

# BLADYG: A Graph Processing Framework for Large Dynamic Graphs

Sabeur Aridhi<sup>a,\*</sup>, Alberto Montresor<sup>b</sup>, Yannis Velegrakis<sup>b</sup>

<sup>a</sup>University of Lorraine, LORIA, Campus Scientifique, BP 239, 54506  
Vandoeuvre-lès-Nancy, France

<sup>b</sup>University of Trento, Italy

---

## 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 [5]. 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 [8], web graphs [2] and spatial networks [18]. In this context, many distributed/parallel graph processing systems have been proposed, such as Pregel [16], GraphLab [15], and Trinity [23]. These systems can be divided into two categories: (1) vertex-centric and (2) block-centric approaches. Vertex-centric approaches divide input graphs into

---

\*Corresponding author

Email addresses: [sabeur.aridhi@loria.fr](mailto:sabeur.aridhi@loria.fr) (Sabeur Aridhi),  
[alberto.montresor@unitn.it](mailto:alberto.montresor@unitn.it) (Alberto Montresor), [velgias@disi.unitn.eu](mailto:velgias@disi.unitn.eu) (Yannis Velegrakis)

partitions, and employ a "think like a vertex" programming model to support iterative graph computation [16, 25]. Each vertex corresponds to a process, and message are exchanged among vertices. In block-centric approaches [29], the unit of computation is a block – a connected subgraph of the graph – and message exchanges occur among blocks. The vertex-centric approaches have been proved to be useful for many graph algorithms. However, they do not always perform efficiently, because they ignore the vital information about graph partitions, which represent a real subgraph of the original input graph, instead of a collection of unrelated vertices.

In our work, we are considering the issues of scale and dynamism in the case of block-centric approaches [4]. 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:

- We introduce BLADYG and its computational distributed model.
- We present an implementation of BLADYG on top of AKKA [26], a framework for building highly concurrent, distributed, and resilient message-driven applications.
- We experimentally evaluate the performance of the proposed framework, by applying it to problems such as distributed  $k$ -core decomposition of large graphs and partitioning of large dynamic graphs.

The rest of the paper is organized as follows. In Section 2, we highlight existing works on distributed graph processing on large and dynamic graphs. In Section 3.1, we present the system overview of BLADYG. In Section 4, we present some research problems that can be solved using BLADYG. Finally, we describe our experimental evaluation in Section 5.

## 2. Related works

In this section we highlight the relevant literature in the field of large graph processing. We consider two kinds of frameworks: (1) graph processing frameworks and (2) frameworks for the processing of large and dynamic graphs.

*Graph processing frameworks.* Pregel [16] is a computational model for large-scale graph processing problems. In Pregel, message exchanges occur among vertices of the input graph. As shown in Figure 1), each vertex is associated to a state that controls its activity.

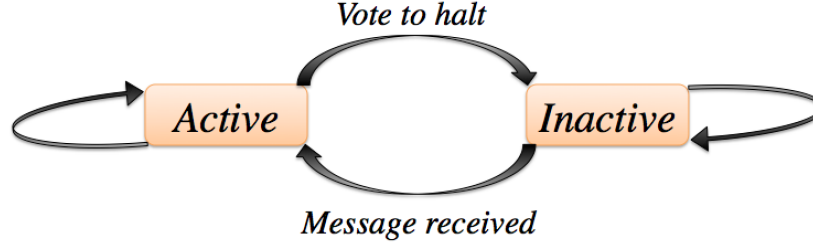


Figure 1: Vertex's state machine in Pregel.

Each vertex can decide to halt its computation, but can be woken up at every point of the execution by an incoming message. At each superstep of the computation a user defined vertex program is executed for each active vertex. The user defined function will take the vertex and its incoming messages as input, change the vertex value and eventually send messages to other vertices through the outgoing edges.

GraphLab [15] is a graph processing framework that share the same motivation with Pregel. While pregel targets Google's large distributed system, GraphLab addresses shared memory parallel systems which means that there is more focus on parallel access of memory than on the issue of efficient message passing and synchronization. In the programming model of GraphLab, the users define an update function that can change all data associated to the scope of that node (its edges or its neighbors). Figure 2 shows the scope of a vertex: an update function called on that vertex will be able to read and write all data in its scope. We notice that that scopes can overlap, so simultaneously executing two update functions can result in a collision. In this context, GraphLab offers some consistency models in order to allow its users to trade off performance and consistency as appropriate for their computation. As described in Figure 2, GraphLab offers a fully consistent model, a vertex consistent model or an edge consistent model.

