Empowering Fast Incremental Computation over Large Scale Dynamic Graphs

Charith Wickramaarachchi
Department of Computer Science
University of Southern California
Los Angeles CA 90089 USA
cwickram@usc.edu

Charalampos Chelmis and Viktor Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles CA 90089 USA
{chelmis,prasanna}@usc.edu

***Abstract*—Unprecedented growth in online social networks, communication networks and internet of things have given birth to large volume, fast changing datasets. Data generated from such systems have an inherent graph structure in it. Updates in staggering frequencies (e.g. edges created by message exchanges in online social media) impose a fundamental requirement for real-time processing of unruly yet highly interconnected data. As a result, large-scale dynamic graph processing has become a new research frontier in computer science.**

In this paper, we present a new vertex-centric hierarchical bulk synchronous parallel model for distributed dynamic graph processing. Our model allows users to easily compose static graph algorithms similar to the widely used vertex-centric model. It also enables incremental processing of dynamic graphs by automatically executing user composed static graph algorithms in an incremental manner. We map widely used single source shortest path and connected component algorithms to this model and empirically analyze them on real-world graphs. Experimental results show that our model improves the performance of both static and dynamic graph computation compared to the vertex-centric model by reducing the global synchronization overhead.

***Keywords*-graph processing; incremental computation; bulk synchronous parallel;**

I. Introduction

There has been a lot of interest in large-scale graph processing, mainly thanks to the increasing popularity of online social networks and communication networks. The ability to mine large graphs has become critical for many real-world applications due to the fact that data generated from such systems often have a graph structure in which data elements have complex relationships among them. Irregular memory access patterns of graph algorithms and the large volume of data, make the processing of such data challenging. Particularly, in graph applications, small compute to memory access ratio has severe scalability problems as these applications hit an early "memory wall" that limits their speedup.

The major technical challenges of scaling graph algorithms and graph mining for massive datasets in the era of Big Data has given rise to high-level, high-performance programming models and graph programming frameworks. With the introduction of Map Reduce [1], numerous systems for process large-scale networked data on large commodity clusters and clouds have been proposed. Specifically, as the Map Reduce model is unsuitable for graph processing (data inter-dependencies incur increased overhead due to continuous data movement according to the graph structure from mappers to reducers), other programming models tweaked for graph algorithms have been proposed. The vertex-centric (VC), message passing model introduced by Google [2], and its open implementations including Apache Giraph [3] and Apache Hama [4] have been widely adopted, mainly for their simplicity and ease of use. Recent work on subgraph-centric programming models [5], [6], [7] have reduced the communication (number of messages) and synchronization (number of iterations required to converge to a solution) overhead of vertex-centric programming model.

Real-time graph processing is increasingly gaining momentum as the preferred method for large-scale graph processing, as many real-world applications operate on massive graphs with hundreds of millions of vertices and edges, changing attributes and evolving graph structure. The high velocity at which changes occur imposes a fundamental requirement for processing continuous graph data flows in real-time by means of *incremental processing*. Existing research has so far focused on static graph processing, while some have explored computation over a sequence of updates to static graphs. Dynamic graphs are commonly condensed into a set of snapshots of static graphs [8], [9], [2] because their static version is much easier to handle. Naively adopting a system designed for static graph processing to perform real-time incremental computation over dynamic graphs is inefficient. Even though research has been performed on large scale dynamic graph processing [10], [8], developing incremental algorithms for large-scale graphs can be a daunting programming task.

To address these challenges, we propose a graph processing model that provides efficient incremental computation over dynamic large-scale graphs, while at the same time hiding the programming complexity from developers. Our vertex-centric hierarchical bulk synchronous parallel model builds upon the strengths of Google Pregel [2] and GraphInc [11] for efficient incremental processing of massive graphs on commodity clusters. Particularly, users develop static graph algorithms based on the widely used vertex-centric programming model, which our framework conveniently and transparently converts into incremental algorithms that can be run in real-time over dynamic graphs. We show that our novel hierarchical bulk synchronous

parallel model can significantly improve the performance of vertex-centric programming abstraction by saving the state across super-steps and automatically identifying opportunities for computation reuse based on memorization [11], [12]. Specifically, our model requires minimal re-execution of computations when changes in the graph occur, thus achieving low latency on-line analysis for dynamic graph processing while at the same time offering the same programming simplicity that made vertex-centric programming prevalent. We map two widely used algorithms: connected component labeling and single source shortest path to this model. By empirically evaluating these algorithms on real-world graph datasets, we show that our model improves the performance of both static and dynamic graph computation compared to the vertex-centric model.

