xDGP: A Dynamic Graph Processing System with Adaptive Partitioning

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

Many real-world systems, such as social networks, rely on mining efficiently large graphs, with hundreds of millions of vertices and edges. This volume of information requires partitioning the graph across multiple nodes in a distributed system. This has a deep effect on performance, as traversing edges cut between partitions incurs a significant performance penalty due to the cost of communication. Thus, several systems in the literature have attempted to improve computational performance by enhancing graph partitioning, but they do not support another characteristic of real-world graphs: graphs are inherently dynamic, their topology evolves continuously, and subsequently the optimum partitioning also changes over time.

In this work, we present the first system that dynamically repartitions massive graphs to adapt to structural changes. The system optimises graph partitioning to prevent performance degradation without using data replication. The system adopts an iterative vertex migration algorithm that relies on local information only, making complex coordination unnecessary. We show how the improvement in graph partitioning reduces execution time by over 50%, while adapting the partitioning to a large number of changes to the graph in three real-world scenarios.

Categories and Subject Descriptors

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

Keywords

Dynamic graphs, large-scale graphs, graph processing, adaptive graph partitioning, distributed heuristic

1. INTRODUCTION

Many critical analytical tasks in real-world systems, such as ranking and recommending online content or discovering groups of correlated stocks, depend on the ability to mine graphs of hundreds 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 [9, 29, 1, 7, 35, 21, 18, 38].

Most graph processing systems adopt a batch job model, greatly influenced by Pregel [22]: A static graph is loaded in the memory of a distributed system to cope with its very large scale. Computations are performed in multiple iterations that involve per-vertex processing and messaging exchange between neighbouring vertices. Graph partitioning is crucial to performance in these systems: Balanced graph partitioning helps with load balancing, and minimising the number of cuts between partitions improves communication cost between neighbouring vertices [22].

These systems process static graphs, but many real-world graphs are dynamic (the graph structure changes over time) and several analytical applications 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 systems must reflect these changes within minutes, otherwise they become irrelevant. Similarly, telecommunications operators must detect fraud by mining the Call Detail Record (CDR) graph in real-time [44]. In addition to the existing batch processing systems, there is a need for frameworks that simplify continuous processing of dynamic massive graphs.

Supporting dynamic graphs brings new challenges. As the graph structure changes over time, if partitions were not updated, performance would constantly be degraded due to additional communication overhead and unbalanced partitions (processing bottlenecks). However, these are often conflicting requirements that make optimisations difficult and more so when rapid responses are needed.

In this paper, we present a system for processing large-scale dynamic graphs. We have addressed the performance challenges described above by implementing a scalable partitioning heuristic with minimum overhead. Our heuristic

is based on decentralised, iterative vertex migration. The heuristic migrates vertices between partitions trying to minimise the number of cut edges, while at the same time keeping partitions balanced upon structural changes at run time. Updates in the graph topology trigger the iterative vertex migration process that adapts partitioning to the new topology.

This paper presents the following contributions:

1. We report how changes in the structure of a graph impact the quality of the graph partitioning, as even high quality initial partitioning strategies failing to adapt to graph dynamics.

2. We describe a system that processes continuously large-scale dynamic graphs, automatically adapting to structural changes on the graph for improved performance.

3. We present a highly scalable, completely decentralised heuristic based on label propagation. The heuristic improves graph partitioning in the event of dynamic changes in the topology of the graph relying only on local information.

4. We present extensive evaluation of the heuristic through system deployments, using synthetic and real datasets. Experiments show the effectiveness of the heuristic in adapting to graph changes, the associated performance improvement, and the observed impact in different real scenarios.

The rest of the paper is organised as follows. Section 2 presents the motivation for addressing partitioning of dynamic graphs. In Section 3, we describe our heuristic in more detail. Section 4 describes some relevant pitfalls that need to be overcome when realising the heuristic into a real system. The heuristic is then tested at scale in a series of lab experiments and real-world use cases in Section 5. Section 6 compares our contributions to the related work. We present the main conclusions and discussion in Section 7.

2. DYNAMISM IN GRAPH PARTITIONING

The number of cut edges in a distributed graph processing system directly impacts the communications overhead of computations across the whole graph, up to the point of becoming a key performance factor [22]. The performance impact of graph partitioning has led to several optimisations at the beginning of the processing, right when the graph is being loaded in memory. For instance, popular heuristics for content ranking converge faster if initialised with smarter graph partitioning heuristics [40]. At load time, clever partitioning heuristics to improve performance in massive graphs have also been employed [42, 34, 37]. None of them is capable of preventing performance degradation arising from changes to the graph structure over time.