Powergraph [9] is an abstraction that exploits the structure of vertex-programs and explicitly factors computation over edges instead of vertices. It uses a greedy approach, processing and assigning each edge before moving to the next. It keeps in memory the current sizes of each partition and, for each vertex, the set of partitions that contain at least one edge of that vertex. If both endpoints of the current edge are already inside one common partition, the edge will be added to that partition. If they have no partition in common, the node with the most edges still to assign will choose one of its partitions. If only one node is already in a partition, the edge will be assigned to that partition. Otherwise, if both nodes are free, the edge will be assigned to the smallest partition.

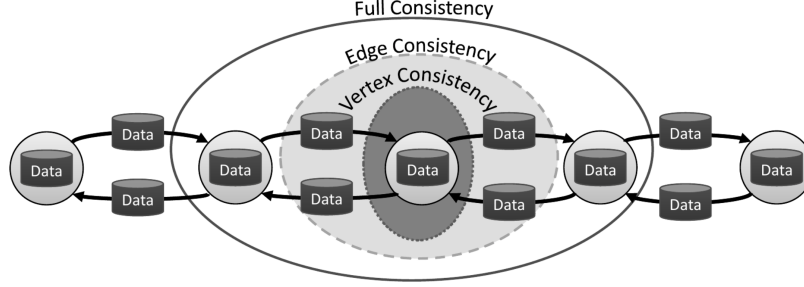


Figure 2: View of the scope of a vertex in GraphLab.

GraphX [27] is a library provided by Spark [30], a framework for distributed and parallel programming. Spark introduces Resilient Distributed Datasets (RDD), that can be split in partitions and kept in memory by the machines of the cluster that is running the system. These RDD can be then passed to one of the predefined meta-functions such as map, reduce, filter or join, that will process them and return a new RDD. In GraphX, graphs are defined as a pair of two specialized RDD. The first one contains data related to vertices and the second one contains data related to edges of the graph. New operations are then defined on these RDD, to allow to map vertices’s values via user defined functions, join them with the edge table or external RDDs, or also run iterative computation.

*Processing of large and dynamic graphs.* Chronos [11] is an execution and storage engine designed for running in-memory iterative graph computation on evolving graphs. Locality is an important aspect of Chronos, where the in-memory layout of temporal graphs and the scheduling of the iterative computation on temporal and evolving graphs are carefully designed. The design of Chronos further explores the interesting interplay among locality, parallelism, and incremental computation in supporting common mining tasks on temporal graphs. We notice that traditional graph processing frameworks arrange computation around each vertex/edge in a graph; while temporal graph engines, in addition, calculate the result across multiple snapshots. Chronos makes a decision to batch operations associated with each vertex (or each edge) across multiple snapshots, instead of batching operations for vertices/edges within a certain snapshot [11].

The problem of distributed processing of large dynamic graphs has attracted considerable attention. In this context, several traditional graph operations such as  $k$ -core decomposition and maximal clique computation have been extended to dynamic graphs [28] [1] [22] [14]. While this field of large dynamic graph analysis represent an emerging class of applications, it is not sufficiently addressed by the current graph processing frameworks and only specific graph operations have been studied in the context of dynamic graphs.

