. Its performance is twice as fast as the one obtained through hash partitioning, due to a reduction of $\simeq 50\%$ in the number of cut edges. This is achieved at the expense of a moderate overhead, due to migrations, that exponentially decreases until it vanishes after convergence, leaving just the performance gain.

Adaptation in Online Social Network Analysis.

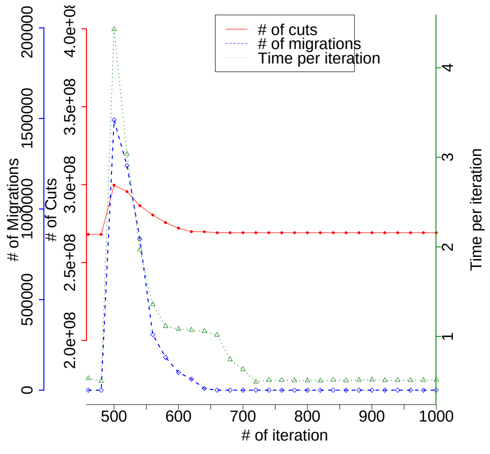
The second use case assesses the adaptation capabilities of the algorithm when applied to a continuously evolving dynamic graph. Making use of local information to migrate vertices should result in a reduced number of accesses to the network and, therefore, in less variability in computing each iteration, given properly dimensioned resources, e.g. lack of swapping.

We captured tweets in real time from Twitter Streaming API, and built a graph where edges are given by mentions of users. Over this power law graph, we continuously estimated the influence of certain users by using the TunkRank algorithm [39]. In this test, execution time is bound by the number of messages sent over the network at any point in time (over 80% of the iteration time)

We ran the experiment simultaneously in two separate clusters: One cluster used the adaptive algorithm, while the other used static hash partitioning instead. In Figure 8 we can observe the average results from processing tweets collected in the London area over a whole day (Friday, 5th Oct 2012), after running continuously for 4 days. The red line shows the rate at which tweets are received and processed by the system, while the blue and orange lines show aver-



(a) Hash partition re-arrangement



(b) Absorption of a huge peak in load

Figure 7: Biomedical use case with a 100 million node/300 million edge graph which is expanded using a forest fire adding 10 million vertices and 30 million edges. Performance is normalised to the value obtained with a static hash partitioning. Results were obtained in a cluster of 63 Worker Blades (64GB RAM, 10GbE and 12 Cores)

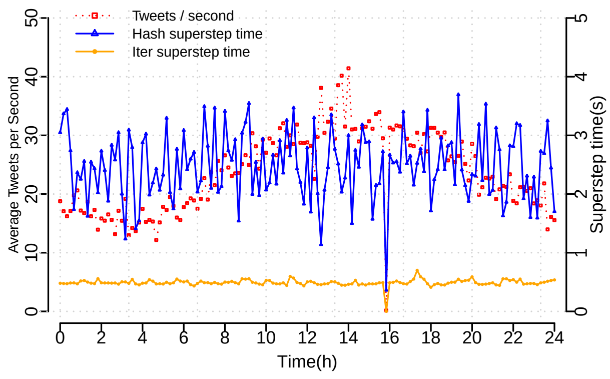


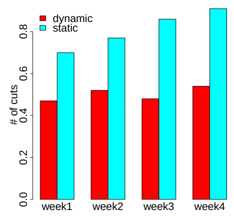
Figure 8: Throughput and performance obtained by processing the incoming stream of tweets from London. Each point represents the average of 10 min of streaming data. The sudden drop in throughput and superstep time is due to a failure in one of the workers that led to the triggering of recovery mechanism.

age execution times per iteration, with and without adaptation, respectively. As it can be clearly observed, the average execution time is significantly improved when applying the adaptive algorithm, (mean of 0.5 secs instead of 2.5 secs, including the added overhead). Importantly, the optimisation of the partitioning with local information only significantly lessens variability in execution times, by reducing the impact of network communications (more neighbours are local).

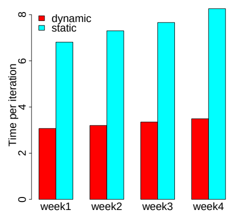
Adaptation in Mobile Network Communications.

The final use case computes the maximum cliques on a dynamic graph, generated from one month of mobile telephone calls. The graph changed by adding nodes and vertices corresponding to new calls as they arrived to the system, and removing them if they were inactive for more than one week. The maximum clique was obtained as follows. In the first iteration, each vertex sends its lists of neighbours to all its neighbours. On the next iteration, given a vertex i and each of its neighbours j, i creates j lists containing the neighbours of j that are also neighbours with i. Lists containing the same elements reveal a clique. As these lists can get large, this algorithm produces heavy messaging overhead for large graphs, especially if these are dense, and not negligible CPU costs, although not as much as the biomedical' use case above.