To illustrate the impact of dynamism in graph partitioning, we built a call graph from mobile CDR data (more details are shown in Section 5). The graph was partitioned
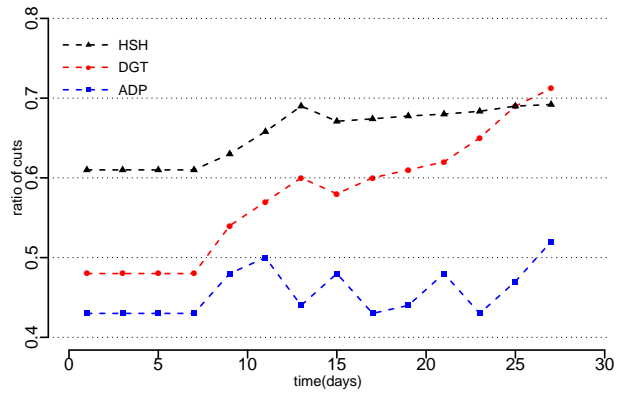

Figure 1: Evolution of the ratio of cuts over time on a dynamic graph generated by processing CDR data on a call window.

using three different techniques: *modulo hash* (HSH), the most popular technique because of its high scalability to produce balanced partitions, results in high communication overheads [17]; a state of art streaming partition technique (*deterministic greedy*, DTG) [37]; and our *adaptive repartitioning* heuristic, as described below, (ADP). Figure 1 shows the evolution of the partitioning (expressed in the percentage of edges that cut across different partitions). As the graph evolves over time, a static or streaming partitioning solution cannot cope with the changes and, consequently, the quality of the partitioning (and the system performance) degrades over time.

Current approaches either allow partitioning to degrade as the graph changes or require graph re-partitioning, which is a very costly process on large-scale graphs. Both of them effectively increase processing time. While this problem does not deeply affect batch processing systems, it can greatly impact the throughput and latency of graph processing systems requiring faster response times.

To enable graph mining applications in real-world environments, such as online social networks, we need scalable partitioning heuristics that take the dynamic nature of the graphs into consideration. This requires the ability to quickly adapt the partitioning as the graph changes to prevent performance degradation.

3. ADAPTIVE ITERATIVE PARTITIONING

In this section the iterative heuristic for improved dynamic graph partitioning is presented. Before we describe the heuristic, the next subsection describes the problem and context that any solution should address.

3.1 Problem Statement

Definition *(Dynamic Graph Partitioning)* 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 $\bigcup\limits_{i,t}^{k} P^i(t) = V$ 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 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 $H(v) \bmod k = i$. This strategy is effective as it is lightweight, it does not require a global lookup table, and, depending on the characteristics of the vertex IDs, 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.

The goal of repartitioning is to minimise the number of cut edges across partitions, while keeping the partitions balanced in order to improve application performance. Reducing the number of cut edges diminishes the communications overhead of computations across the whole graph, while load balancing can have a significant impact on overall processing time [22]. The characteristics of dynamic graphs and the context of application also bring several challenges that must be considered for any solution:

- *Graph structure changes are not predictable*. It is not possible to know beforehand the nature of the changes, or the impact they will have in performance, unless working on a specific use case. Therefore, partition optimisation should apply to any change to the graph, and must be able to run continuously if the graph changes all the time.

- *The computational overhead from the partitioning optimisation must be low*. As the objective is to improve application performance, the selected technique must be lightweight and scalable to work over large graphs.

- *Synchronising distributed state at a large scale is very costly in a dynamic environment*. Computations will have a partitioned view of the complete system state. Propagating global information across the network incurs a significant overhead, which must be considered for every repartitioning technique.

3.2 Greedy Vertex Migration

We have defined a heuristic for dynamically adapting graph partitions that considers the challenges above. The heuristic is based on label propagation [31], adapting the approach of performing iterative computations based on per-vertex neighbour information. Label propagation was first proposed as an efficient method for learning missing categories in semi-supervised learning scenarios. Unlabelled nodes iteratively adopt the label of the majority of their neighbours until no new labels are assigned (convergence is reached). The technique has been adopted in the literature for supporting adaptive migrations on static graphs[42].