II. Memorization on Giraph

Cai et al proposed a technique for enabling incremental computation using vertex-centric programming model [11]. In this section we give an overview of this approach.

To understand the room for computation reuse in vertex centric programming model simple vertex centric graph algorithm with an updated graph can be used. Figure 1(b) shows the execution of single source shortest path algorithm for the sample graph shown in Figure 1(a) (using vertex 1 as the source vertex). Figure1(c) shows the re-execution of single source shortest paths algorithm after removing edge (1,5). Light dark colored vertices denote vertices that perform exactly the same computation, while dotted lines highlight repeated communication. We observe that a significant number of computations and messages are repeated when re-computing the single source shortest paths algorithm on the updated graph. Ideally, one would like to skip all repeated computation and associated communication when the graph is updated and the same analysis is to be iteratively performed.

GraphInc [11] proposed a technique to perform incremental computation using vertex-centric model for deterministic graph algorithms by reusing the state of previous graph computations. It assumes that in a vertex-centric program vertex computation at any super-step only depends on input messages and the vertex state at that point in time. Given these assumptions, GraphInc executes a static vertex-centric algorithm provided by the user in an incremental manner on an updated graph by pruning out repeated computations and communications when recomputing analysis. To avoid recomputing the analysis from scratch, GraphInc memorizes the incoming messages and state for each vertex for each super-step, and uses the memorized states to skip re-computation appropriately.

Once a graph is updated, the framework marks some vertices as affected; these vertices become candidates for re-execution on the updated graph. Affected vertices need to be potentially re-executed to get the correct results on the updated graph. The procedure for identifying affected vertices when the graph is updated is described in [11]. The framework starts the execution on the updated graph by re-computing the state of affected vertices from super-step 0. For each super-step $i > 0$ the framework decides to execute a vertex if at least one of the following conditions are satisfied: 1) At least one incoming message is different from the previous execution; 2) Vertex state is different from previous execution; 3) Vertex is marked as affected.

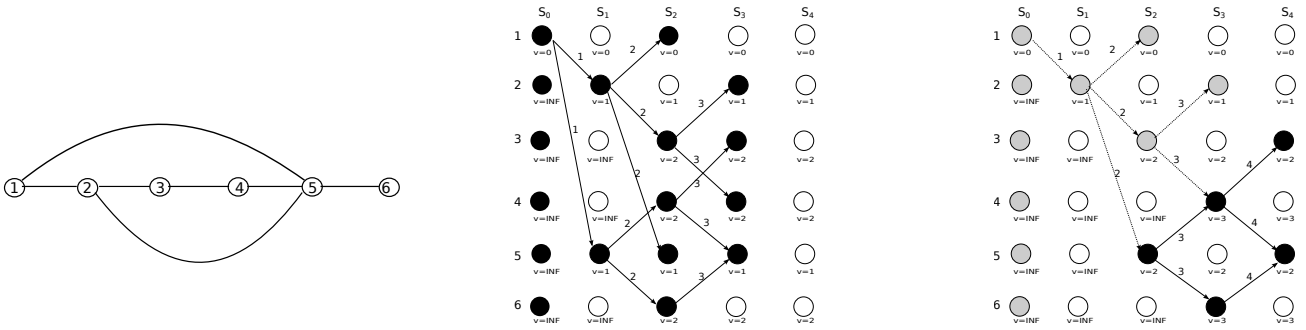
In all other cases, the framework avoids re-executing a vertex. When the state of a vertex needs to be updated, its memorized state (includes incoming messages and state at each super-step) is updated so that memorized state can be used in future computations.

III. Vertex-Centric Hierarchical Bulk Synchronous Parallel(HBSP) Model

We designed and implemented a vertex-centric hierarchical bulk synchronous parallel (HBSP) model by extending Apache Giraph software framework. In this model, BSP executions happen at two levels. 1) Partition level (Local computation) 2) Cluster level (Global computation). Initially, the graph is partitioned, and each partition is assigned to a worker machine in the cluster in the data loading phase. Then, the vertex-centric program provided by the user executes within each partition locally following the vertex-centric BSP model. In this step messages sent to vertices in other partitions (remote vertices) are buffered so that they can be sent once the local computation is completed. The global computation phase starts once the local computation phase finishes its computations. In a global computation step, each worker communicates with each other using the buffered messages from the previous local computation step. A global barrier synchronization step follows. Once all workers finish communicating, local computation starts again within each partition using the messages received during the global computation step. These two BSP stages are continued until all vertices vote to halt, similar to the vertex-centric programming model, i.e, all vertices are inactive with no incoming messages to process.