In contrast with the previous two scenarios, this application requires freezing the graph topology until a result is obtained, therefore requiring to buffer all the graph changes until the computation finishes. This characteristic makes the scenario more challenging than the previous one, as every iteration will trigger the adaptation to a batch set of changes to the graph. Call data was streamed into the system with a speed up factor of 15, to increase the amount of buffered changes per iteration, further testing the adaptive algorithm performance. The dataset yielded weekly addition/deletion rates of 8 and 4%, respectively, which is higher than those reported previous studies due to the shorter period of analysis [10]. This turnover is low enough to advise to keep most of the unchanged graph in memory, rather than re-loading from scratch.



(a) Number of cuts



(b) Time per iteration

Figure 9: Evolution of the number of cuts normalised to the total number of edges (Left) and average iteration (step) time (Right) during the 4 weeks of available data. The experiments were performed in a cluster of 5 96GB RAM, 10 GbE, 12 core workers.

We run the clique finding application in two separate clusters, with and without the adaptive algorithm. Figure 9 shows the weekly results in both cases. It can be seen that the adaptive partitioning maintained a stable number of cuts, resulting in consistently reduced time per iteration (less than 50% time per iteration). Moreover, weekly trends show the the non-adaptive option experiences farther performance degradation over time due to the higher cut ratio.

5. RELATED WORK

To the best of our knowledge, there is no single approach that satisfies all the challenges we outlined in Section 1.1. Our algorithm adapts partitioning to large scale graph changes while 1) minimising the number of cut edges to reduce communication overhead, 2) producing balanced partitioning with capacity capping for load balancing, and 3) relying only on decentralised coordination based on a local vertex-centric view. The heuristic is generic and can be applied to a variety of workloads and application scenarios.

The idea of partitioning the graph to minimise network communication is not new and it has inspired several techniques to co-locate neighbouring vertices in the same host [5, 26, 15, 23, 6, 4, 11]. These approaches try to exploit the locality present in the graphs, whether due to the vertices being geographically close in social networks, close molecules establishing chemical bonds, or web pages related by topic or domain, by placing neighbouring vertices in the same partition. The work presented in [35], where the authors authors evaluate a set of simple heuristics based on the idea of exploiting locality, and apply them on a single streaming pass over a graph, with competitive results and low computation cost. The authors show the benefits of this approach in real systems. However, the streaming technique adopted requires global information, which can become an scalability bottleneck whereas our approach relies only on local information.

Community detection algorithms concentrate on finding groups vertices that are more densely connected internally than with the rest of the network. Unfortunately, as these algorithms are designed to extract the underlying community structure, they do not focus on finding balanced partitions in the number of vertices or edges. Moreover, given these algorithms are very sensitive to the graph structure, small changes to the graph can lead to very different partitions and too many vertex migrations [17].

The parallel version of METIS [14], ParMETIS [25], leverages parallel processing for partitioning the graph, through multilevel k-way partitioning, adaptive re-partitioning, and parallel multi-constrained partitioning schemes but requires a global view of the graph that greatly reduces its scalability [32]. Other techniques have been explored that study graph properties projected onto a small subset of vertices [18, 11]. These may be effective in some particular contexts, but they are not broadly applicable. [40] uses label propagation [29], to feed a linear programming solver that iteratively optimises the partitioning. However, the approach makes use of global information, which complicates its application as a scalable solution for adapting to graph dynamism.

The need to continuously adapt to the evolution of the graph, without the overhead of re-loading an updated snapshot of the graph or re-partitioning from scratch, has been recently reported in practical [31, 22, 12] and more theoretical [24] studies. However, the previously mentioned techniques cannot handle structural graph changes, either degrading partition quality, or fully triggering the partition process. Recently, a different approach attempts to dynamically adapt the partitioning of the graph to the bandwidth characteristics of the underlying computer network to maximise throughput [8]. Their decisions are not local and their scalability level was tested up to 100 GB graph, while our 100 million vertex graph occupied 3 TB once loaded in RAM.