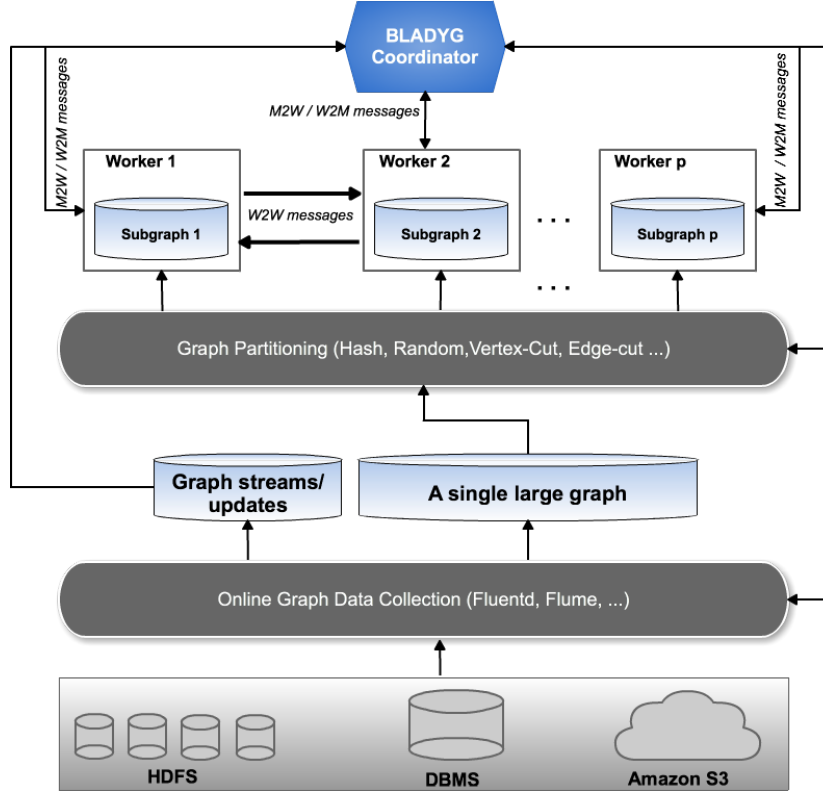


Figure 3: BLADYG system overview

### 3. The BLADYG framework

In this section, we first describe BLADYG and its main components. Then, we give a running example that helps to understand the basic BLADYG operations.

#### 3.1. BLADYG system overview

Figure 3 provides an architectural overview of the BLADYG framework. BLADYG starts its computation by collecting the graph data from various data sources including local files, Hadoop Distributed File System (HDFS) and Amazon Simple Storage Service (Amazon S3). In BLADYG, graph data collection can be done using existing open source collection tools including FLUME [7] and FLUENTD [24]. After collecting the graph data, BLADYG partitions 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 such as hash partitioning, random partitioning, edge-cut and vertex-cut.

- In hash partitioning, edges are distributed across machines according to a user-defined hash function.
- In random partitioning, edges are distributed across machines randomly.
- In vertex-cut, edges are evenly distributed across machines with the goal of minimizing the number of replicated vertices.
- In edge-cut partitioning, the vertices of a graph are divided into disjoint clusters of nearly equal size, while the number of edges that span separated clusters is minimum.

In addition to the provided partitioning techniques, BLADYG users may deploy existing graph partitioning techniques including Metis [12] and JaBeJa [19]. BLADYG users may also implement their own 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 coordinator/master worker 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 either the status of the block and/or the states of the nodes inside 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/W2M-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: (1) an input graph, (2) a set of incremental changes, (3) a sequence of worker/master operations and (4) an output.

1. The input of BLADYG framework is an undirected graph. This graph is represented by a set of vertices and a set of edges. A vertex is defined by its unique ID and a value, whereas an edge is defined by its source ID, target ID, and value.
2. Incremental changes or graph updates consists of edge/node insertions and/or removals. Graph updates are continuously read from the data sources using one of the data collection tools provided by BLADYG.
3. 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.
4. The output of a BLADYG program consists of an updated list of vertices and an updated list of edges.

### 3.2. Illustrative example

Here, we provide an illustrative example to explain the principle of our approach.

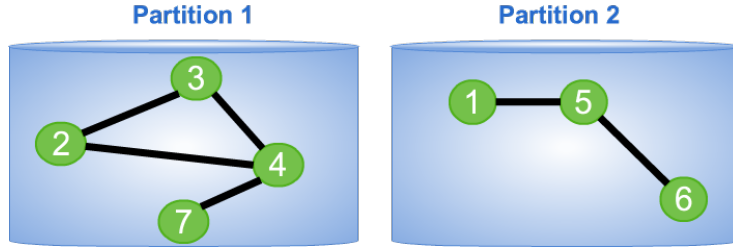


Figure 4: A graph example distributed into two partitions.

Consider the graph  $G = (V, E)$  included in Figure 4, and suppose that it is splitted in two partitions, each processed by a separate worker. We consider the task of computing the degree of all the nodes in  $G$ . The system is completed by the master node, as shown in Figure 5.

