Adaptive Partitioning of Large-Scale Dynamic Graphs

Luis M. Vaquero,
HP Labs, Bristol
Email: luis.vaquero@hp.com

Felix Cuadrado
Queen Mary University of London
Email: felix.cuadrado@qmul.ac.uk

Dionysios Logothetis
Telefonica Research Labs
Email: dl@tid.es

Claudio Martella
VU University Amsterdam
Email: claudio.martella@vu.nl

***Abstract*—In the last years, large-scale graph processing has gained increasing attention, with most recent systems placing particular emphasis on latency. One possible technique to improve runtime performance in a distributed graph processing system is to reduce network communication. The most notable way to achieve this goal is to partition the graph by minimizing the number of edges that connect vertices assigned to different machines, while keeping the load balanced. However, real-world graphs are highly dynamic, with vertices and edges being constantly added and removed. Carefully updating the partitioning of the graph to reflect these changes is necessary to avoid the introduction of an extensive number of cut edges, which would gradually worsen computation performance.**

In this paper we show that performance degradation in dynamic graph processing systems can be avoided by adapting continuously the graph partitions as the graph changes. We present a novel highly scalable adaptive partitioning strategy, and show a number of refinements that make it work under the constraints of a large-scale distributed system. The partitioning strategy is based on iterative vertex migrations, relying only on local information. We have implemented the technique in a graph processing system, and we show through three real-world scenarios how adapting graph partitioning reduces execution time by over 50% when compared to commonly used hash-partitioning.

I. Introduction

The importance of large-scale graph analytics has given rise to a variety of distributed graph processing architectures [17], [1], [5], [16], [30]. Graph partitioning is crucial to the scalability and efficiency of these systems. The distribution of the underlying graph across machines directly impacts communication overhead and load balancing, and may indirectly affect aspects such as memory usage by the graph management layer. Although graph partitioning is a well studied subject [12], [20], the massive scale of graphs available today and the emergence of such graph processing systems have renewed the interest in this problem [29], [34], [6], [13].

Most graph analytics system so far have been designed for batch processing. However, recently we have been observing a shift toward dynamic graph programming paradigms and processing architectures. This reflects the fact that in many real-world scenarios the underlying graph data are naturally dynamic and analytics is a continuous process, often requiring real-time responses to graph changes. For instance, the Twitter graph may receive thousands of updates per second [3], and systems like [7] are designed to accomodate such graph mutations during processing and keep the analysis results up-to-date with low latency. These new models diverge from the batch paradigm, as graphs are loaded and partitioned across the machines, and kept in memory for a long period of time.

In this paper we argue that retrofitting partitioning algorithms designed for static graphs in such systems impedes their operation, by hurting performance. Figure 1 shows the evolution of partitioning over time in a graph created from mobile CDR (Call Detail Records) (details in Section IV). Partitioning quality is measured as the fraction of cut edges. Starting from an initial partitioning, as the graph changes over time, static approaches like standard hash-partitioning (HSH) and deterministic-greedy (DGT)[29], a state-of-the-art partitioning algorithm, do not adapt to the changes, allowing the partitioning quality to gradually degrade.

Maintaining a high quality partitioning can have high impact on continuous analytical applications that require low latency [7]. Currently the standard practice is to schedule full graph repartitions. Although this rectifies the problem seemingly, repartitioning may actually be an expensive task that takes hours in large-scale graphs [31], practically prohibiting frequent adaptation.

We believe that these new graph management systems call for partitioning techniques that embrace the dynamic nature of graphs. Toward this, we propose a partitioning algorithm that scales to large graphs and produces partitions with good locality and, importantly, efficiently adapts the partitioning upon graph changes. Our algorithm is based on decentralised, iterative vertex migration. Starting from any initial partitioning,

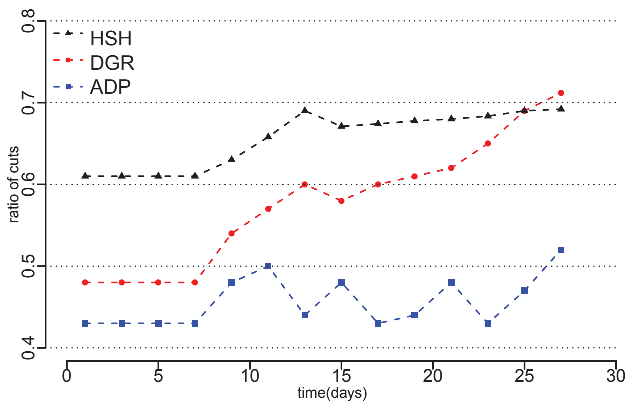

Figure 1. Evolution of the ratio of cuts over time on a dynamic graph generated by processing CDR data over a sliding window. We compare three algorithms: *hash partitioning* (HSH), *deterministic greedy* (DTG) and our *adaptive repartitioning* (ADP) approach. The initial point for each algorithm shows the cut ratio after processing 24 hours of CDR data.

实际情况：数据是动态变化的，分析是连续的
问题：现有的数据分片策略导致分片的质量下降；如果重新分片则会导致大量的开销。
本论文：提出动态适应性的partition策略。--顶点迁移

the algorithm migrates vertices between partitions based on a heuristic that minimises the number of cut edges, while at the same time keeping partitions balanced. The migration heuristic uses only local per-vertex information and requires no global coordination, affording a scalable distributed implementation.

Furthermore, we go beyond just an algorithmic solution and investigate the challenges of integrating such an algorithm into a real graph processing system. For instance, any practical adaptive partitioning scheme must ensure that vertex migrations do not cause inconsistencies during operation and that routing information is efficiently maintained. In particular, we describe the design of a Pregel-like system extended for continuous graph analytics that integrates our approach and addresses these challenges. We show that by efficiently maintaining an optimal partitioning, our system can sustain a processing latency that is 5 times lower than a non-adaptive approach.