The iterative heuristic works as follows. 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. 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 $|E_c|$. 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 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 the number of neighbours placed together.

Dynamism comes natively in this iterative approach. New vertices are initially assigned a partition according to an strategy (we opted for hash modulo) and the heuristic will attempt automatically to move them closer to their neighbours.

For vertex migration decisions we evaluated multiple alternatives based on local information [37, 30]. We chose a greedy heuristic that had the lowest computational cost while yielding strong results regarding minimising cut edges. The heuristic works as follows: At each iteration, a vertex will decide to migrate to the partition with the highest number of neighbour vertices. 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: $cand(v,t) = \{P^i(t) \in P(t), \exists \, w, \, w \in (P^i(t) \cap \Gamma(v,t))\}$, 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.

The heuristic relies on local information, as each vertex v only needs to know the location of its neighbours in order to choose its destination. In a real system, that information will already be available at each partition (in order for vertices to communicate with their neighbours). The heuristic does not need of additional coordination mechanisms for sharing further information, which might hamper scalability.

3.3 Maintaining Balanced Partitions

The greedy nature of the presented heuristic will naturally

---

[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

cause higher concentration of vertices in some partitions. We refer to this phenomenon, common to general label propagation algorithms [31], 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 capacity 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 4 for system implementation details

Quotas can defer migration decisions, which has a side effect on the real system performance. On a real system, vertex migrations are a costly activity than can affect application performance: They involve messaging across partitions, object creation/destruction, and memory reservation, incurring on significant overhead. A lesser number of migrations per iteration reduces the extra load imposed by the migration decisions to the system, as well as the maximum overhead imposed to the system in a single iteration.

This strategy to manage partition capacity 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.

3.4 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 whatso-
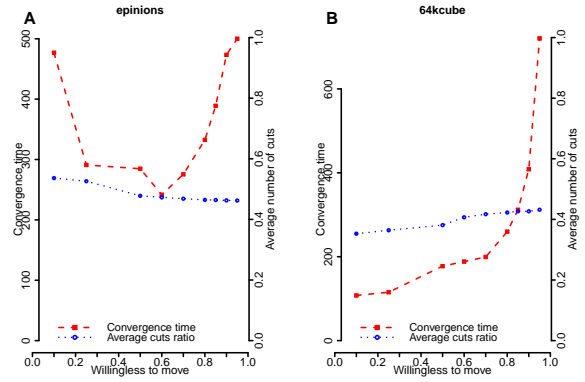

Figure 2: Effect of s into Convergence and Number of Cuts (normalised to the total number of edges in the graph). Average of 10 experiments performed over two graphs: 64kcube (A) and Epinions(B) from Table 1, partitioning over 9 nodes.

ever, while $s = 1$ allows 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 selected graphs are provided in Section 5.1. 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 1.

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.

4. SYSTEM DESIGN

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

4.1 Computation Model

xDGP takes Pregel's computational model ("think like a vertex") as inspiration and expands it to a continuous dynamic graph processing model, instead of focussing on batch jobs. Computation is composed of iterations, where the same function is executed for each vertex. A job defines a function applied to each vertex, synchronising messages at the end of every iteration.

A job starts processing an initial graph, which is loaded into the system with an initial partitioning. The system also provides an external API for modifying the topology of the graph at any time. Functions allow adding and removing vertices and edges from the graph. API topology change requests are added to a change queue, and are processed at the end of every iteration (or potentially, after n iterations, as shown in the CDR case (Section 5.3).

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

4.2 Vertex Migration Support

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

Deferred Vertex Migration.

At any iteration t, vertices take 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 $t+1$ (this would require immediate notification to their neighbours of the migration decision, that would need to be received and processed in the same iteration, increasing the overhead from the migration process). Migrating a vertex at the next iteration 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 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. At the end of iteration t, at which the vertex requested the migration, the host worker sends a message to 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.