This model can be thought of as an extension to the sub-graph/partition centric models proposed in [5], [6] where local computation within the partition is executed using vertex centric model. Vertices within partitions are executed in parallel using the multiple cores in each worker machine. Each core is responsible for executing a subset of vertices in a graph partition. Users are also provided with a programming abstraction to reduce the number of messages communicated in global computation step by performing summarization when possible (similar to combiners in the vertex-centric model). We call iterations in the local computation step *sub-super-steps* while iterations in global level *super-steps*.

To demonstrate the effectiveness of our HBSP model, we provide a sample application that finds the maximum value in a connected graph. Algorithm 1 presents the algorithm. Figure 2 shows the execution of this algorithm on a simple graph. We note that only two super-steps are required to complete the algorithm. For reference, the traditional vertex-centric algorithm requires four super-steps. This translates to



(a) (b) (c)

Figure 1: Figure (a) shows the initial graph that we want to find single source shortest path from source vertex 1. Figure (b) shows the execution steps of vertex centric single source shortest path algorithm on the initial graph and Figure (c) shows the execution of vertex centric single source shortest path algorithm on the updated graph after removing edge (1,5) from the initial graph

a 50% reduction in the number of supersteps in this simple example.

We further extended our HBSP model to support memorization by extending the technique described in Section II. In our model we try to avoid re-computation both at partition and vertex level using memorized states kept for each super-step and sub-super-step. This approach not only reduces the number of super-steps required for incremental computation compared to vertex centric model, but also enables the pruning of computation both at partition and vertex levels which can potentially reduce the added overhead (computation time required to process memorized state before pruning out re-computations) imposed by memoization.

Algorithm 1 Max Vertex Using HBSP

```
1:  procedure COMPUTE(Vertex v, Iterator<Messages> msgs)
2:      if super-step == 0 and sub-super-step == 0 then
3:          BROADCASTGREATESTNEIGHBOR(v)        ▷ Find the greatest vertex id m from the neighborhood set (including self), set m as the current value, and sent it to all neighbors
4:          return
5:      end if
6:      changed ← false
7:      maxId ← v.value
8:      while msgs.hasNext do
9:          m = msgs.next
10:         if maxId < m.value then
11:             maxId ← m.value
12:             change ← true
13:         end if
14:     end while
15:     if changed then
16:         v.value ← maxId
17:         BROADCASTUPDATE(v)    ▷ Send the vertex value to all neighbors of v
18:     end if
19: end procedure
```

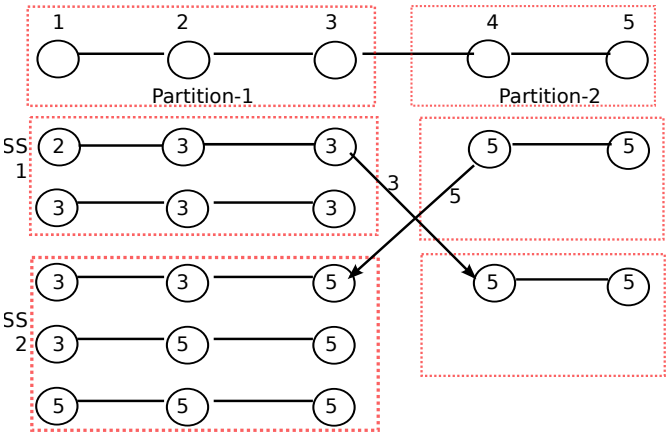


Figure 2: Execution of Alg 1 in HBSP model.

IV. Experimental Results

A. Implementation

We implemented our HBSP model by extending the latest released version of Apache Giraph [1] [3] (1.1.0). In-memory data-structures (semaphores) were used to implement local barriers. In our implementation, all internal communication between vertices within partitions are performed using in-memory data structures. During local computation process each machine initially assigns a fix number of threads equal to the number of processors in the system, for vertex processing. Vertices are assigned to processors at the start of each super-step. To avoid unbalanced execution, we have implemented a work-stealing mechanism to re-balance the work across workers in the presence of stranglers.