In this paper, we make the following contributions:

1) We propose a scalable graph partitioning heuristic based on iterative vertex migration that produces partitions with good locality and can efficiently adapt the partitioning to dynamic graph changes.
2) We describe the design of a Pregel-like graph processing system for continuous graph analytics, and study the challenges of integrating an adaptive partitioning algorithm into a real system. We show how we extend the core heuristic to address these challenges and make this integration possible.
3) We present an extensive evaluation of our approach. Using synthetic and real graphs, we show that our algorithm produces partitionings with good locality, improving application performance. Further, we show that our algorithm can adaptively maintain a good partitioning upon graph changes with minimum cost. Finally, through the deployment of a real Twitter-feed analysis application we show that the integration of our adaptive approach inside our system can sustain a processing latency up to 5 times lower than a static partitioning approach.

The rest of the paper is organised as follows. In Section II, we describe our heuristic in more detail. Section III describes the design decisions that we took for implementing the heuristic on a real system with scalability requirements. The heuristic is then tested at scale in a series of lab experiments and real-world use cases in Section IV. Section V presents related work on partitioning dynamic graphs. We present the main conclusions and discussion in Section VI.

II. Adaptive Iterative Partitioning

In this section, we present the core of our approach, an algorithm that iteratively applies a greedy vertex migration heuristic to find a partitioning with good locality. Before we present the algorithm, we proceed with a few definitions.

A. Problem statement

Definition A dynamic graph $G(t) = (V(t), E(t))$ is a graph whose vertices V and edges E can change over time t, either with addition or removal of elements. Let $P(t)$ be the set of partitions on V at time t and $P^i(t)$ the individual partition i, with $|P^i| = k$. These partitions will be such that $\bigcup_{i,t}^{k} P^i(t) = V$ and $P^i(t) \cap P^j(t) = \emptyset$ 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.

A distributed graph processing system splits the partitions between compute nodes. Every vertex belonging to the graph has an assigned partition. At time $t = 0$, the graph is loaded with an initial partitioning. New vertices appearing at $t > 0$ also have to be assigned to a partition according to a strategy. 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 $P^i(0)$ if $H(v)$ *mod* $k = i$. Hash partitioning is popular because it is a highly scalable, lightweight technique that balances the vertices across partitions, as long as the values generated by $H()$ function are uniformly distributed.

The ideal repartitioning strategy for dynamic graphs should work on the following assumptions: 1) changes are not predictable and partitions need to be updated to prevent performance degradation; 2) partitioning should be updated on graph changes as fast as possible in order to prevent performance degradation; 3) partition optimisation decisions should be highly scalable; requiring shared global state brings additional overhead from synchronising multiple worker machines.

B. Greedy vertex migration

We have defined a heuristic for dynamically adapting graph partitions that considers the assumptions above. Our heuristic is based on label propagation [23], which has also been adopted for optimising the initial load in memory of static graphs [34]. Vertices iteratively adopt the label of the majority of their neighbours (no global state needed) until no new labels are assigned (convergence is reached).

On every iteration t[1] after the initial partitioning, each vertex will make a decision to either remain in the current partition, or to migrate to a different one. 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: $cand(v,t) = \{P^i(t) \in P(t), \exists$ w, w $\in (P^i(t) \cap \Gamma(v,t))\}$, where $\Gamma(v,t)$ is the set of v plus its neighbours at iteration t. 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.

At the end of the iteration, all vertices who decided to migrate will change to their desired partitions. Video 1[2] shows how the heuristic evolves partitioning over time in a 2d slice of a 3d cube of a 10^6 vertices mesh graph, where every vertex is physically surrounded by its neighbours. As time goes, the initial hash partitioning across 9 partitions (represented with a different colour each) is improved by increasing locality.

[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.

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

The heuristic relies on local information, as each vertex v chooses its destination based only on the location of its neighbours. Dynamism comes natively in this iterative approach. New vertices are initially assigned a partition according to a strategy (we opted for the *de facto* standard, hash modulo) and the heuristic will automatically attempt to move them closer to their neighbours (see Figure 4 and related text below for more details on this). Note that the partitioning algorithm is executed in the background, inside the processing engine and in a fully transparent fashion with respect to the applications running in the system.

C. *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, common to general label propagation algorithms [23], as node densification. As our goal is to obtain a balanced partitioning, we set a capacity limit for every partition.

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

In order to control node densification, vertices need to be aware of the maximum partition capacities C^i. The remaining capacity of each partition i at iteration t is $C^i(t) = C^i - |P^i(t)|$. These values change every iteration, forcing to relax our local information constraint.

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 $C^i(t)$ 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.

We ensure these limits will not be surpassed by independent decisions by working 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: $Q^{i,j}(t) = \frac{C^j(t)}{|P(t)|-1}; j \neq i$. See Section III for system implementation details

This strategy 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.

D. *Ensuring convergence*

The independent nature of the migration decisions may delay convergence. 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.

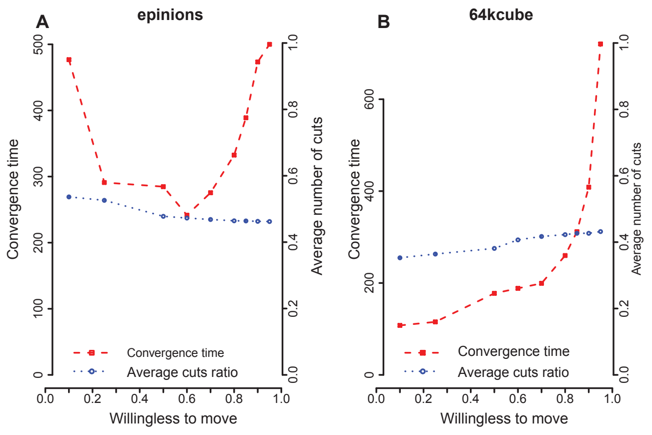

Figure 2. Effect of s on Convergence (time per iteration) and Number of Cuts (normalised to the total number of edges in the graph). Average of 10 experiments performed over two graphs: 64kcube (A) and Epinions(B) from Table I, partitioning over 9 nodes.