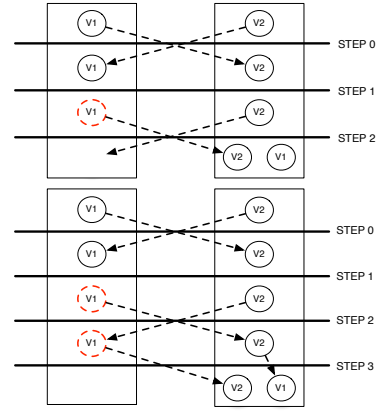The heuristic requires the system to maintain an extra el-

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.**

ement of global information: Each worker must notify the $|C^i(t)|$ of its partitions to the other workers. We implemented these communications using the scalable messaging form our computation model, with the restriction that messages will only be received and processed by the next iteration. Therefore, when using this mechanism, workers send information about their capacity at iteration $t+1$, ensuring partial freshness. The predicted capacity will be $C^i(t+1) = C^i(t) - V_{out}^i(t+1) + V_{in}^i(t+1)$, where $V_{in}^i(t) \subset V$ are the vertices migrating to i in $t+1$, and $V_{out}^i(t) \subset V$ are the vertices migrating from i to other partitions in $t+1$. $V_{out}^i(t+1)$ is known by the worker as it is based on local decisions. $V_{in}^i(t+1)$ is also known by the worker at iteration $t+1$ as deferred vertex migration ensures that the workers will be aware of this value.

4.3 System Implementation

xDGP is implemented in Java, and has as main design goals support for dynamic graph adaptation, failure tolerance, and intermediate results snapshot. Following the Bulk Synchronous Processing [43] model, the main system blocks are Master and Workers, as shown in Figure 4. Master and workers communicate synchronously (RMI) to enforce the global synchronisation barrier. Similarly to Pregel, xDGP implements an abstract Vertex class that hides all the complexity to 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 interworker vertex communications, sending messages through a loosely coupled asynchronous delivery method (we used RabbitMQ and ZeroMQ as interchangeable message handlers). xDGP also provides a generalisable message aggre-

gation mechanism (grouping messages sharing same source and destination workers, in order to reduce the routing inefficiency introduced by RabbitMQ).

Snapshots (for failure-tolerance or for keeping the output of the running heuristic) are stored on a distributed-column store cluster (replicated configuration), keeping balance between writing speed and consistency. xDGP tries to find the right balance between failure-tolerance and high write throughput (so as not to delay computation). Frequent snapshotting and high write-throughput are specially important for dynamic graphs, since intermediate analysis results must be kept for the external applications to show the output and its evolution.

The system supports two types of messages. Application (vertex to vertex) messages contain data related to the computation, and system messages (worker to worker) 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 helps to find the current location of that vertex.

When new vertices are added, the partitioner of each of the worker nodes is in charge of properly allocating them to one of the workers. This decision is important to reduce convergence time: a random placement strategy works just as well since that new vertex would be migrated around until it finds most of its neighbours are local. This approach is preferred to more complex placements (that involve more coordination messages and delay the migration process that occurs in between two iterations). Buffers (distributed queues) are in charge of dampening new requests to add/delete graph elements. Queues for vertex or edge deletion/addition can be prioritised.

5. EVALUATION

In this section, we present the evaluation experiments we have undertaken for testing the dynamic graph adaptation capabilities of the proposed system and heuristic. First, we validate the capabilities of the heuristic through a set of microbenchmarks, looking at different aspects of adaptive migration (quality of the partitioning, convergence time and distribution of migrations, and absorption of graph changes). Second, we demonstrate through three use cases how the system and heuristic can improve the processing of real scenarios, namely real-time social network analysis, online queries from CDR records, and large-scale biomedical simulations.

5.1 Datasets

We have selected a diverse 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 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 [39]. Power law synthetic graphs have been generated with networkX [13], using its power law degree distribution and approximate average clustering [14]; the intended average degree is $D = log(|V|)$, with rewiring probability $p = 0.1$.

When required, we mimicked dynamic changes to the synthetic graphs by adding nodes and vertices using the well-known "forest fire" model [19], and updating the graph with these additions in a single step.

Table 1: 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	[36]
4elt	15606	45878	FEM	[36]
plc1000	1000	9879	pwlaw	synth
plc10000	10000	129774	pwlaw	synth
plc50000	50000	1249061	pwlaw	synth
wikivote	7115	103689	pwlaw	[20]
epinion	75879	508837	pwlaw	[32]
livejournal	4847571	68993773	pwlaw	[2]

In addition to these graphs, we also used two 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 data of anonymised calls in a mobile European operator. We fed these data chronologically, building a dynamic graph of call interactions, consisting of 21 million vertices, 132 million reciprocated social ties.

5.2 Microbenchmarks

The goal of these experiments is to understand the quality, performance and cost of the adaptive migration heuristic. For quality, we adopted the *cut ratio*, i.e. the ratio of edges cutting across different graph partitions. From the application performance point of view, the lower the cut ratio the lower the number of communication messages that will be sent across the distributed system.