A BLADYG solution for computing the degree of all the nodes in a given graph consists of two steps. The first step consists in executing several worker operations in order to compute the degree of nodes in all subgraphs in parallel. As a result of this step, the degree values of all the nodes of  $G$  are computed. The degree values of the nodes of our graph example  $G$  are presented in Figure 5. We assume that the incremental changes in our example consists of only one new edge that links node 4 and node 1. The second step of our BLADYG solution consists in selecting the set of nodes that need to be updated after considering the graph updates (insertion of the edge  $(4, 1)$ ). In this example, only the nodes of the new edge need to be updated (nodes 1 and 4). The master sends a M2W

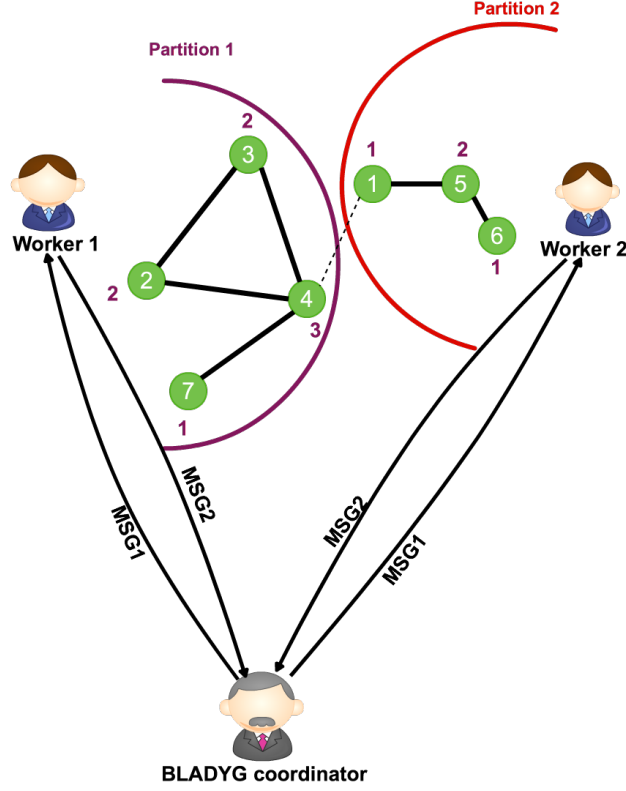


Figure 5: The sequence of messages exchanged among the coordinator and the worker nodes.

message (MSG1) to worker 1 (respectively to worker 2) and asks the worker to increment the degree of node 4 (respectively node 1). The updated degree values of the nodes of our graph example  $G$  are presented in Figure 6.

After updating the degree of the node 4 (respectively node 1), worker 1 (respectively worker 2) sends a notification message (MSG2) to the master. The master checks that all the graph updates were processed and stops the execution of the BLADYG program.

In this example, we only considered an insertion of a new edge between two existing nodes. It is important to mention that in real world applications, graph updates consists of insertion/deletion of several nodes/edges. We also mention that the complexity of the task of selecting the nodes that need to be updated after considering graph updates depends on the considered graph operation.



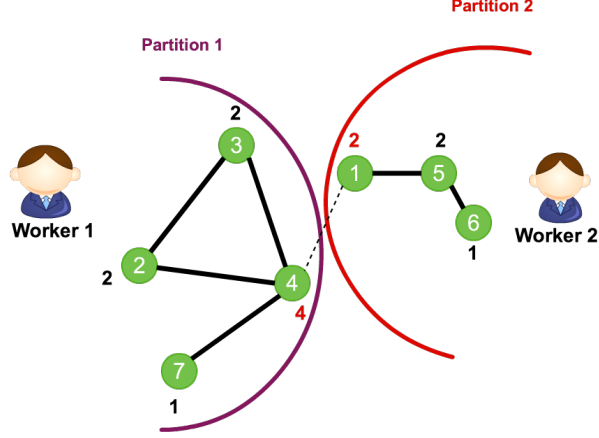


Figure 6: The updated graph.

## 4. Applications

In this section, we apply BLADYG to solve some classic graph operations such as  $k$ -core decomposition [17] [3], clique computation [28] and graph partitioning [10] [20].

### 4.1. 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  $\{V_1, \dots, V_p\}$ ; 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 [6] 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  $\bar{C} \supset C$ , there exists  $v \in \bar{C}$  such that  $d_{G(\bar{C})}(v) < k$ .

**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 [14], 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  $v$  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  $u$  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.

#### 4.2. Distributed edge partitioning

Edge partitioning is a classical problem in graph processing in which edges of a given graph, rather than its vertices, are partitioned into disjoint subsets. Given a graph  $G = (V, E)$  and a parameter  $K$ , an edge partitioning of  $G$  subdivides all edges into a collection  $E_1, \dots, E_K$  of non-overlapping edge partitions, i.e.  $E = \bigcup_{i=1}^K E_i$   $\forall i, j : i \neq j \Rightarrow E_i \cap E_j = \emptyset$ . The  $i^{th}$  partition is associated with a vertex set  $V_i$ , composed of the end points of its edges:  $V_i = \{u : (u, v) \in E_i \vee (v, u) \in E_i\}$ .