Our implementation allows users to use any partitioning strategy when loading initial partitions. This was implemented by extending the Mapping Store feature [2] of Apache Giraph. Users can this way assign vertices to partitions; this informa-

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

[2] https://issues.apache.org/jira/browse/GIRAPH-908

tion is used for mapping vertices to worker machines in the cluster.

We implemented both our HBSP model and the vertex-centric model with memoization, since the original implementation of GraphInc [11] is not publicly available. Memorized states were stored in-memory data structures at the partition level. While we implementing the core functionality of the aforementioned models we refrained from performing low level engineering optimizations. Since such optimizations can play a major role in the overall system performance of a runtime system, we avoid reporting execution time as a representative metric in our experiments for fair comparison.

B. Experimental Setup

We conducted a series of experiments to evaluate the advantage of our approach for both static and dynamic graph computation. All experiments were conducted in an cluster of 15 nodes. Each node consists of 8-core Intel Xeon CPU with 16GB RAM. All Giraph jobs were executed on 12 workers. 14GB of RAM was allocated per each worker. All the applications were executed on Java 7 runtime environment on 64 bit Linux environment (CentOS).

Two real-world datasets from the Stanford Large Network Dataset Collection [13] were used in our experiments: (i) California road network and (ii) Slashdot social network from 2009. Table I summarizes the number of vertices and edges in each dataset. We used two applications for evaluation purposes: (i) Connected component (CC) labeling (same as Algorithm 1) and (ii) Single Source Shortest Paths (SSSP) (See Algorithm 2).We experimented with two partitioning strategies for our HBSP model. Specifically, we used (i) a random vertex assignment strategy and (ii) Metis graph partitioning tool [14] to partition the graph during pre-processing.

Algorithm 2 SSSP Using HBSP

```
1: procedure COMPUTE(Vertex v, Iterator<Messages> msgs)
2:     if super-step == 0 and sub-super-step == 0 then
3:         v.value ← +inf
4:     end if
5:     minDist = IS_SOURCE(v) ? 0 : +inf;
6:     while msgs.hasNext do
7:         m ← msgs.next
8:         if midDist > m.value then
9:             midDist ← m.value
10:        end if
11:    end while
12:    if minDist < v.value then
13:        v.value ← minDist
14:        BROADCASTDISTANCE(v)   ▷ Send the distance through this vertex to all its neighbours
15:    end if
16: end procedure
```

To evaluate the impact of HBSP model on memorization, we generated two sets of updated graphs for each data set, for VC and HBSP models by adding 100 random edges and deleting 30 random edges from each data set. Same applications (CC and SSSP) were executed incrementally on updated graphs using memorization (see Section II. We logged the number of vertices executed when re-computing without memorization (r_e) and when using memorization (m_e). We then calculated the fraction of computations saved as $\frac{r_e - m_e}{r_e}$.

Dataset	# Vertices	# Edges
SlashDot (SD)	82,168	948,464
Road Network - CA (RN)	1,965,206	2,766,607

Table I: High level statistics of the two datasets used for evaluation.

C. Results and Analysis

As explain in Section III, our HBSP model can improve the performance of traditional vertex centric model by reducing the number of global synchronization steps. We compared vertex centric model (VC) with our model using random (HBSP-R) and Metis (HBSP-M) partitioning schemes. As shown in Figures 3 and 4 a reduction in number of super-steps when using the HBSP model can be observed. The number of super-steps required to converge to a solution were reduced drastically when Metis partitioning scheme was used. A significant difference in the number of super-steps required for RN and SD datasets can also be observed. This is mainly due to the difference in the diameter of two graphs; RN has a large diameter compared to SD network which exhibits small-world characteristics. As a result, both applications take large number of super-steps in VC model for converging to a solution on RN dataset. Contrarily, the number of super-steps is significantly reduced when Metis partitioning scheme is employed. This signifies the importance of graph partitioning schemes for partition-centric graph computation models.

As shown in Figures 6 and 7, similar reduction in the number of super-steps can be observed when HBSP model is used in conjunction to memorization. Also, our experimental results (Figure 5) suggest that HBSP model does not drastically reduce the number of saved computations when used for incremental computation. Given the above observations we conclude that vertex-centric memorization model benefits from our HBSP model, and building upon the strengths of subgraph-centric computing , significantly improves the performance of static graph computation and more importantly that of incremental computation over dynamic graphs.