We estimate the runtime overhead of the heuristic by characterising how the heuristic triggers *migration of vertices* over multiple iterations.

Finally, we compare our estimations with the measured performance of the heuristic (average *computation time* of an iteration).

We provide for comparison the results obtained by running the same experiments in our system, without adaptive partitioning. All the experiments shown below are the mean of $n = 10$ repetitions. Variability is reported in the form of standard deviation in the error bars.
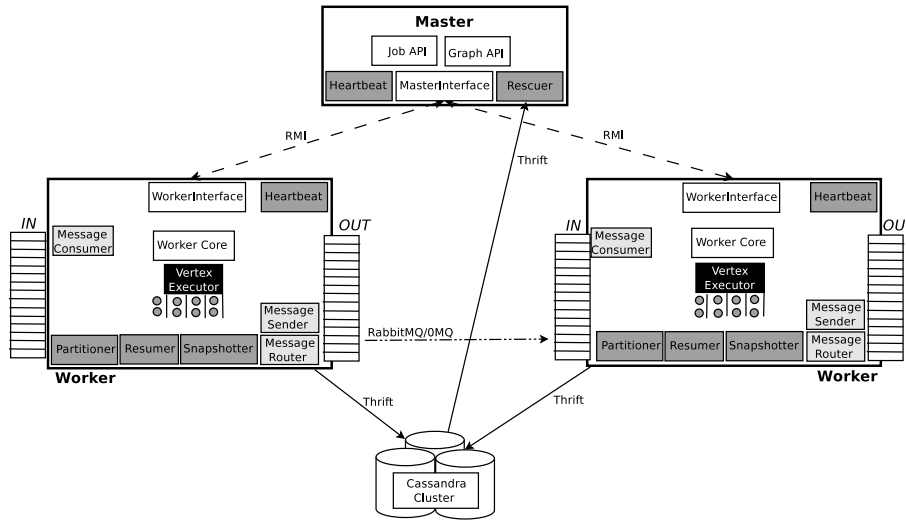
Figure 4: xDGP overview. Small grey dots represent the vertices and the vertical lines under the Vertex Executor box indicate that several concurrent executors run in parallel in a multi-core host.

5.2.1 Quality of partitioning

As we mentioned above, some systems just optimise the initial loading of the graph in memory, but provide no later adaptation to cope with structural changes in dynamic graphs. Different initial partitioning techniques split the graph in different ways that may affect the way our heuristic behaves during run time adaptations.

In this experiment we used different graphs and studied if our heuristic could lead to further improvements in performance on nine partitions. We tested several initial partitioning strategies: 1) *Hash Partitioning (HSH)*: the destination partition is computed by hashing vertexId with modulo number of partitions; 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 [37]; 4) *Minimum Number of Neighbours (MNN)*: applies the same stream-based approach to the "minimum number of neighbours" heuristic presented in [30]. After initialising the partitions with one of four different initial strategies, we ran our adaptive iterative heuristic until convergence.

Figure 5 provides the results obtained for each graph on average of 10 experiments. Each group of four bars shows the results for a different graph, partitioned with the four initial partitioning strategies. The graph overlaps the bars for the cut ratio of both the initial partitioning (dashed colour fill), and the improved final partitioning after running the adaptive heuristic (filled colour). The improvement obtained by the iterative heuristic (if any) from the initial partitioning is represented by the visible dashed colour bar.

Looking at HSH initial partitioning, the iterative heuristic significantly improves the cut ratio for FEM graphs (the five leftmost bar groups), with greater than 0.6 improve-

ment. Adaptive partitioning also provides substantial improvements for RND and MNN partitioning strategies. When applying it to a state-of-the-art initial partitioning technique (DGR), it only slightly improves the cut ratio, since the heuristics have a very similar (greedy) nature. It is worth noting that for large-scale graphs, DGR and MNN require full graph knowledge, which poses significant limits to its scalability and its applicability to real deployments [42]. For real use cases we have used hash, as it is the de facto standard used by most other large-scale partitioning systems. However, we believe these results show that the adaptive heuristic is compatible with state-of-the-art partitioning techniques, and would optimise the partitioning when graph dynamism is required (see Figure 1).

Looking now at power law graphs (the four rightmost bar groups), non-DGR initial partitioning is also improved by the iterative heuristic. For these types of graphs the final cut ratio is higher than the levels achieved on FEM graphs. The results show that DGR could not provide either a low cut ratio for these graphs, showing that they are more difficult to partition.

5.2.2 Convergence Study

We study how the adaptive heuristic converges to a partition distribution. To this end, we collected the number of migrations between vertices at each iteration, as well as the evolution of the ratio of cut edges. Figure 6 shows cut ratio (red), and ratio of migrations completed (blue) for the Livejournal graph. The graph was partitioned using modulo hash. Cumulative vertex migrations are triggered rapidly in the initial iterations, with more than 50% of the moves completed at 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
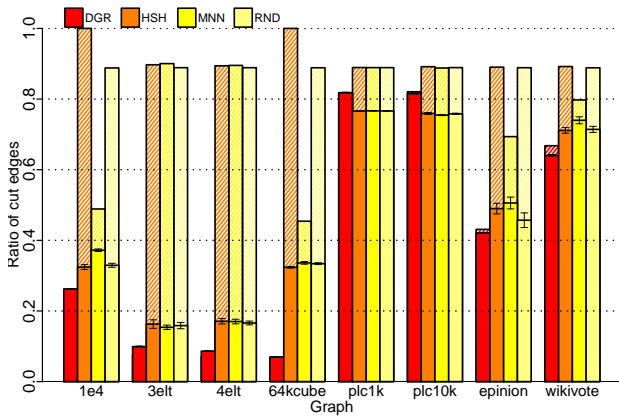
Figure 5: Ratio of cut edges for each graph after running the iterative heuristic over four different initial partitioning strategies. Clearer bars show the ratio with the initial partitioning

Figure 6: Accumulated migrations and cut ratio evolution on Livejournal graph.

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 triggered number of migrations with different graphs and partitioning strategies. The first iterations of the heuristic trigger the majority of the movements, as well as a significant part of the improvement in the partitioning. This has an important impact in the system performance. As the cut ratio is reduced, computation performance will improve 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.

It can be predicted that from a performance point of view the initial iterations will be affected the most by the additional overhead. In this setting iteration execution times quickly go down as the cut ratio improves and later iterations will quickly improve computation execution performance, as both the overhead from vertex migrations goes down and the quality of partitioning quickly improves, therefore optimising computation execution time. We present the observed relationship between migrations, quality of the partitioning and performance in the following subsection.

5.2.3 Performance of Adaptation to Dynamic Changes

We evaluate the performance (in iteration time) of the system can adapt to controlled changes in the graph. We loaded the Livejournal graph to our system, partitioned initially with modulo hash, and calculated the estimated diameter, with the heuristic used in [17]. Every 50 iterations we injected to the running graph a burst of new vertices (based on a forest-fire expansion), each addition increasing the current graph size by 1, 2, 5 and 10%, respectively. The graph was continuously processed to compute the diameter of the graph. Figure 7 shows the average time per step (in minutes) on LiveJournal with a static approach (hash partitioning - blue)
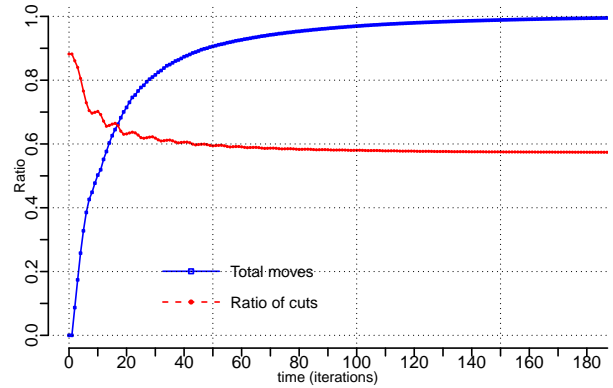
and our dynamic heuristic (red). Time is divided in five regions, where we can observe the adaptation of the heuristic to the initial partitioning, and each one of the changes to the graph.

At the initial stage (0%), we can 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 going down to 54% of the time required by the hash baseline. The following iterations show considerably smaller improvements in the iteration execution time.

We can conclude that the performance overhead from the heuristic is strongly dependent on vertex migrations. The decision part of the heuristic is executed at every iteration, and does not seem to overweights the benefits from an improved partitioning. For performance reasons, it will be worth to adapt the graph when computations will take place for an extended number of iterations. Otherwise, it is possible that the initial overhead will cancel any benefits form achieving a improved partitioning.

