BLADYG: A Novel Block-Centric Framework for the Analysis of Large Dynamic Graphs

Sabeur Aridhi*
Aalto University, School of Science, P.O. Box 12200, FI-00076, Finland
sabeur.aridhi@aalto.fi

Alberto Montresor
University of Trento, Italy
alberto.montresor@unitn.it

Yannis Velegrakis
University of Trento, Italy
velgias@disi.unitn.eu

ABSTRACT

Recently, distributed processing of large dynamic graphs has become very popular, especially in certain domains such as social network analysis, Web graph analysis and spatial network analysis. In this context, many distributed/parallel graph processing systems have been proposed, such as Pregel, GraphLab, and Trinity. These systems can be divided into two categories: (1) vertex-centric and (2) block-centric approaches. In vertex-centric approaches, each vertex corresponds to a process, and message are exchanged among vertices. In block-centric approaches, the unit of computation is a block, a connected subgraph of the graph, and message exchanges occur among blocks. In this paper, we are considering the issues of scale and dynamism in the case of block-centric approaches. We present BLADYG, a block-centric framework that addresses the issue of dynamism in large-scale graphs. We present an implementation of BLADYG on top of AKKA framework. We experimentally evaluate the performance of the proposed framework.

Keywords

Distributed graph processing, Dynamic graphs, AKKA framework

1. INTRODUCTION

In the last decade, the field of distributed processing of large-scale graphs has attracted considerable attention. This attention has been motivated not only by the increasing size of graph data, but also by its huge number of applications, such as the analysis of social networks [4], web graphs [2] and spatial networks [10]. In this context, many distributed/parallel graph processing systems have been proposed, such as Pregel [8], GraphLab [7], and Trinity [12]. These systems can be divided into two categories: (1) vertex-centric and (2) block-centric approaches. Vertex-centric approaches divide input graphs into partitions, and employ a "think like a vertex" programming model to support iterative graph computation [8, 13]. Each vertex corresponds to a process, and message are exchanged among vertices. In block-centric approaches [16], the unit of computation is a block – a connected subgraph of the graph – and message exchanges occur among blocks.

In our work, we are considering the issues of scale and dynamism in the case of block-centric approaches. Particularly, we are considering big graphs known by their evolving and decentralized nature. For example, the structure of a big social network (e.g., Twitter, Facebook) changes over time (e.g., users start new relationships and communicate with different friends).

We present BLADYG, a block-centric framework that addresses the issue of dynamism in large-scale graphs. BLADYG can be used not only to compute common properties of large graphs, but also to maintain the computed properties when new edges and nodes are added or removed. The key idea is to avoid the re-computation of graph properties from scratch when the graph is updated. BLADYG limits the re-computation to a small subgraph depending on the undertaken task. We present a set of abstractions for BLADYG that can be used to design algorithms for any distributed graph task.

More specifically, our contributions are: (i) we introduce BLADYG and its computational distributed model; (ii) we present an implementation of BLADYG on top of AKKA [14], a framework for building highly concurrent, distributed, and resilient message-driven applications; (iii) we experimentally evaluate the performance of the proposed framework, by applying it to the example problem of distributed k-core decomposition of large graphs.

The rest of the paper is organized as follows. We present BLADYG in the following section. Sec. 3 presents some research problems that can be solved using BLADYG. Finally, we describe our experimental evaluation in Sec. .

*Work primarily done while the author was at the University of Trento.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPGP'16, May 31, 2016, Kyoto, Japan
© 2016 ACM. ISBN 978-1-4503-4350-3/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2915516.2915525

2. THE BLADYG FRAMEWORK

Figure 1 provides an architectural overview of the BLADYG framework. BLADYG starts its computation by partitioning the input graph into multiple partitions, each of them assigned to a different worker. Each partition/block is a connected subgraph of the input graph. This partitioning step is performed by a partitioner worker that supports several types of predefined partitioning techniques. BLADYG