6. CONCLUSIONS

We have presented the first algorithm for adaptive partitioning of large dynamic graphs. We consciously chose a lightweight heuristic that relies on local per-vertex information only, allowing the algorithm to scale to large graphs but also accommodate high graph update rates. Our heuristic provides a good tradeoff between minimising the number of cut edges across partitions and balanced partitioning. We have shown that the algorithm can adapt the partitioning to the dynamism of the underlying graph at very large scales under extreme conditions (instantaneous addition of an additional 10% of a massive graph). Indeed, in some scenarios its performance is competitive with centralised best-of-breed partitioning algorithms like METIS.

By integrating the algorithm as a component of our parallel and distributed graph processing system, we have demonstrated its effect on performance for real-world graph analysis. The algorithm copes with continuous changes to the graph structure, maintaining the quality of partitioning over time. Despite the small additional computational cost introduced by the algorithm it diminishes effectively (more than halves in our experiments) the total computational time on a series of problems. This is achieved by diminishing the internode communication between the cluster nodes, as a result of the minimisation of the cut edges. The initial overhead incurred by moving vertices to other partitions is rapidly overcome by the gain obtained from better placement.

For our future work, we would like to explore two main directions. First, as many graph algorithms like PageRank have a complexity that is proportional to the number of edges, we would like to extend our heuristic to create partitions that are balanced on the number of edges. This should have an impact on the load balancing of the system. Second, we would like to take into account runtime statistics, such as the hot spots (i.e. partitions that are more active than others), in order to achieve a better load balancing of the system.

7. REFERENCES

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

[2] Laboratory for Web Algorithms. Universita de Milano, http://law.di.unimi.it/datasets.php.

[3] Tweets about steve jobs spike but don't break twitter peak record, http://searchengineland.com/tweets-about-steve-jobs-spike-but-dont-break-twitter-record-96048.

[4] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.

[5] S. N. Bhatt and F. T. Leighton. A framework for solving vlsi graph layout problems. Technical report, Cambridge, MA, USA, 1983.

[6] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10).

[7] B. Chao, H. Wang, and Y. Li. The trinity graph engine, March 2012.

[8] R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, 2012.

[9] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, 2012.

[10] C. Cortes, D. Pregibon, and C. Volinsky. Computational methods for dynamic graphs. *Journal of Computational and Graphical Statistics*, 2003.

[11] A. Das Sarma. *Algorithms for large graphs*. PhD thesis, 2010.

[12] O. Gaci. A dynamic graph to fold amino acid interaction networks. In *IEEE CEC*, july 2010.

[13] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Phys. Rev. E*, 65, Jan 2002.

[14] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.

[16] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal. A high-level framework for distributed processing of large-scale graphs. In *Distributed Computing and Networking*, volume 6522 of *LNCS*. 2011.

[17] H. Kwak, Y. Choi, Y.-H. Eom, H. Jeong, and S. Moon. Mining communities in networks: a solution for consistency and its evaluation. In *IMC*, 2009.

[18] J. Leskovec, S. Dumais, and E. Horvitz. Web projections: learning from contextual subgraphs of the web. In *WWW*, 2007.

[19] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, 2010.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODC*, 2009.

[22] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, and J.-P. Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 328(5980), 2010.

[23] M. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23), 2006.

[24] V. Nicosia, J. Tang, M. Musolesi, C. Mascolo, G. Russo, and V. Latora. Components in time-varying graphs. *AIP Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22, 2012.

[25] ParMETIS. Parmetis - parallel graph partitioning and fill-reducing matrix ordering, October 2012.

[26] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN Europe*, 1996.

[27] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[28] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, 2012.

[29] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76, 2007.

[30] M. Richardson, R. Agrawal, and P. Domingos. *Trust Management for the Semantic Web*. 2003.

[31] C. Rostoker, A. Wagner, and H. Hoos. A parallel workflow for real-time correlation and clustering of high-frequency stock market data. In *IPDPS*, 2007.

[32] S. Salihoglu and J. Widom. Gps: A graph processing system. Technical report, Santa Clara, CA, USA, 2012.

[33] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CLOUDCOM*, 2010.

[34] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *J. Global Optimization*, 29, 2004.

[35] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, 2012.

[36] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC*, 2010.

[37] K. Ten Tusscher, D. Noble, P. Noble, and A. Panfilov. A model for human ventricular tissue. *Am J Physiol Heart Circ Physiol*, 2004.

[38] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *ICDM*, Oct. 2006.

[39] D. Tunkelang. A twitter analog to pagerank, http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.

[40] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, WSDM '13, 2013.

[41] S. Weigert, M. Hiltunen, and C. Fetzer. Mining large distributed log data in near real time. In *Workshop on Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, Cascais, Portugal, Oct. 2011.