We have addressed these issues by introducing a random factor to the migration decisions. At each iteration, each vertex will consider migration with probability s, $0 < s < 1$. A value of $s = 0$ causes no migration whatsoever, while $s = 1$ allows vertices to migrate on every iteration they attempt to.

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 selected graphs are provided in Section IV-A. We assumed full convergence when the number of vertex migrations was zero for more than 30 consecutive iterations. Figure 2 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 I.

However, s can have a significant impact on convergence time. Low values of 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 2 (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.

III. System Design

In this section, we present our large-scale dynamic graph processing system. We provide an overview of the computational model, the distributed system architecture, and finally detail how we have integrated the iterative adaptation heuristic.

A. *Computation model*

The main design goals of the system are dynamic graph adaptation, failure tolerance, and optional snapshotting of intermediate results. The system implements a continuous dynamic

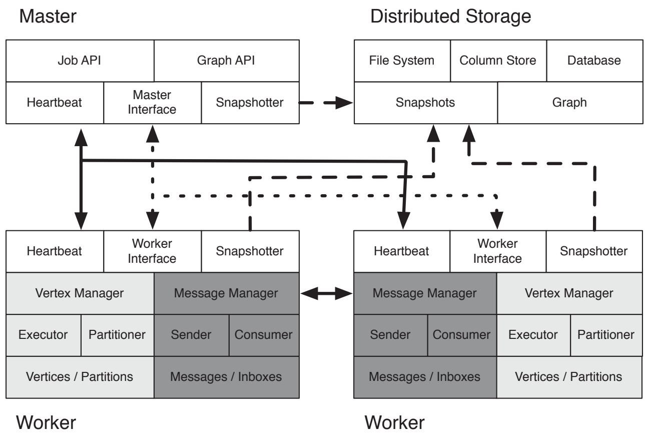

Figure 3. System overview, following a Master-Worker architecture. The grayed sets of boxes are executed in parallel in multi-core machines.

graph processing model, following the Bulk Synchronous Parallel [35] computational model and Pregel's "think like a vertex" philosophy. A job is composed of a sequence of one or more user-defined functions to be executed at each vertex in the graph. Messages propagate to vertex neighbors at the end of each superstep. The job executes continuously on the graph (in order to update the results according to graph changes), with execution concluding only with an explicit request of the user, e.g. for maintenance purposes.

When the system starts, the graph is loaded into the system with an initial partitioning (modulo hash). The system provides an API for modifying the topology of the graph at any time (adding and removing vertices and edges from the graph). Topology change requests are added to a change queue, and are processed at the end of every job iteration (therefore after all the functions of a job iteration have completed, in n supersteps). The system exports this API as an external interface, allowing other applications to modify the graph in a streaming fashion. Changes to the graph topology are applied at the end of each job iteration.

At the start of every superstep, the adaptive migration heuristic runs over the graph, potentially triggering decisions to adapt the graph to the last changes to the graph.

Some algorithms are insensitive to changes in the topology of the graph while the computation is running. For instance PageRank computation on a changing graph will be inaccurate, but the algorithm achieves what we refer to as asymptotic convergence, meaning that the error is progressively reduced (although never zero). Other algorithms (e.g. single source shortest path) are very sensitive to changes in the topology of the graph. For this second type of algorithms, our system buffers changes while the job iteration is running so that they do not affect the structure of the graph.

B. System implementation

The main system elements are the Master and the Workers, as shown in Figure 3. Master and workers communicate synchronously (RMI) to enforce the global synchronisation barrier. Similarly to Pregel, our system implements an abstract Vertex class that hides all the complexity from the user, with the system orchestrating the vertex-level computations in parallel across the workers. An execution controller creates a number of threads, depending on the number of CPUs available.

Workers keep input and output message queues for inter-worker vertex communications, sending messages through a loosely coupled asynchronous delivery method (we used RabbitMQ and ZeroMQ as interchangeable message handlers). Workers group messages sharing the same source and destination workers, in order to improve transmission efficiency.

The system stores computation snapshots (for failure-tolerance or for keeping the intermediate results of the running function) on a distributed-column store cluster (Cassandra nodes with replicated configuration), keeping balance between writing speed and consistency. Frequent snapshotting and high write-throughput are especially important for dynamic graphs, since intermediate analysis results must be kept for the external applications to show the output and its evolution.

Workers exchange two types of messages. Application (vertex to vertex) messages, containing data related to the computation, and system messages, to support information exchange (e.g. notifying current capacity to other workers). While system messages routing is straightforward, dynamic vertex migration makes routing of application messages more complicated. A Vertex Locator in each of the workers is responsible of finding the current location of a vertex.

Each worker has a partitioner that allocates new vertices to one of the workers. We employ hash partitioning for this initial decision due to its flexibility. These new vertices will be later migrated by the heuristic to maximise locality. A more complex initial placement strategy would involve more coordination and potentially delay the next computing iteration. The partitioner contains a migration manager, which attends the migration requests from local vertices, and allows them to occur unless there is no capacity quota left for the desired destination. Buffers are in charge of dampening new requests to add/delete graph elements. Queues for vertex or edge deletion/addition can be prioritised. A new vertex can initially be placed in any of these queues.