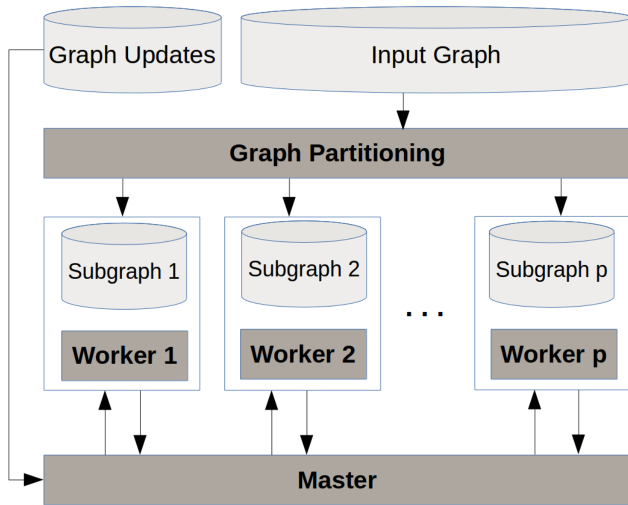

Figure 1: BLADYG system overview

users may also implement their partitioning methods. It is important to mention that BLADYG allows to process large graphs that already distributed among a set of machines. This is motivated by the fact that the majority of the existing large graphs are already stored in a distributed way, either because they cannot be stored on a single machine due to their sheer size, or because they get processed and analyzed with decentralized techniques that require them to be distributed among a collection of machines. Each worker loads its block and performs both local and remote computations, after which the status of the blocks is updated. The master worked orchestrates the execution of BLADYG in order to deal with incremental changes on the input data. Depending on the graph task, the coordinator builds an execution plan which consists of an ordered list of both local and distant computation to be executed by the workers.

Each worker performs two types of operations:

1. **Intra-block computation**: in this case, the worker do local computation on its associated block (partition) and modifies the status of the block.

2. **Inter-block computation**: in this case, the worker asks distant workers to do computation and after receiving the results it updates the status of its associated block.

BLADYG framework for large dynamic graph analysis operates in three computing modes:

- In *M2W-mode*, message exchanges between the master and all workers are allowed. The master uses this mode to ask a distant worker to look for candidate nodes i.e., nodes that need to be updated depending on the undertaken task. The worker uses this mode to send the set of computed candidate nodes to the master.

- In *W2W-mode*, message exchanges between workers are allowed. The workers use this mode in order to propagate the search for candidate nodes to one or more distant workers.

- In *Local-mode*, only local computation is allowed. This mode is used by the worker/master to do local computation.

A typical BLADYG computation consists of an input graph, a set of incremental changes, a sequence of worker/master operations and an output. The input of BLADYG framework is an undirected graph. Each vertex is uniquely identified by a vertex identifier and each edge is identified by a unique edge identifier and its associated vertices. Incremental changes or graph updates consists of edge/node insertions and/or removals. A worker operation is a user-defined function that is executed by one or many workers in parallel depending on the logic of the graph task. Within each worker operation, the state of the associated block is updated and all the computing modes of BLADYG are activated. Within each master operation, a user defined function that defines the orchestration mechanism of the master is executed. During a master operation *Local-mode* and *M2W-mode* are activated.

3. APPLICATIONS

In this section, we apply BLADYG to solve some classic graph operations such as k–core decomposition [9] and clique computation [15].

3.1 Distributed k-core decomposition

Let $G = (V, E)$ be an undirected graph with $n = |V|$ nodes and $m = |E|$ edges. G is partitioned into p disjoint partitions $\{V_1, \ldots, V_p\}$; in other words, $V = \cup_{i=1}^{p} V_i$ and $V_i \cap V_j = \emptyset$ for each i, j such that $1 \leq i, j \leq p$ and $i \neq j$. The task of *k–core decomposition* [3] is condensed in the following two definitions:

DEFINITION 1. *A subgraph $G(C)$ induced by the set $C \subseteq V$ is a k-core if and only if $\forall u \in C : d_{G(C)}(u) \geq k$, and $G(C)$ is maximal, i.e., for each $\overline{C} \supset C$, there exists $v \in \overline{C}$ such that $d_{G(\overline{C})}(v) < k$.*

DEFINITION 2. *A node in G is said to have* coreness k *($k_G(u) = k$) if and only if it belongs to the k-core but not the $(k+1)$-core.*

A k-core of a graph $G = (V, E)$ can be obtained by recursively removing all the vertices of degree less than k, until all vertices in the remaining graph have degree at least k. The issue of distributed *k–core decomposition* in dynamic graphs consists in updating the coreness of the nodes of G when new nodes/edges are added and/or removed.

BLADYG solves the problem of distributed *k–core decomposition* in two steps. The first step consists in executing a workerCompute() operation that computes the coreness inside each of the blocks. Inside a block, each vertex is associated with $block(u)$, $d_G(u)$ and $k_G(u)$, denoting the block of u, the degree and the coreness of u in G, respectively. The second step consists in maintaining the coreness values after considering the incremental changes. Whenever a new edge (u, v) is added to the graph, BLADYG first activates the *M2W-mode* and computes the set of candidate nodes i.e., nodes whose coreness needs to be updated. This is done by two workerCompute() operations inside the workers that hold u and v. The workerCompute() operations exploit Theorem 1, first stated and demonstrated by Li, Yu and Mao [6], that identifies what are the *candidate nodes* that may need to be updated whenever we add an edge:

THEOREM 1. *Let $G = (V, E)$ be a graph and (u, v) be an edge to be inserted in E, with $u, v \in V$. A node $w \in V$ is said to be a* candidate *to be updated based on the following three cases:*

- *If $k(u) > k(v)$, w is candidate if and only if w is k-reachable from u in the original graph G and $k = k(u)$;*
- *If $k(u) < k(v)$, w is candidate if and only if w is k-reachable from v in the original graph G and $k = k(v)$;*
- *If $k(u) = k(v)$, w is candidate if and only if w is k-reachable from either u and v in the original graph G and $k = k(u)$.*

A node w is *k-reachable* from u if w is reachable from u in the k-core of G; i.e., if there exists a path between u and w in the original graph such that all nodes in the path (including u and w) have coreness equal to $k = k(u)$.

We notice that the executed workerCompute() operations may activate the *W2W-mode* since the set of nodes to be updated may span multiple blocks/partitions. The nodes identified as potential candidates are sent back to the coordinator node that orchestrates the execution and computes, by executing a masterCompute() operation, the correct coreness values of the candidate nodes.

3.2 Distributed maximal clique computation

Given an undirected graph $G = (V, E)$, a clique is a subset of vertices $C \subseteq V$ such that every vertex in C is connected to every other vertex in C by an edge in G. A clique C is called to be maximal if any proper superset of C is not a clique. The problem of maximal clique enumeration (MCE) is to compute the set $M(G)$ of maximal cliques in G. Considering the issue of dynamism, the problem of MCE in dynamic graphs [15] consists in incrementally update the set of maximal cliques for every graph update.