A BLADYG solution for edge partitioning in large dynamic graphs consists of two steps. The first step computes the initial partitioning of the input graph. Each vertex of each block maintains  $block(u)$ , which denote the block of the node  $u$ . The second step consists in updating the partitioning according to the incremental changes. Whenever a new edge  $(u, v)$  is added to  $G$ , BLADYG first activates the *M2W-mode* and assigns the edge  $(u, v)$  to a selected block. Block assignment is done by a `masterCompute()` operation that decides the block of each new edge considering predefined objective functions such as balance, communication efficiency and connectedness [10]. The coordinator asks the worker

that holds  $u$  (respectively  $v$ ) to compute the predefined objective functions inside  $block(u)$  (respectively  $block(v)$ ). The result of this computation is sent to the coordinator that decides the block that will hold the edge  $(u, v)$ . Whenever an edge  $(u, v)$  is removed from  $G$ , BLADYG asks all the workers to compute a *repartitioning threshold* in order to decide if the partitioning of  $G$  needs to be recomputed. In each worker, the *repartitioning threshold* is computed by a `workerCompute()` operation and sent to the coordinator. The coordinator decides, by executing a `masterCompute()` operation, if a repartitioning of  $G$  is needed or not.

#### 4.3. 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 [28] 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  $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$  [28]. Maximal cliques that need to be added to the existing ones consists of new maximal cliques that contain  $u, v, w$ , for each  $w \in ((adj_{<}(u) \cap adj_{<}(v)) \cup \{u\})$  [28]. 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  $w \in ((adj_{<}(u) \cap adj_{<}(v)) \cup \{u\})$  [28]. 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	$\phi$	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

## 5. Experiments

We have applied BLADYG framework to both the problem of distributed  $k$ -core decomposition in large dynamic graphs and the problem of partitioning of 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. Implementation details of BLADYG can be found in the following link: <https://members.loria.fr/SAridhi/files/software/bladyg/>.

### 5.1. Experimental environment

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).

### 5.2. Experimental results

#### 5.2.1. $k$ -core decomposition of large dynamic graphs

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 [13], and synthetic datasets, created by a graph generator based on the Nearest Neighbor model [21]. Varying the input data enabled us to avoid biased results specific to a single dataset and thus to have a better interpretation of the results.

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

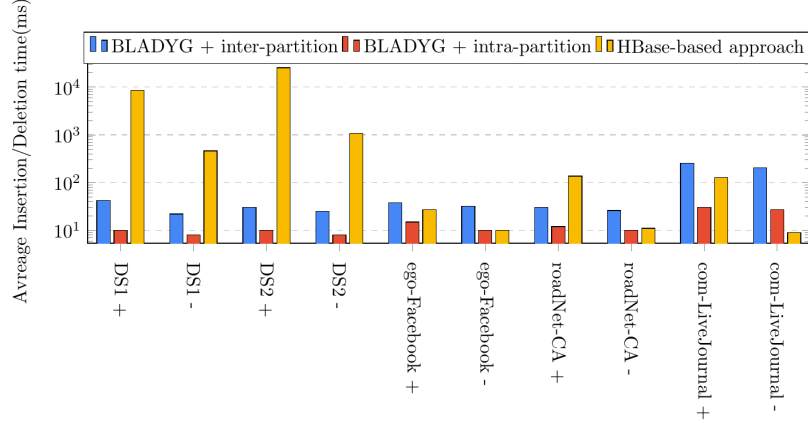


Figure 7: Average insertion/deletion time

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-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 7 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 7, 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.

### 5.2.2. Partitioning of large dynamic graphs

The goal here is to evaluate the performance and the scalability of BLADYG solution for the partitioning of large and dynamic graphs. For our tests, we used the graph datasets described in Table 1 and we considered three partitioning techniques (1) hash partitioning, (2) random partitioning and (3) DYNAMICDFEP, a previously published distributed partitioning algorithm [20]. DYNAMICDFEP is based on two main phases. The first phase consists of four steps:

1. We randomly choose a single node for each of the desired partitions, and give it an initial amount of "funding" associated to that partition.
2. Each node will use its funding to the neighbors to try to "buy" additional edges. The partition will therefore buy the edges that are closer to the randomly chosen nodes and start getting bigger.
3. Since the initial amount of funding is insufficient for the partitions to cover the entire graph, additional funding is assigned to the partitions, in a manner inversely proportional to their size. A small partition (which may have been started far from the center of the graph) will receive more funding and therefore be more likely to grow than a larger partition.
4. Steps 2-3 are repeated until all edges have been bought by a partition.