C. Vertex migration support

In this subsection we provide the main insights derived from our experience implementing the system.

Deferred vertex migration At any iteration t, vertices make independent migration decisions, and potentially send messages to be processed by their neighbours. At t, a vertex does not know the destination of neighbour vertices at $t + 1$. Migrating a vertex at the very next iteration after its decision would require one of the following strategies to avoid losing messages (see Figure 4 (top)): either forwarding the incoming messages to the new destination of the vertex, or updating 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 challenge the scalability of the heuristic.

Instead, we solved this coordination problem with no additional overhead: we force vertices to wait for one iteration before they migrate. The vertex requests the migration at

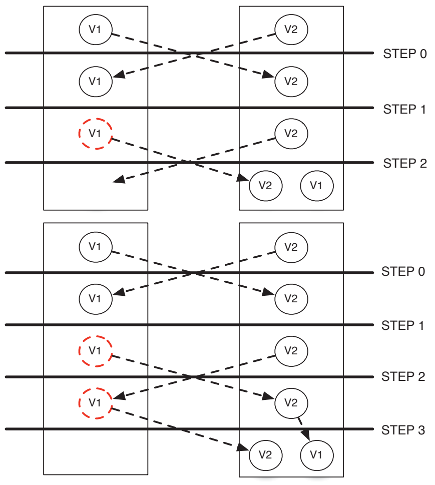

Figure 4. Deferred Vertex Migration to Ensure Message Delivery. *Top:* Failed message delivery due to incorrect synchronisation. *Bottom:* Correct delivery. The dashed-red circle indicates when the vertex is in a "migrating" state waiting for one iteration (step) before actually migrating.

iteration t, and at the end of the migration the worker sends a message to the other workers about the upcoming migration. At the start of the following iteration $t+1$ all the workers have been notified, and the new messages produced during iteration $t+1$ can be sent directly to the new destination (see Figure 4 (bottom)). This way the computation is not directly affected by the migrations.

Worker to worker capacity messaging The heuristic requires the system to maintain an extra element of global information: each worker must notify the $|C^i(t)|$ of its partitions to the other workers. For scalability purposes, these messages are sent asynchronously, so that they will be received and processed at the start of the next iteration. Workers send information about their predicted future capacity at iteration $t+1$, ensuring partial freshness. The predicted capacity will be $C^i(t+1) = C^i(t) - V^i_{out}(t+1) + V^i_{in}(t+1)$, where $V^i_{in}(t) \subset V$ are the vertices migrating to i in $t+1$, and $V^i_{out}(t) \subset V$ are the vertices migrating from i to other partitions in $t+1$. $V^i_{out}(t+1)$ is known by the worker as it is based on local decisions. $V^i_{in}(t+1)$ is also known by the worker at iteration $t+1$ as deferred vertex migration ensures that the workers will be aware of this value.

IV. Evaluation

In this section, we evaluate the ability of our technique to efficiently maintain good quality partitions in the face of graph changes. We compare our results with state-of-the-art algorithms. Further, we evaluate our approach by deploying real-world applications on a Pregel-like continuous graph processing system that integrates our partitioning algorithm, and measuring the impact on application performance.

A. Datasets

For our evaluation we use a diverse collection of synthetic and real-world graphs with varying sizes of up to 300 million edges and different edge distributions: homogeneous finite-element meshes (FEM) and power-law degree distribution. Table I summarizes the dataset. The synthetic mesh graphs have a 3d regular cubic structure, modelling the electric connections between heart cells [32]. The power law synthetic graphs have been generated with networkX [9], using its power law degree distribution and approximate average clustering [10]; the intended average degree is $D = log(|V|)$, with rewiring probability $p = 0.1$.

We mimic dynamic changes to the synthetic graphs by adding nodes and vertices using the well-known "forest fire" model [14], updating the graph with these additions in a single step.

Table I. Summary of the evaluation datasets.

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	[28]
4elt	15606	45878	FEM	[28]
plc1000	1000	9879	pwlaw	synth
plc10000	10000	129774	pwlaw	synth
plc50000	50000	1249061	pwlaw	synth
wikivote	7115	103689	pwlaw	[15]
epinion	75879	508837	pwlaw	[24]
livejournal	4847571	68993773	pwlaw	[2]

In addition to these graphs, we use three real-world sources of dynamic data:

1) We processed tweets from Twitter's streaming API in real-time for a week, generating nodes from users and edges from user mentions in tweets.
2) We processed one-month of anonymised Call Detail Records from a European mobile operator. Processing these data in chronological order, results in a dynamic graph of call interactions, consisting of 21 million vertices and 132 million edges.
3) We run a biomedical simulation of cell apoptosis during a cardiac infarction, modelling the human heart as a FEM. Each cell computed over 70 differential equations per step in a graph of 100M vertices, occuping 3TB in RAM.

We ran all the experiments from this section on a datacenter with 10Gb ethernet. The machines for the experiments had 8-12 cores each, and 64-96GB of RAM. Depending on the size of the graph, and the complexity of each algorithm, we involved a proportionate amount of computing nodes, aiming at 90% memory occupation by the graph partition at the set capacity. We tried to report our results in a hardware-agnostic manner (in number of iterations, and relative values) to ease comparison with other systems.

B. Comparison to State-of-the-art

As we showed in Figure 1, static partitioning approaches allow the partitioning quality to degrade as the graph changes. In order to compare them with our heuristic, we would need to repartition the graph from scratch each time the graph changes. Hash partitioning, which is the most common practice, does not change the assigned partition regardless of changes to the graph, as the assignment depends purely on the id of the vertex and the number of workers.