V. Related Work

Large scale dynamic graph processing has recently become a very active research area in computer science. Several systems has been proposed and presented for large scale dynamic graph processing in last few years. We summarize here the most relevant to our work.

STINGER[3] focuses on large-scale dynamic graph processing on massively multi-threaded shared-memory machines where our work focus on distributed cluster environments. It provides a shared memory data structure [10] for large

[3]http://www.stingergraph.com/

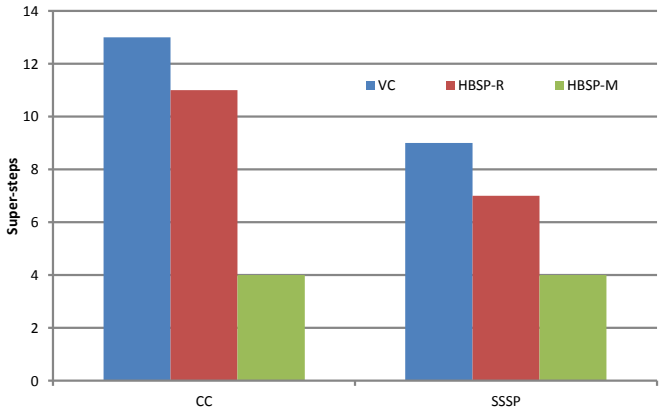


Figure 3: Super step comparison for CC and SSSP applications on Slashdot social network dataset

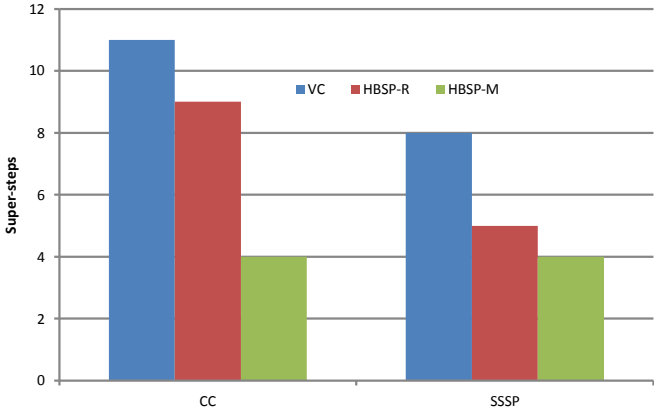


Figure 6: Super step comparison for CC and SSSP applications using memorization on Slashdot dataset

Figure 4: Super step comparison for CC and SSSP applications on California road network dataset

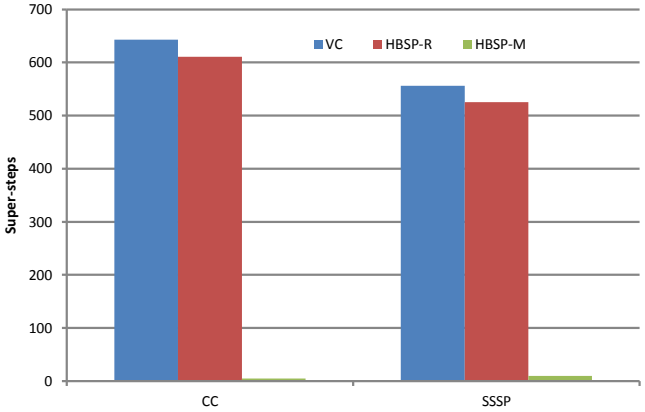


Figure 7: Super step comparison for CC and SSSP applications using memorization on California road network dataset

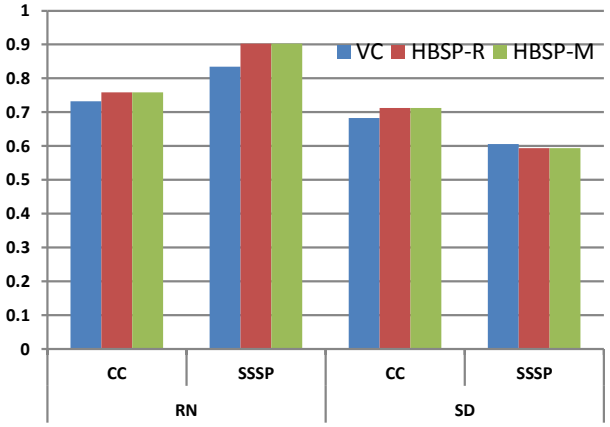


Figure 5: Comparison of fraction of computations saved for CC and SSSP applications on California road network dataset (RN) and Slashdot dataset (SN)

dynamic graph processing. A series of dynamic graph algorithms have been developed using this data structure [15], [16]. However, developing dynamic graph algorithms using STINGER can be non trivial, requiring significant skills in algorithmic design and programming. Similarly, Cheng et al presented a system (Kineograph) for real-time dynamic graph analysis [8]. While Kineograph enables online incremental computation on fast changing dynamic graphs, Kineograph's programming abstraction leaves to users the responsibility to develop incremental graph algorithms, which can be a non-trivial task. Instead, our HBSP model takes the burden of developing dynamic graph algorithms while at the same time offering a simple programming abstraction, much similar to the widely adopted vertex-centric programming model.

Simmhan et al [9] presented a set of programming patterns that can be used for distributed processing of time series graphs. This work mainly focused on processing series of snapshots of temporal graphs that are stored in the disk. Unlike our work, their programming model does not provide native support for incremental graph computation. Since our approach can be used to perform incremental computation on graph snapshots, we believe that combining our approach with these proposed programming models can be used to enable low latency analysis over time series graphs. We intent to explore

this lead in future work.

Cai et al. [11] exploited memorization for incremental graph computation (GraphInc) based on the vertex-centric model. In our paper, we showed that our hierarchical BSP model, when augmented with memorization can significantly outperform GraphInc. While memorization is applicable to recently proposed sub-graph or partition-centric models presented in [5], [6], [7], since these models do not exert control at the vertex level, fine graph computation reuse cannot be achieved.

VI. Conclusion and Future Work

We introduced a vertex-centric hierarchical bulk synchronous parallel model for distributed incremental graph computation. While keeping the simplicity and scalability of widely used vertex-centric model, our approach can be used to improve the performance of vertex-centric model by reducing its global synchronization overhead. Using a proof of concept system implementation on Apache Giraph, we empirically showed that our model improves the performance of both static and dynamic graph computation, reducing the global synchronization overhead by up to 128x for connected component algorithm and up to 55x for single source shortest path algorithm.

One major issue with memorization [11] is the overhead of additional computation power required to prune computations. We believe that memorization model is much suitable when per vertex computation is comparatively larger than the computation overhead of memorization. This opens up space for new future research directions including strategies to enable bulk pruning strategies which can identify maximum computation reuse opportunities with less additional computation overhead.

Our experimental results shows that graph partitioning plays major role when it comes to performance. This observation is consistent with slimier observations reported in other studies [5]. A major research challenge is to come up with dynamic graph partitioning techniques to maintain work balance between workers while keeping highly modular partitions. In a real-world online environment where graph is changing fast, dynamic graph partitioning schemes must be implemented in order to maintain performance benefits of our approach.

We plan to further evaluate this model on different types of graphs and partitioning schemes to better understand performance behavior in the future. Our proof of concept implementation gives us further opportunities to peruse those future research directions.

Acknowledgment

This work was partially supported by a the US NSF under grand NSF:1355377 and a research grant from the DARPA XDATA grant no. FA8750-12-2-0319. Authors would like to thank Alok Kumbhare for his feedback.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[3] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011. [Online]. Available: http://giraph.apache.org/

[4] 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 *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 721–726.

[5] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," *arXiv preprint arXiv:1311.5949*, 2013.

[6] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, 2013.

[7] A. Quamar, A. Deshpande, and J. Lin, "Nscale: Neighborhood-centric analytics on large graphs,," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, 2014.

[8] 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 *ACM european conference on Computer Systems (EuroSys)*, 2012.

[9] Y. Simmhan, C. Wickramaarachchi, A. G. Kumbhare, M. Frîncu, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna, "Scalable analytics over distributed time-series graphs using goffish," *CoRR*, vol. abs/1406.5975, 2014. [Online]. Available: http://arxiv.org/abs/1406.5975

[10] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 2012, pp. 1–5.

[11] Z. Cai, D. Logothetis, and G. Siganos, "Facilitating real-time graph mining," in *Proceedings of the fourth international workshop on Cloud data management*. ACM, 2012, pp. 1–8.

[12] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 7:1–7:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038923

[13] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[14] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," University of Minnesota, Tech. Rep., 1995.

[15] D. Ediger, R. McColl, J. Poovey, and D. Campbell, "Scalable infrastructures for data in motion," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 875–882.

[16] D. E. S. A. E. Briscoe and R. M. J. Poovey, "Real-time streaming intelligence: Integrating graph and nlp analytics."