The second phase of DYNAMICDFEP deals with incremental changes by applying one of the supported update strategies. For our tests, we used the Unit-Based Update strategy (UB-UPDATE) described in [20].

In order to simulate dynamism in each dataset, we use only 90% of the graph in the partitioning step and we insert the remaining 10% in the update step. Each experiment is repeated five times and the numeric results in the following sections consists of the average over all runs.

Tables 3 and 4 illustrate the results obtained with both the real and the synthetic datasets. For each dataset and for each partitioning method, we record the partitioning time (PT) and the update time (UT). The update time is computed for two different partitioning strategies: (1) INCREMENTALPART and (2) NAIVEPART. The first update strategy consists in applying the used partitioning technique only on the incremental changes. The second update strategy is a naive partitioning technique that consists in destroying the old graph partitioning and all further information associated to the assignment and restarts from the scratch by running the used partitioning technique.

As shown in Tables 3, 4 and 5, we observe that, for all partitioning methods, BLADYG results using INCREMENTALPART strategy are much better than those using NAIVEPART strategy. This can be explained by the fact that BLADYG

Table 3: Experimental results using hash partitioning on BLADYG

Dataset	Partitioning time (s)	Update time (s)	
		INCREMENTALPART	NAIVEPART
DS1	18	3	19
DS2	43	5	40
ego-Facebook	11	2	13
roadNet-CA	180	16	193
com-LiveJournal	209	25	227

Table 4: Experimental results using random partitioning on BLADYG

Dataset	Partitioning time (s)	Update time (s)	
		INCREMENTALPART	NAIVEPART
DS1	21	3	20
DS2	36	6	42
ego-Facebook	12	2	10
roadNet-CA	171	21	202
com-LiveJournal	211	27	232

Table 5: Experimental results using DYNAMICDFEP on BLADYG

Dataset	Partitioning time (s)	Update time (s)	
		UB-UPDATE	NAIVEPART
DS1	30	3	32
DS2	56	5	62
ego-Facebook	80	2	91
roadNet-CA	254	31	321
com-LiveJournal	509	34	572

allows to process only incremental changes without restarting the partitioning from the scratch.

## 6. 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 two different problems: (1) distributed  $k$ -core decomposition in large dynamic graphs and (2) partitioning of 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.

## 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] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis. Distributed  $k$ -core decomposition and maintenance in large dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS ’16, pages 161–168, New York, NY, USA, 2016. ACM.
- [4] S. Aridhi, A. Montresor, and Y. Velegrakis. BLADYG: A novel block-centric framework for the analysis of large dynamic graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP ’16, pages 39–42, New York, NY, USA, 2016. ACM.
- [5] S. Aridhi and E. M. Nguifo. Big graph mining: Frameworks and techniques. *Big Data Research*, 6:1–10, 2016.
- [6] 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.



- [7] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.
- [8] 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.
- [9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.
- [10] A. Guerrieri and A. Montresor. DFEP: distributed funding-based edge partitioning. In *Proc. of the 21st Int. Conf. on Parallel and Distributed Computing (Europar'15)*, pages 346–358, 2015.
- [11] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.
- [12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.
- [13] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.
- [18] 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.

- [19] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 51–60, Sept 2013.
- [20] C. Sakouhi, S. Aridhi, A. Guerrieri, S. Sassi, and A. Montresor. DynamicD-FEP: A distributed edge partitioning approach for large dynamic graphs. In *Proceedings of the 20th International Database Engineering & Applications Symposium, IDEAS '16*, pages 142–147, New York, NY, USA, 2016. ACM.
- [21] 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.
- [22] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, Apr. 2013.
- [23] 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.
- [24] S. Singh. Cluster-level logging of containers with containers. *Queue*, 14(3):30:83–30:106, May 2016.
- [25] 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.
- [26] D. Wyatt. *Akka Concurrency*. Artima Inc., 2013.
- [27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES '13*, pages 2:1–2:6, New York, NY, USA, 2013. ACM.
- [28] 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.
- [29] 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.
- [30] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.