Deterministic Greedy (DGR) is a state-of-the-art algorithm that produces partitions with good locality. It is a streaming technique that makes a single pass through the whole graph. Repartitioning our largest graph, which has a size of 3TB in memory, using this technique, would require a whole pass on the graph, incurring in overheads that would cancel the potential benefits from the improved partitioning. See Section V for more details on the limitations of scratch-map for dynamic graphs.

Minimum Number of Non-neighbours (MNN) is an alternative heuristic that attempts to minimise the number of non-neighbours when assigning the partition to a vertex [21] instead of maximising the number of neighbours. We implemented an iterative migration version of MNN and compared its results with our heuristic. MNN behaves similarly to our heuristic for small sized graphs, but its scalability is limited, as most vertices have a large number of non-neighbours that we need to locate and count, resulting in 3-5 times slower iteration times on average for the graphs under study.

Because of these restrictions, in all reported experiments we compare our adaptive heuristic with the industry standard hash partitioning.

C. *Evaluation of Heuristic*

In this section we explore different aspects of the performance of our heuristic. We measure how fast and with what overhead it can (i) converge to a good partitioning, and (ii) adapt to graph changes.

1) Convergence speed and overhead: Here we explore how the partitioning quality and the cost of our heuristic change during its execution. In particular, we measure the evolution of the ratio of cut edges across iterations, as well as the cumulative number of vertex migrations at each iteration. Figure 5 shows the cut ratio (dashed red), and ratio of migrations completed (solid blue) for the Livejournal graph. The graph was initially partitioned using modulo hash. The number of vertex migrations grows quickly in the initial iterations, with more than 50% of the migrations completed until the tenth iteration. During this stage the cut ratio is also improved to less than 0.7 from the initial 0.9. The rate of migrations slows down rapidly, and it takes to iteration 47 for the heuristic to migrate 90% of the vertices. At this stage, 90% of the ratio of cuts improvement has been achieved.

We observed similar behaviour in the improvement of cut ratio and number of migrations with different graphs and initial partitioning strategies. The first iterations of the heuristic trigger the majority of the migrations, as well as a significant part of the improvement in the partitioning. This has an important impact in performance. As the cut ratio decreases, computation performance improves thanks to the reduced communications cost. However, migrating vertices brings an additional overhead that might cause performance bottlenecks if too many migrations happen at the same iteration. The dampening factor of s, migration quotas, and the deferred migration technique, help to smoothen the initial peak of migrations.

From a performance point of view, the initial iterations will be affected the most by the additional overhead. Execution

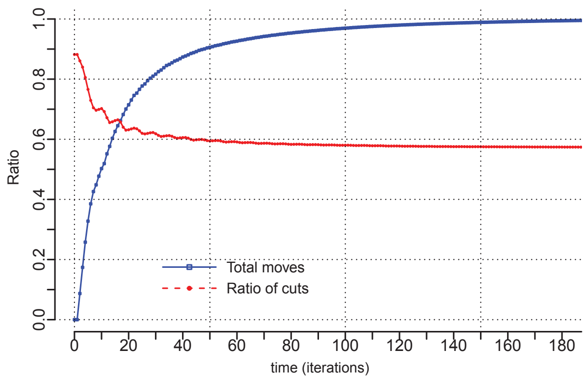

Figure 5. Accumulated migrations and cut ratio evolution on Livejournal graph.

times quickly decrease as the cut ratio improves, and later iterations quickly improve computation execution performance. As the overhead from vertex migrations decreases and the quality of partitioning improves, the computation execution time is optimized. We present the observed relationship between migrations, quality of the partitioning and performance in the following subsection.

2) Adapting to graph changes: Next, we evaluate the efficiency in adapting to changes in the graph. We loaded the Livejournal graph to our system, partitioned initially with modulo hash, and ran an analytical query that calculates an estimation of the diameter, using the algorithm in [13]. Every 50 iterations we inject to the graph a burst of new vertices based on a forest-fire expansion. Each addition increases the current graph size by 1, 2, 5 and 10%, respectively. Figure 6 shows the average time per step in seconds on LiveJournal with a static hash-partitioning approach and our dynamic heuristic (red). Time is divided in five regions, where we can observe the adaptation of the heuristic to the initial partitioning, and each one of the changes to the graph.

At the initial stage (0%), we observe the performance impact of the convergence behaviour of the algorithm we just discussed. Over the initial 10 iterations, where 50% of migrations take place, the overhead from migrations significantly affects computation performance, with the first five running almost 80% slower than the hash baseline, and the following five at roughly the same time. The next five iterations show a substantial improvement, with the average time decreasing to 54% of the time required by the hash baseline. The following iterations show considerably smaller improvements in the iteration execution time.

By observing the effect on a static partitioning we can conclude that the performance overhead from the heuristic is strongly dependent on vertex migrations. The heuristic is executed at every iteration, and does not overweight the benefits from an improved partitioning.