Now let us observe the changes on execution time when we inject graph changes. First, looking at the static partitioning, execution time increases, growing up to an increase over 50% from the initial 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 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 the more abrupt changes occur.
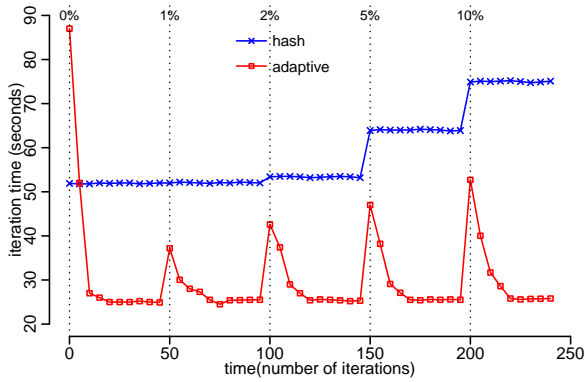
Figure 7: Execution time evolution after injecting changes to the Livejournal graph.
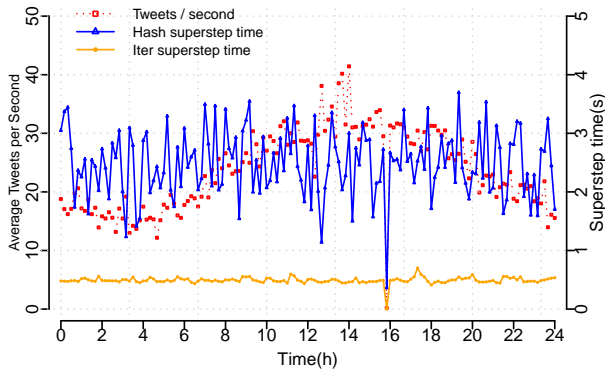

Figure 8: Throughput and performance obtained by processing the incoming stream of tweets originated from London. Each point represents the average of 10 min of streaming data.

5.3 Real-Word 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.

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 given by mentions of users. Over this power law graph, we continuously estimated the influence of certain users by using the TunkRank heuristic [41]. 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 heuristic, 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 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 the triggering of xDGP recovery mechanism.

As can be observed, the average execution time is significantly improved when applying the adaptive heuristic, (mean of 0.5 secs instead of 2.5 secs, including the added overhead). Importantly, the optimisation of the partitioning with only local information significantly lessens variability in execution times, by reducing the impact of network communications (more neighbours are local).

Adaptation in Mobile Network Communications.
The second use case shows how xDGP 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 [10].

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 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 compu-
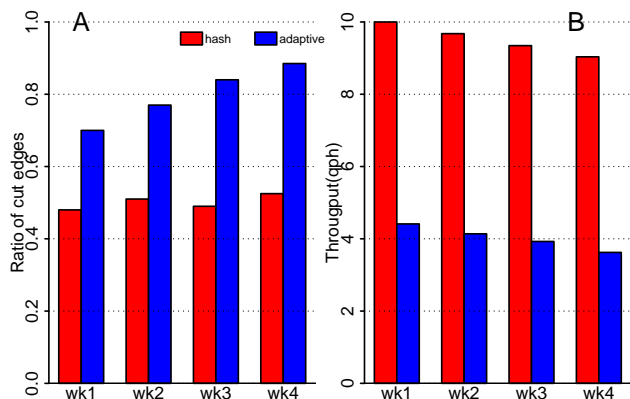
Figure 9: Evolution of the ratio of cuts (Left) and average iteration (step) time (Right) during the 4 weeks of available data. The experiments were performed in a cluster of 5 workers (96GB RAM, 10 GbE, 12 cores).

tation 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 run the clique finding application in two separate clusters, with and without the adaptive heuristic. 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 higher throughput (more than twice the throughput provided by hash portioning). Moreover, weekly trends show that the static scenario experiences further performance degradation over time due to the higher cut ratio.

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 and blood pumping [39]. 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 it-
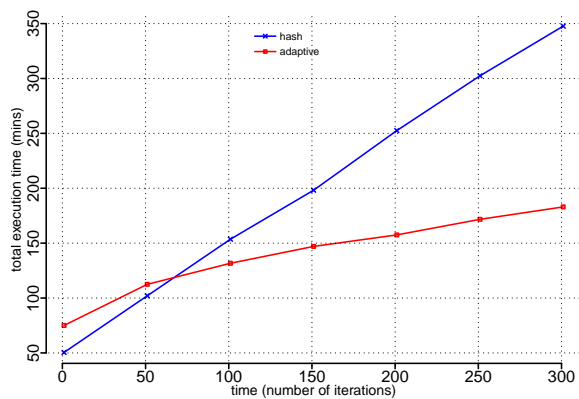

Figure 10: Accumulative execution time after expanding the heart cell FEM using a forest fire model (extra 10^7 vertices and $3 * 10^7$ edges. Results were obtained in a cluster of 63 workers (64GB RAM, 10GbE and 12 Cores)
.

erative 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 10 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.

6. RELATED WORK

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.

6.1 General Partitioning

The idea of partitioning the graph to minimise network communication is not new and it has inspired several tech-

niques to co-locate neighbouring vertices in the same host [5, 28, 16, 25, 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 parallel version of METIS [15], ParMETIS [27], 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 [34]. Other techniques have been explored that study graph properties projected onto a small subset of vertices [19, 11]. These may be effective in some particular contexts, but they are not broadly applicable.

6.2 Changing Graph Structure at Runtime

Beyond these fixed techniques for static graphs, the need to continuously adapt to changes to the graph structure without the overhead of re-loading an updated snapshot of the graph or re-partitioning from scratch, has been recently reported in practical [33, 23, 12] and more theoretical [26] studies. Few systems can cope with run time changes in structure [24, 45, 9]. However, these cannot handle structural graph changes, either degrading partition quality or fully triggering the partitioning process.

6.3 Initial Partitioning Strategies

Some techniques try to alleviate performance degradation by optimising partitioning during the initial loading of the graph in memory (i.e. they do not adapt in run time). For instance, in [37] 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. In addition to adaptations to changes in structure, some systems dynamically adapt the partitioning of the graph to the bandwidth characteristics of the underlying computer network to maximise throughput [8]. Mizan [17] ignores the graph topology and instead optimises the system by performing runtime monitoring and load balancing. The graph processing system finds hotspots in specific workers and migrates vertices to a paired worker who have the highest number of outgoing messages in an attempt to balance the load.

GPS [34] applies the technique most similar to ours from the heuristic point of view, but system implementation limits its application to static graphs. There are two main differences: 1) they allow vertices to move while an iteration is still running, while we move vertices between two consecutive steps; 2) to simplify location of a migrated vertex, they modify the ID. This prevents adding new elements, since their ID may conflict with one of a previously loaded and migrated vertex. We preserve de ID of the vertex by using a more complex vertex localisation mechanism, which en-

ables near real-time changes in the topology and subsequent optimisations to increase vertex locality.

Initial partitioning strategies only optimise the starting graph, with several of these techniques requiring some extent of global information. This poses significant scalability problems [42], whereas our approach relies only on local information. Additionally, as shown in Figure 1, these approaches cannot cope with changes that alter the structure of the elements already loaded (e.g. node and vertex deletion) or keeping optimal partitioning as the graph changes.

6.4 Dynamic Adaptation beyond Initial Partitioning

In addition to adapting the initial partitioning of the graph, some systems attempt to keep a small overhead when processing changing graph structures. In [42], partitioning was optimised in slowly changing graphs, with changes being applied the next time the graph was loaded. The authors employ a label propagation mechanism enhanced with geographical information to improve graph partitioning. The process involves linear programming, being computationally very expensive (reported calculations of 100 CPU days) and implying global aggregation of local (vertex-level) utility functions.

Sedge [45] is a dynamic replication mechanism (as opposed to a re-partitioning one). Sedge keeps a fixed set of non overlapping partitions and then dynamically creates new ones or replicates some of them in different machines to cope with variations in workload. Replicated systems are more focused at providing low latency to multiple concurrent and short-lived queries. Our system tries to keep a few long-lasting (continuous) queries which results are modified as a consequence of changes in the information or the topology of the graph.

7. CONCLUSIONS

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.

We believe this work contributes a first step towards the study of the performance of systems based on dynamic graphs, but there is significant work ahead. The space of dynamic graphs is still not well known, neither from models that characterise the structure and temporal dimensions of their growth, nor from deeply characterising the performance implications, finding better techniques to decide when and how to adapt to changes.

8. ACKNOWLEDGMENTS

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

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

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

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] M. Najork. The scalable hyperlink store. In *20th ACM conference on Hypertext and hypermedia*, 2009.

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

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.

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

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

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

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

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

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

[42] 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.

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

[44] 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.

[45] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, 2012.