BLADYG deals with the problem of MCE in dynamic graphs in the following way. Each edge of each block maintains $ID(v)$, $adj(u)$, M_u and T_u, which denote the identifier of u, the adjacent vertices of u, the set of maximal cliques of u and a prefix-tree such that the root of T_u is u and each root-to-leaf path represents a maximal clique in M_u, respectively. We assume that adjacency list representation of the graph G, the set V of vertices are ordered in ascending order of their IDs. We further define $adj_<(u) = \{v : v \in adj(u), ID(v) < ID(u)\}$ and $adj_>(u) = \{v : v \in adj(u), ID(v) > ID(u)\}$. When an edge (u, v) is inserted into G, BLADYG coordinator asks workers containing u and v to update the set of maximal cliques. Each of the workers of u and v executes a workerCompute() operation in order to remove existing maximal cliques that become non-maximal and insert maximal cliques that should be inserted. An existing maximal clique C becomes non-maximal if C contains either u or v, and verifies $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$ [15]. Maximal cliques that need to be added to the existing ones consists of new maximal cliques that contain u, v, w, for each $w \in ((adj_<(u) \cap adj_<(v)) \cup \{u\})$ [15]. When an edge (u, v) is deleted from G, BLADYG coordinator notifies workers containing the nodes u and v by the edge deletion. Workers that hold u and v execute a workerCompute()operation that deletes all the existing maximal cliques that contain both u and v, where such maximal cliques appear in T_w, where $w \in ((adj_<(u) \cap adj_<(v)) \cup \{u\})$ [15]. Then, we generate all new maximal cliques that contain only u or v, and insert them into T_w, where $w \in ((adj_>(u) \cap adj_<(v)) \cup \{v\})$ or $w \in ((adj_<(u) \cap adj_<(v)) \cup \{u\})$. A notification is sent to BLADYG coordinator when all the workers finish the update process.

Table 1: Experimental data

Dataset	Type	♯ Nodes	♯ Edges	∅	Avg. CC	Max(k)
DS1	Synthetic	50,000	365,883	4	0.3929	42
DS2	Synthetic	100,000	734,416	4	0.3908	46
ego-Facebook	Real	4,039	88,234	8	0.6055	115
roadNet-CA	Real	1,965,206	2,766,607	849	0.0464	3
com-LiveJournal	Real	3,997,962	34,681,189	17	0.2843	296

Table 2: Experimental results

Dataset	AIT (ms)		ADT (ms)	
	inter-partition	intra-partition	inter-partition	intra-partition
DS1	42	10	32	8
DS2	30	10	25	8
ego-Facebook	38	15	32	10
roadNet-CA	30	12	26	10
com-LiveJournal	256	30	205	27

4. EXPERIMENTS

We have applied BLADYG framework to the problem of distributed k-core decomposition in large dynamic graphs. We have performed a set of experiments to evaluate the effectiveness and efficiency of BLADYG framework on a number of different real and synthetic datasets.

Since the goal is to compute k-core decomposition, the characteristic properties of our datasets (shown in Table 1) are the number of nodes, edges, the diameter, the average clustering coefficient and the maximum coreness. We have used two groups of datasets: real-world ones, made available by the Stanford Large Network Dataset collection [5], and synthetic datasets, created by a graph generator based on the Nearest Neighbor model [11].

We have implemented BLADYG on top of the AKKA framework, a toolkit and runtime for building highly concurrent, distributed, resilient message-driven applications. In order to evaluate the performance of BLADYG, we used a cluster of 17 `m3.medium` instances on Amazon EC2 (1 virtual 64-bit CPU, 3.75GB of main memory, 8GB local instance storage).

In order to simulate dynamism in each dataset, we consider two update scenarios. For each scenario, we measure the performance of the system to update the core numbers of all the nodes in the considered graph after insertion/deletion of a constant number of edges:

- In the *inter-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *different* partitions;
- In the *intra-partition* scenario, we either delete or insert 1000 random edges connecting two nodes belonging to *the same* partition.

Table 2 illustrates the results obtained with both the real and the synthetic datasets. For each dataset, we record the average insertion time (AIT) and the average deletion time (ADT) over the 1000 insertions/deletions for both *inter-partition* and *intra-partition* scenarios. To generate the results of Table 2, we randomly partition the graph dataset into 8 partitions. As shown in Table 2, we observe that in the *intra-partition* scenario, the values of the average insertion/deletion time are much smaller than those in the *inter-*