Now, let us observe the changes on execution time when we perform graph changes. First, looking at the static partitioning, execution time increases, growing up to an increase over 50% from the initial execution time. On the other hand, the adaptive heuristic shows similar behaviour for each graph injection. Initially execution time degrades due to the migrations overhead, but quickly the graph is adapted, and the execution time returns to figures almost identical to the ones obtained with the

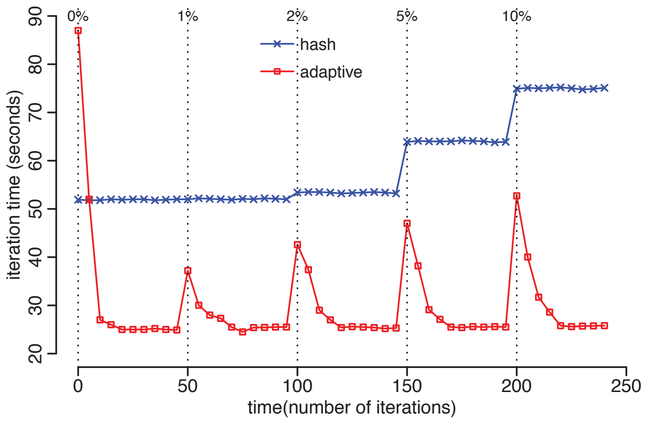

Figure 6. Execution time evolution after injecting changes to the Livejournal graph. Vertical lines show when changes were injected to the graph, as well as the percentage of additional nodes and edges that were injected in each case.

initial graph (0%). Larger additions to the graph inflict higher performance degradation over the first subsequent iterations, although after 10 iterations the heuristic has returned to values close to the optimal. The exact nature of the migrations overhead is heavily system dependent, but it becomes more taxing to the system when more abrupt changes occur.

The more abrupt the graph changes are, the more migration decisions will be potentially triggered in a single migration, therefore increasing the transient overhead of adaptive migration. It must be noted that we have tested the heuristic under the arrival of abrupt changes, whereas changes to dynamic graphs from real world sources tend to be considerably more gradual. We show details of these real world experiments in the following section.

D. Real-world Use Cases

We validated our system with a set of real-world use cases. We aimed at testing the scalability of the system, and the capability to cope with dynamic graphs in different scenarios. We believe that the diversity in the workloads of each application helps to support the general validity of our approach.

In all three cases, we ran the same experiments on two deployments of our system. One with the adaptive partitioning heuristic, and one with static hash partitioning.

*1) **Adaptation in real-time Social Network Analysis**:* Our first use case evaluates the capability of the system to analyse a dynamic graph modelled after a continuous stream of information. We aim to assess the adaptation capabilities of the heuristic, with respect of the evolution in the quality of the partitioning, and the impact in execution time.

We captured tweets in real-time from Twitter Streaming API, and built a graph where edges are generated from mentions of users. Over this power law graph, we continuously estimated the influence of certain users by using the TunkRank heuristic [33]. 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)

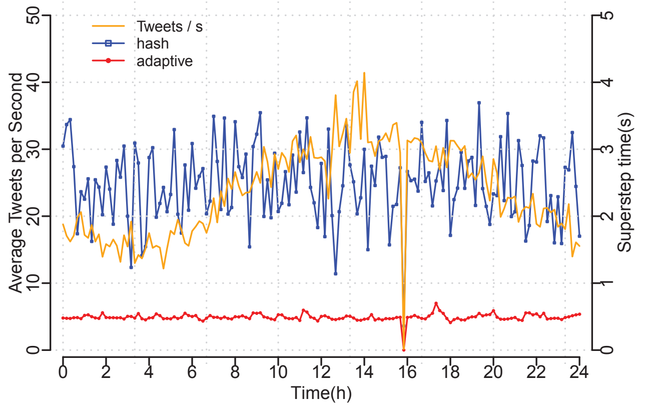

Figure 7. Throughput and performance obtained by processing the incoming stream of tweets originated from London. Each point represents the average of 10 min of streaming data.

We ran the experiment simultaneously in two separate clusters: One cluster used the adaptive heuristic, while the other used static hash partitioning instead. In Figure 7 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 yellow line shows the rate at which tweets are received and processed by the system, while the blue and red lines show average execution times per iteration, with and without adaptation, respectively. The sudden drop in throughput and iteration time is due to a failure in one of the workers that led to triggering recovery mechanisms. Note that the fact that it is represented as a 0 value does not mean it was faster, but it indicates the iteration was stopped to get back to a previous snapshot.

As can be observed, the average execution time is significantly improved when applying the adaptive heuristic, with mean of 0.5 secs instead of 2.5 secs, including the added overhead.

*2) **Adaptation in Mobile Network Communications**:* The second use case shows how our system can support online querying over a large-scale dynamic graph. We used a dataset from a mobile operator, with one month of mobile telephone calls. The dynamic graph was created by applying a sliding window to the incoming stream of calls as follows: Nodes represent users and calls are modelled as edges between these users. Therefore, new calls add nodes and vertices to the graph and both are removed from the graph if they are inactive for more than the window length (one week). The window size yielded weekly addition/deletion rates of 8 and 4%, respectively, which is higher than those reported in previous studies due to the shorter period of analysis [8].

Over this graph, we continuously computed the maximum cliques of each node. 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 heuristic 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

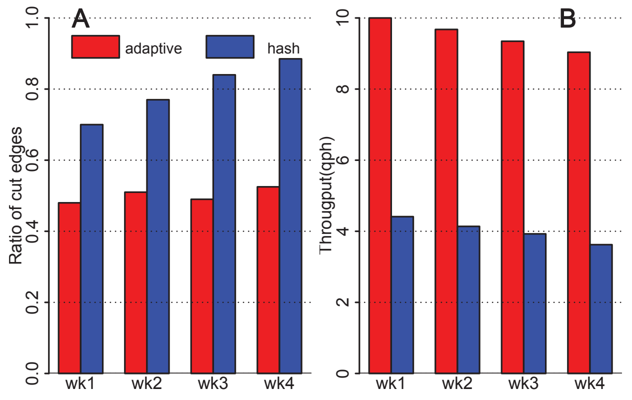

Figure 8. Evolution of the ratio of cuts **(Left)** and average iteration (step) time **(Right)** during the 4 weeks of available data. The experiments were performed in a cluster of 5 workers (96GB RAM, 10 GbE, 12 cores).

described later. The main problem of applying the iterative heuristic to this use case is that optimising message passing in iteration 1 places neighbours together and creates hotspots (all the members of a clique will be calculating the same cliques in parallel on the same host). To reduce duplicate calculations (reduce "hot zones") only lists for $j > i$ are created and only neighbours of neighbours with ID $j > i$ are added to those lists.

In contrast with the previous scenario, this application requires freezing the graph topology until a result is obtained, therefore buffering all the graph changes until the computation finishes (for two iterations instead of one). Meanwhile, adaptation occurs at every iteration. 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 cycle, further testing the adaptation capabilities of the heuristic.

We ran the clique finding application in two separate clusters, with and without the adaptive heuristic. Figure 8 shows weekly average figures for both cut ratio and throughput, in order to trace the performance impact. It can be seen that the adaptive partitioning heuristic maintained a stable number of cuts, resulting in consistently higher throughput (more than twice the throughput provided by hash partitioning). Moreover, weekly trends show that the static scenario experiences further performance degradation over time due to the higher cut ratio.

*3) **Adaptation in Biomedical Simulations**:* The final scenario assesses the suitability of the proposed system and heuristic for implementing large scale biomedical simulations. Biomedical simulations require long-running computations, with heavy CPU usage. Simulations are often implemented on specialised clusters, using message-passing libraries such as MPI. The use case presents a different type of application (long-running simulations), that operates at a considerably higher scale than the previous scenarios.

The input graph is 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

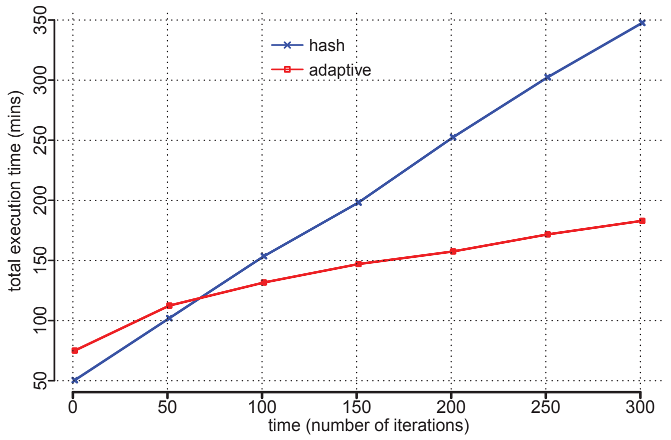

Figure 9. Cumulative execution time after expanding the heart cell FEM using a forest fire model (extra 10^7 vertices and $3 * 10^7$ edges). Results were obtained in a cluster of 63 workers (64GB RAM, 10GbE and 12 Cores)

and blood pumping [32]. The graph state occupies a total of 3TB in memory among the 63 worker machines running the simulation. Using static hash partitioning (without the adaptive heuristic), simulation time is still dominated by the exchange of messages (more than 80% of the time), even though CPU time is not negligible (more than 17%). The iterative heuristic works in this use case similarly to the other experiments, achieving a final speedup of 2.44 after convergence.

At a certain point in the simulation, we mimic the effect that adding stem cells differentiating into cardiac tissue would have. These new cells are injected to the graph as an additional 10M vertices and 30M edges, joining the tissue in the border of the infarcted region, based on preferential attachment. These changes bring the total memory usage close to full occupation in the cluster.

We show in Figure 9 the cumulative execution time, from the instant changes were added to the graph structure, for both a static hash partitioning (blue) and our iterative heuristic (red). As expected, the first iterations are affected by the overhead from the triggered vertex migrations, but in the long term the improved partitioning significantly shortens simulation time. Comparing these results with previous use cases the heuristic performs better on continuously changing graphs. It will be worth to adapt to abrupt changes in the graphs only when facing long-running computations, such as biomedical simulations.

V. Related Work

The idea of dynamically adapting a static graph to changes in load of the graph was pioneered by the high performance computing (HPC) for FEMs. Dynamic partitioning methods can be broadly classified as: scratch-map, those that create a new partition from scratch, and diffusive, those that take previous partitions into account (see [27], for example).

Scratch-map Partitioning heuristics are executed at load time to improve performance in massive graphs [34]. Another example of initial load optimisation heuristics are stream partitioning techniques [29]. Streaming heuristics process a stream of graph nodes and assign each node to a partition based on the knowledge of previous decisions (accumulating global knowledge as the graph is streamed). The streaming finished once the graph file has been completely read.

Being executed only once, scratch-map techniques cannot prevent performance degradation arising from changes to the graph structure over time. In the FEM graphs community the proposed approach to adapt to graph changes is to execute partitioning periodically, creating a whole new set of partitions each time [27]. However, recomputing from scratch and creating new sets of partitions is not only time consuming because the whole graph needs to be re-evaluated, but also implies a huge overhead in object deletion and creation.

Recently more sophisticated techniques have been introduced to optimise graph reloading (assuming not many changes have occurred) [19]. These re-streaming mechanisms make a new pass of the graph every time adaptation is needed, which may not scale even when partition parallelisation is doable in separate workers (streaming the whole graphs is still required on every parallel worker). Also, there is a high volume of communication between workers between restreams: each worker reports on their share of the partitioning and this compiled list is distributed to all workers for the next restream.

Summing up, scratch-map partitioning produces poorer results when adaptation is light or changes are scattered across the whole graph, since the new partitioning may be totally different to the old one. They also require some degree of global knowledge that calls for synchronisation mechanisms in parallel settings. In the context of FEMs, scratch-map geometrical, spectral or multi-level methods require global visibility of the graph to make the best partition decision [27]. Streaming techniques accumulate previous partitions to make decisions more accurate on incoming vertices, but still require keeping track of already "seen" vertices at a global level. Also, scratch partitioning of large graphs may take a few hours (see [31]), which shows how costly this may be.