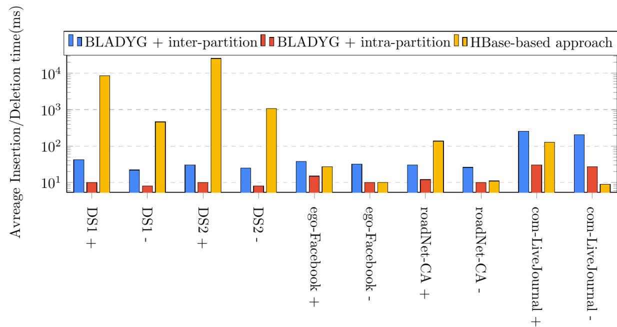

Figure 2: Average insertion/deletion time

partition scenario. This can be explained by the fact that the inserted/deleted edges in the *intra-partition* scenario are internal ones. Consequently, the amount of data to be exchanged between the distributed machines in the case of internal edges is smaller, in most cases, than the amount of exchanged data in the case of edges of the *inter-partition* scenario. During the k-core maintenance process after insertion/deletion of an internal edge, there is always the chance of not having to visit distributed workers/partitions other than the partition that holds the internal edge.

Figure 2 presents a comparison of our BLADYG solution with the HBase-based approach proposed by Aksu et al. [1] in terms of average insertion/deletion time. For our approach, we used 9 `m3.medium` instances on Amazon EC2 (1 acting as a master and 8 acting as workers). For the HBase-based approach, we used 9 `m3.medium` instances on Amazon EC2 (1 master node and 8 slave nodes). As stressed in Figure 2, our approach allows much better results compared to the HBase-based approach for almost all datasets. It is noteworthy to mention that the presented runtime values of the HBase-based approach correspond to the maintenance time of only one fixed k value core ($k = max(k)$ in our experimental study). This means that, for each dataset, the maintenance process of the HBase-based approach needs to be repeated $max(k)$ times in order to achieve the same results as our approach.

5. CONCLUSIONS

This paper deal with the problem of graph processing in large dynamic networks. We presented BLADYG framework, a block-centric framework that addresses the issue of dynamism in large scale graphs. The presented framework can be used not only to compute common properties of large graphs but also to maintain the computed properties when new edges and nodes are added or removed. We implemented BLADYG on top of AKKA, a framework for building highly concurrent, distributed, and resilient message-driven applications. We applied BLADYG to the problem of distributed k-core decomposition in large dynamic graphs. By running some experiments on a variety of both real and synthetic datasets, we have shown that the performance and scalability of the proposed framework are satisfying for large-scale graphs.

In the future work, we aim at studying data communications, networking and scalability of BLADYG framework with respect to the number of distributed machines.

6. REFERENCES

[1] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and O. Ulusoy. Distributed k -core view materialization and maintenance for large dynamic graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 26(10):2439–2452, Oct 2014.

[2] J. I. Alvarez-Hamelin, A. Barrat, A. Vespignani, and et al. k-core decomposition of internet graphs: hierarchies, self-similarity and measurement biases. *Networks and Heterogeneous Media*, 3(2):371, 2008.

[3] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

[4] C. Giatsidis, D. Thilikos, and M. Vazirgiannis. Evaluating cooperation in communities with the k-core structure. In *Proc. of the Int. Conf. on Advances in Social Networks Analysis and Mining*, July 2011.

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

[6] R. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014.

[7] Y. Low, D. Bickson, J. Gonzalez, and et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[8] G. Malewicz, M. H. Austern, A. J. Bik, and et al. Pregel: A system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146. ACM, 2010.

[9] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.

[10] R. Patuelli, A. Reggiani, P. Nijkamp, and F.-J. Bade. The evolution of the commuting network in Germany: Spatial and connectivity patterns. *Journal of Transport and Land Use*, 2(3), 2010.

[11] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proc. of the 19th Int. Conf. on World Wide Web (WWW'10)*. ACM, 2010.

[12] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proc. of the Int. Conf. on Management of Data*. ACM, 2013.

[13] Y. Tian, A. Balmin, S. A. Corsten, and et al. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, 2013.

[14] D. Wyatt. *Akka Concurrency*. Artima Inc., 2013.

[15] Y. Xu, J. Cheng, A. W. Fu, and Y. Bu. Distributed maximal clique computation. In *Proc. of the IEEE Int. Congress on Big Data*, pages 160–167, 2014.

[16] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, Oct. 2014.