Diffusive Partitioning methods work under the assumption that only a small fraction of the graph has changed and most of the work done in partitioning, loading and instantiating objects can be reused.

Most diffusion-based repartitioners for FEMs are based on *flow solutions* that prescribe the amount of vertices to be transferred between partitions, which requires global knowledge of the graph, the partitions and the work performed by each vertex (see [27]). Multilevel local partitioning solutions exist for FEMs [26], [4], but these assume the initial coarsening phase has already been done and work on a static mesh graph where load needs to be balanced and partitions consequently change but no new entities are introduced in the graph at runtime.

ParMETIS [20] is arguably the most salient example, it moves vertices belonging to the borders of neighbouring partitions trying to minimise edge-cuts across partitions while keeping partitions balanced until no additional gain can be obtained. ParMETIS leverages parallel processing for partitioning the graph, through multilevel k-way partitioning and parallel multi-constrained partitioning schemes. While its hierarchical approach is excellent for FEM networks, it requires global visibility during the initial partitioning phase: all the pieces of the graph are scattered to all threads using an all-to-all broadcast operation.

As mentioned above, ParMETIS relies on the Fiduccia-Mattheyses algorithm, which makes heavy use of a bucket data structure to sort all possible vertex moves per descending edge cut gain. Candidate vertices are taken out of the bucket to check whether moving them would preserve balanced partitions. If not, the vertex is put aside and the next one is taken out of the bucket. Once a vertex has been moved all the vertices kept aside are reintroduced in the bucket. While this works well for a single partitioning, it is extremely ineffective when applied to repartitioning. This heavy cost of re-executing for repartitioning the graph as its topology changes explains why ParMETIS has been used by some systems for initial partitioning only [13].

Beyond FEMs, GPS [25] lets vertices move to the partition where most of their neighbours are. To simplify location of a migrated vertex its ID (used to determine in which machine the vertex has been placed) is changed when the vertex is migrated. Adding new vertices would required fine grained synchronisation between workers, so that the IDs of new vertices do not conflict with migrated ones.

Other Dynamic Approaches. While we focused on changes in the topology of the graph, other works focus on different aspects of graph dynamism. For instance, some systems dynamically adapt the partitioning of the graph to the bandwidth characteristics of the underlying computer network to maximise throughput [6].

Dynamic replication techniques replicate parts of the partition or whole partitions to load balance work across replicas, preserving low latency in responses [36], [22]. These approaches cannot cope with a continuous stream of changes in the topology and their initial static partitions would eventually be obsolete, making replication too fragmented.

Enabling graph mining applications in real-world environments calls for scalable partitioning heuristics that take dynamic changes in the topology of the graph into consideration to mitigate performance degradation. Current parallel diffusive/scratch-map partitioning methods do not scale well since they some degree of require global knowledge and coordination.

Algorithms for Dynamic Graphs In our experiments we have shown how our system applies algorithms to dynamic graphs in two ways. Some algorithms, such as TunkRank, can operate on a changing graph, whereas in other cases we have to buffer graph changes and freeze the graph during computation. As information becomes more dynamic, and results are needed in a shorter time, a new breed of algorithms that can cope with changes in the topology of the graph is needed.

Previous approaches on algorithms for dynamic graphs were based on observing the discrete evolution of classic topological metrics of the graph (such as time-respecting paths, connectivity, network efficiency, centrality, patterns/motifs), see [11] for a recent review. More recent work tries to redefine some of these metrics to take dynamism into account [18].

VI. Conclusions

To the best of our knowledge, there is no system that continuously processes large-scale dynamic graphs, while adapting the internal partitioning to the changes in graph topology. Our system adapts to large-scale graph changes by repartitioning while 1) greatly reducing the number of cut edges to avoid 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.

Real world graphs are dynamic, and mining information from graphs without considering the evolution of their structure over time can have a significant impact on system performance.

In this work we have focussed on adapting to graph changes in a highly scalable way, while working under the challenges of migrating vertices in a distributed system. The presented heuristic adapts the graph partitioning to graph dynamics at the same time as computations take place. We show through our experiments that the heuristic improves computation performance (with higher than 50% reduction in iteration execution time), adapting to both continuous and abrupt changes.

A key performance factor for adapting to graph changes is the tradeoff between the additional overhead incurred by repartitioning the graph, and the effective performance improvement from a better graph partitioning. We have found vertex migration to be the predominant source of overhead (specially when migrating a high number of vertices), and we will work on further system optimisations for efficient vertex creation and migration and characterisation of graph growth/shrinkage.

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] J. G. Castaños and J. E. Savage. Repartitioning unstructured adaptive meshes. In *IPDPS*, pages 823–832. IEEE Computer Society, 2000.

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

[6] 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.

[7] 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.

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

[9] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2008.

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

[11] P. Holme and J. Saramaki. Temporal networks. *Phys. Rep.* 97–125, 2012.

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

[13] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, Apr. 2013.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] J. Nishimura and J. Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '13, pages 1106–1114, New York, NY, USA, 2013. ACM.

[20] ParMETIS. Parmetis - parallel graph partitioning and fill-reducing matrix ordering, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview, October 2012.

[21] 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.

[22] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 375–386, New York, NY, USA, 2010. ACM.

[23] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), 2007.

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

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

[26] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes, 1997.

[27] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, May 2001.

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

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

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

[31] Y. Tian, A. Balmin, S. Corsten, S. Tatikonda, and M. J. From ̈think like a vertex ̈to ̈think like a graph. ̈*The Proceedings of the VLDB Endowment (PVLDB)*, 77(5-6):185–195, 2014.

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

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

[34] 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.

[35] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[36] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition