DOI 10.12694/scpe.v18i1.1232
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 35–49. http://www.scpe.org

TELOS: AN APPROACH FOR DISTRIBUTED GRAPH PROCESSING BASED ON OVERLAY COMPOSITION

PATRIZIO DAZZI*, EMANUELE CARLINI†, ALESSANDRO LULLI‡, AND LAURA RICCI§

Abstract. The recent years have been characterised by the production of a huge amount of data. As matter of fact, human being, overall, is generating an unprecedented amount of data flowing in multiple and heterogeneous sources, ranging from scientific devices, social network, business transactions, etc. Data that is usually represented as a graph, which, due to its size, is often infeasible to process on a single machine. The direct consequence is the need for exploiting parallel and distributed computing frameworks to face this problem. This paper proposes Telos, an high-level approach for large graphs processing. Our approach takes inspiration from overlay networks, that is a widely adopted approach for information dissemination, aggregation and computing orchestration in highly distributed systems. Telos consists of a programming framework supporting the definition of computations on graphs as a composition of overlays, each devoted to a specific aim. The framework is implemented on the top of Apache Spark. A set of experimental results is presented to give a clear evidence of the effectiveness of our approach.

Key words: Programming Models, Distributed Architectures, Self-Organisation

AMS subject classifications. 68M14, 68N19, 68W15

1. Introduction. Our world contains an unimaginably vast amount of digital information which is getting ever vaster rapidly. In fact, the amount of data produced every day by human beings is far beyond what has ever been experienced before. According to some statistics, the 90 percent of all the data in the world available in May 2013 has been generated over 2012 and 2013. As a further example, in 2012 were created, every day, 2.5 exabytes (2.5×10^{18}) of data [33] which comes from multiple and heterogeneous sources, ranging from scientific devices to business transactions. A significant part of this data is gathered and then modelled as a graph, such as social network graphs, road networks and biological graphs.

A careful analysis and exploitation of the information expressed by this data makes it possible to perform tasks that previously would be impossible to do: to detect business trends, to prevent diseases, to combat crime and so on. If properly managed, data can be used to unlock new sources of economic value, provide fresh insights into science and hold governments to account. However, the vastity of this data makes infeasible its processing by exploiting the computational and memory capacity of a single machine. Indeed, a viable solution to tackle its analysis relies on the exploitation of parallel and distributed computing solutions.

Many proposal for the parallel and distributed processing of large graphs have been designed so far. However, most of the methodologies currently adopted fall in two main categories. On the one hand, the exploitation of low-level techniques such as send/receive message passing or, equivalently, unstructured shared memory mechanisms. This is the way traditionally adopted to process large dataset, usually with parallel machines or clusters. Unfortunately, this approach leads to few issues. Low-level solutions are complex, error-prone, hard to maintain and their tailored nature hinder their portability.

On the other hand, a wide use of the MapReduce paradigm [19] can be observed, which has been inspired by the well-known map and reduce paradigms that is part of the algorithmic skeletons, which across the years have been provided by a number of different frameworks [37, 32, 18, 4, 2, 3, 13]. MapReduce-based solutions often exploit the MapReduce paradigm in a "off-label" fashion, i.e., in ways and contexts that are pretty different from the ones it has been conceived to be used in. Some of the most notable implementations of such paradigm are frequently used with algorithms which could be more fruitfully implemented with different parallel programming paradigms or different ways to orchestrate their computation. This is especially true when dealing with large graphs [29]. In spite of this, some MapReduce based frameworks achieved a wide diffusion due to their ease of use, detailed documentation and very active communities of users.

*Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", National Research Council of Italy, Pisa, Italy, (patrizio.dazzi@isti.cnr.it).
†Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", National Research Council of Italy, Pisa, Italy, (emanuele.carlini@isti.cnr.it).
‡CS Department, University of Pisa, Pisa, Italy, (lulli@di.unipi.it).
§CS Department, University of Pisa, Pisa, Italy, (ricci@di.unipi.it).

Recently, a few solutions have been proposed to let the MapReduce frameworks more suitable for performing analysis on large graphs in a more natural way. Some of them have been inspired by the BSP bridging model [44] that resulted in a good approach to implement certain algorithms whose MapReduce implementation would be either too complex or counter-intuitive. Malewicz *et al.* presented this approach when proposing the Pregel framework [29]. It provides the possibility to describe graph processing applications from the viewpoint of a graph vertex, which processes independently and can only access its local state and is aware only of its neighbourhood.

In this paper we propose Telos[1], an approach to process graphs by leveraging the vertex-centric approach to provide a solution that focuses on the orchestration of nodes and their interactions. Our approach takes inspiration by the similarities existing between large graphs and massively distributed architectures, e.g., P2P systems. In fact, many of these solutions orchestrate the computation and spread the information exploiting *overlay networks*. Essentially, an overlay is a virtual network, built upon the existing physical network, connecting nodes by means of logical links established according to a well-defined goal. The building blocks of overlays match the key elements of graphs. Vertices can be seen as networked resources and edges as links.

We believe that by means of these similarities overlay-based solutions can be fruitfully ported and exploited for graph processing. We already presented a first analysis about the applicability of Telos to solve graph partitioning [8]; we also gave a brief description of its conceptual architecture [26] and its programming model [9]. This paper gives a consolidated presentation of Telos. Along with a more detailed description of the approach, we present an advanced application scenario showing the advantages deriving from the exploitation of our proposed approach. More in details, the contributions of this paper are the following:

- the definition of a *high-level* programming approach based on the composition of overlays, targeting computations on large graphs;
- the presentation of *our framework* built on top of Apache Spark [50] providing both a high level API to define custom layers and some built-in layers. This gives the possibility to dynamically define graph topologies different from the original one, so enabling each vertex to choose the most promising neighbours to speed-up the convergence of the distributed algorithm;
- an extensive proof-of-concept demonstrator to assess the feasibility of Telos;
- a case study where we show how to port a popular graph partitioning algorithm in Telos. In addition, we enhanced such algorithm by means of the Telos' layered architecture showing through an experimental testbed the improvements on the quality of the obtained results.

The remainder of this paper is organised as follows. In Section 2 we discuss the current related work on distributed graph processing framework, with a focus on those able to evolve the graph during the computation. Section 3 presents the main design choices and the structure of Telos as well as its programming model and architecture. We evaluate our approach in Section 4 by sketching the Telos implementation of an overlay building protocol and a balanced graph partitioning case study. Finally, Section 5 reports some final remarks and further ideas for future work.

2. Related Work. Comprehensive surveys of widely adopted distributed programming models targeting graphs, and of the derived frameworks is presented in several survey contributions, such as by McCune *et al.* [34], Kalavri *et al.* [21], Doekemeijer *et al.* [16]. Most of the current frameworks provide a set of basic features for distributed graph processing, while several approaches have been developed with the goal of extending and/or optimizing the basic frameworks. In this section we focus on the approaches that are closely related to our work.

Among the several existing distributed programming frameworks focusing on graphs, a notable amount are characterised by the *vertex-centric* programming model. However, not all of them allow graph mutation during the execution, which is a core feature in our proposed approach.

Pregel [29] is the Google framework for distributed graph processing, and the one that de-facto defined the vertex-centric model. Pregel has been inspired by the BSP model [44], in which computations are organised in sequential supersteps; in Pregel, like in BSP, during each superstep each vertex executes a specific function. Vertices communicate in a message-passing model, by sending and receiving messages from/to other vertices.

[1] https://github.com/hpclab/telos

Apache Giraph [12] is another vertex-centric framework initially developed by Yahoo!. Conversely from Pregel, Giraph is open-source and runs over Hadoop.

More recent solutions are aimed at optimising the basic vertex-centric model provided by Pregel. GPS [41] provides a few interesting optimisations specifically targeted to vertex-centric graph processing. These solutions include an adaptive rescheduling of the computation, essentially aimed at performing the computation sequentially on the master node when the number of active nodes is below a given threshold. In such a case the exploitation of parallelism is no longer justified but just became an overhead. Another optimisation consists in merging several vertices together to form supervertices to reduce the communication cost. Both of these optimisations focus on reducing the completion time and the communication volume by instrumenting the support with automated recognition features, activated to speed-up the computation.

Another interesting approach, which goes beyond classical vertex-centric solutions, has been proposed by Tian *et al.* [43] and it is called as *partition-centric*. Their idea is to shift the reasoning from the single vertex perspective to the viewpoint of a set of vertices aggregated into a partition, in order to reduce the inter-vertex communication and to accelerate the convergence of vertex-centric programs. By means of such a change of perspective they have been able, on selected problems, to yield a notable speeds-up in the computation. The partition-centric model has been further specialized in other works, such as Simmhan *et al.* [42] and Yan *et al.* [48]. These proposals and our approach are based on the idea of pushing, by design, a shift in the paradigm that allows the implementation of more efficient solutions.

Distributed Graph Partitioning. Graph partitioning is an NP-Complete problem well known in graph theory. Distributed graph partitioning assumes the impossibility to access the entire graph at once, with the graph distributed among various workers nodes. In the context of a MapReduce computation, several approaches have considered the problem of distributed graph partitioning.

JA-BE-JA [40] is a distributed approach based on iterations of local nodes swapping to obtain a partition of the graph. Such an approach has been originally designed for extremely distributed graphs, in which a node can be associated with a vertex of the graph, but can be ported to MapReduce platforms with some tweaking [8]. Sheep [30] consists in a distributed graph partitioning algorithm that reduces the graph to an elimination tree to minimizes the communication volume among the workers of the MapReduce computation. Spinner [31] employs an heuristics based on the label propagation algorithm for distributed graph partitioning. A penalty function is applied to discourage the movements of nodes toward larger partitions. Similarly, Meyerhenke et al. [35] employs a size-constrained label propagation targeting heavily clustered and hierarchical complex networks, such as social networks and web graphs.

3. From vertices to overlays. The vertex-centric approach, also known as Think Like A Vertex (TLAV), is currently being exploited in several applications aimed at computing big data analysis involving large graphs, and widely used by both academia and industry. Many popular implementations of the TLAV model [29, 12] are based on a synchronized execution model according to the well-known BSP approach [44]. The core of BSP computations consists of three main pillars:

- *Concurrent computation*: Every participating computing entity (a vertex in this case) may perform local computations, i.e., computing by making use of values stored in the local memory/storage.
- *Barrier synchronisation*: After the termination of its local computation, every vertex is blocked until all the other vertices composing the graph have reached the same stage.
- *Communication*: All the vertices exchange data between themselves, in a point-to-point fashion, after they reach the synchronization barrier. Each message prepared during superstep S is received by the recipient at superstep $S + 1$.

From an operative viewpoint, during a superstep, each vertex receives messages sent in the previous iteration and executes a user-defined function that can modify its own state. Once the computation of such a function has terminated, it is possible for the computing entity to send messages to other entities.

To orchestrate the computation, the BSP model relies on a synchronisation barriers occurring at the end of every superstep. The same occurs in TLAV models, even if the vertex centric abstraction could, in principle, be realized with other computation models that do not rely on a strong synchronisation. Indeed, the programming approach that characterises the vertex-centric model, in which programmers are in charge of coding the behaviour of each single element, which globally cooperates to contribute to the whole computation, is neither

new nor exclusive of this kind of solutions. For instance, it is also the approach adopted in actor-based models.

The correspondence with the superstep defined by the BSP model is realized as the following. The computation is described by the method *compute()* described in Section 3.1.1. Regarding the communication, the messages are generated by the compute method. A message can be inter-vertex, intra-vertex or extra-protocol as defined in Section 3.1.2. The actual communication is materialized in the reduce phase. The synchronisation is enforced by the Spark framework, which assures that an iteration begins only when all nodes have executed the reduce of previous iteration.

In some ways, programming according to the vertex-centric model also recalls the definition of epidemic (or gossip) computing [5, 36, 15, 9, 17]. A widely used approach in massively distributed systems leading nodes of a network to work independently but sharing a common, global, aim. Indeed, according to a common strategy followed by many existing gossip protocol, nodes build logic network overlays to exploit for information exchange. An overlay consists of a logical communication topology, built over an underlying network, which is maintained and used by nodes. Across the years, several overlay-based protocols have been proposed for data sharing, diffusion, aggregation and for computation orchestration, as well. Usually, these solutions are adopted in highly distributed systems, e.g., massively multiplayer games [10], distributed replica management [22], distributed resource discovery [14, 7], etc. Most of these overlay-oriented solutions are layered [24] and exploit the different overlays to achieve very different tasks. Overlay-based algorithms are characterised by several different interesting features:

- *Local knowledge*: algorithms for overlays are based on local knowledge. During the computation each node relies only on its own data and the information received from its neighbours, resulting in a reduced access to the network and consequently improving the scalability of the approach.
- *Multiple views*: multi-layer overlays have been a successful trend [20]. These approaches build a stack of overlays, each organised accordingly to a specific goal. By means of their combination more complex organisations can be achieved, fostering an efficient and highly targeted dissemination of information.
- *Approximate solutions*: since overlays are usually based on an approximated knowledge on the graph, several algorithms designed for overlays are conceived to deal with approximated data. Usually, solutions are achieved by these approaches in an incremental and iterative way, allowing to achieve in a flexible way approximated solutions.
- *Asynchronous approach*: differently from vertex-centric algorithm, gossip-based approaches operate in an unreliable, not synchronized environment, in which messages can be received later than expected and their content be not consistent. Due to this, gossip protocols are able to work under the assumption of approximate and incremental results.

Our approach is aimed at introducing techniques for large graph processing borrowed from solutions targeting massively distributed systems, for instance solutions defined within the scope of Peer-to-peer computing area, into large graph processing framework. We believe that this approach could be useful to support large graph processing by means of the composition of solutions based on multiple-layer graphs, eventually combined to define more effective graph processing algorithms.

3.1. Telos. Telos builds upon the overlay-based approach to provide a tool aimed at large graph processing whose programming model is organised according to the vertex-centric approach for graph processing.

A set of protocols, one for each network overlay, is associated with each vertex. For each overlay, each vertex maintains a local context and a neighbourhood. The former represents the "state" of the vertex w.r.t. a given overlay, the latter represents the set of vertices that are exchanging messages with such a vertex by means of that overlay, according to the associated protocol. Both the context and the neighbourhood can change during the computation and across the supersteps, given the possibility of building evolving "graph overlays".

Telos brings to the vertex-centric model many of the typical advantages of a layered architecture:

- *modularity*: protocols can be composed and additional functions can be obtained by stacking different layers;
- *isolation*: changes occurring in a overlay, by means of a protocol, do not affect the other protocols;
- *reusability*: protocols can be reused for many and possibly different computations.

Telos supports programmers in combining different protocols. Each protocol is managed independently and all the communications and the organisation of the records are all in charge of Telos.

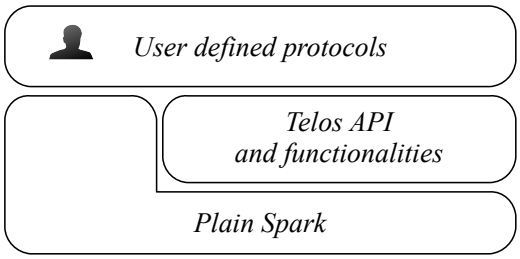


FIG. 3.1. *Telos integration with Spark*

TABLE 3.1
The Protocol Interface

function	**description**
getStartStep() : Int getStepDelay() : Int	defines the first step on which `compute()` is called defines the delay in terms of steps between two successive calls of `compute()`
compute(ctx:Context, mList:List[Message]) : (Context, List[Message])	contains the business code of a protocol, e.g., message handling, data precessing, etc.

In its current implementation, the Telos framework is built on top of Spark [50]. One of the key aspects of this tool are the Resilient Distributed Dataset (RDD) [49], on top of which are natively defined collective operations such as map, reduce and join. By relying on these operations, Telos provides its own TLAV abstraction. All the tasks for managing RDDs are transparent to the programmers and managed by Telos.

Figure 3.1 shows the integration of the Telos framework with Spark. The framework coordinates the protocols and masquerades the underlying support to ease the application development. The framework handles all the burden required to create the initial set of vertices and messages and provides to each vertex the context it requires for the computation. Communications are completely hidden to the programmers as well. Telos handles data dispatch to the target vertices.

3.1.1. Protocols. Protocols are first-class entities in Telos. They are the abstractions that orchestrate the computation and organise the topologies of each overlay. Protocols manage the context and the neighborhood of all the vertices. In fact, each protocol is in charge of:

- modifying the state associated to each vertex;
- defining a representation describing the state of the vertices, which is eventually sent as messages to the other vertices;
- defining custom messages aimed at supporting the overall orchestration of the nodes;

Table 3.1 describes the protocol interface, i.e., the main methods provided by the interface of a protocol (for the full table please refer to our former papers focusing on the description of such an interface [26, 9]). In a nutshell, the Protocol interface abstracts the structure of the computation running on each vertex. Basically, the core logic of a Protocol is contained in the `compute()` method. Once called, say at superstep S, the `compute()` method can access the whole state, associated to the vertex, and to the messages received by such vertex during the step S - 1. The output of the `compute()` method is a new vertex context and a set of messages to be dispatched to the target vertices.

The termination of a Protocol is coordinated by a "halt" vote. At the very end of its computation each vertex votes to halt, and when all vertices voted to halt, the computation terminates.

The frequency at which a protocol is activated (w.r.t. the supersteps) is regulated by `getStartStep()` and `getStepDelay()` methods. These methods are useful when a computation involves different protocols characterised by different convergence time, i.e., the amount of steps it needs to converge to a useful result. By means of these methods it is possible to regulate the activations of the protocols with respect to the amount of

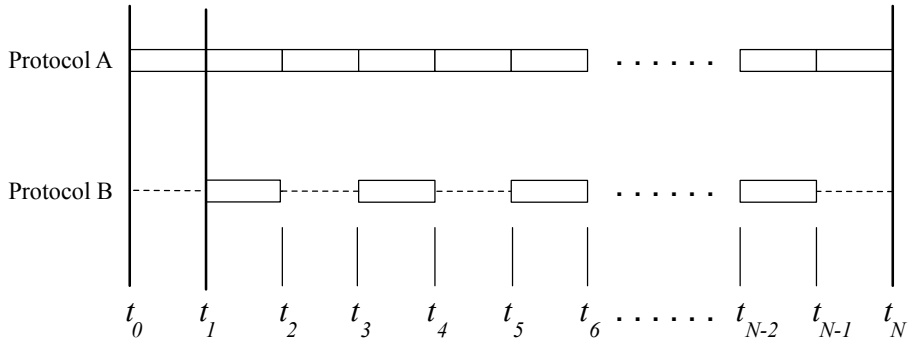


FIG. 3.2. *The effect of* `getStartStep()` *and* `getStepDelay()` *methods*

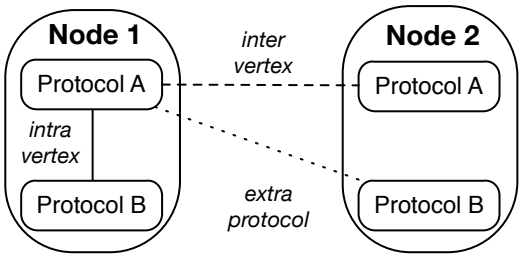


FIG. 3.3. *Telos interactions*

elapsed supersteps.

Figure 3.2 graphically depicts the behaviour of these methods. In the figure, *Protocol A* is activated at each superstep, i.e., if invoked, its implementation of `getStepDelay()` method, will return 1, whereas the very same call on *Protocol B* will return 2. In a pretty similar fashion the method `getStartStep()` drives the *first* activation of a protocol. Going back to the aforementioned figure, when called on *Protocol A* it will return 0, whereas it will return 1 if called on *Protocol B*.

3.1.2. Interactions. Telos enables three different types of interactions that involve the vertices belonging to the graph.

- *intra-vertex*: the internal access made by a given vertex, when executing a certain protocol, to the context associated at a different protocol.
- *inter-vertex*: the remote access made by a given vertex, when executing a certain protocol, to the context of another vertex, associated to the same protocol.
- *extra-protocol*: the remote access made by a given vertex, when executing a certain protocol, to the context of another vertex, associated to a different protocol.

Figure 3.3 shows an example of the aforementioned interactions. In the example, the Protocols A and B of Node 1 have an inter-vertex interaction. Protocol A of Node 1 and the same protocol on Node 2 have a inter-vertex interaction. Finally, Protocols A of Node 1 and protocol B of Node 2 have a inter-vertex interaction.

To summarise, the computation of TELOS is defined as the execution of a sequence of supersteps. At each superstep, the messages from the previous superstep are collected, a subset of the protocols are executed and both inter-vertex and extra protocol messages are sent, afterward the synchronisation barrier is executed.

3.1.3. Built-in Protocols. To evaluate the effectiveness of porting approaches from massively distributed system into the distributed large graph analysis we developed two protocols with Telos, which we briefly present in this section.

- *Random protocol.* The aim of this protocol is to provide to each vertex with a random vertex identifier upon request. The vertex identifier must be taken uniformly and randomly in the space of identifiers of graph vertices. It is implemented according to a random peer sampling approach [45]. From an

operative viewpoint, the random peer sampling protocol requires that each vertex periodically shares information about its neighbourhood with a random vertex, that will do the same. At the end, each vertex remains with a neighbourhood composed by k vertices, with k size of the neighbourhood.

- *Ranking-function protocol.* Highly distributed solutions exploit ranking functions to create and manage overlays built according to a specific topology [20, 46]. The ranking function determines the similarities between two vertices and the topology is built accordingly. In Telos we implemented a generic ranking protocol able to take as input the ranking function to realise the consequent topology.

4. Evaluation. To assess the evaluation of our approach we conducted several experiments. First, in Section 4.1 we provide a proof-of-concept implementation that validates our framework by implementing a stack composed by the two protocols, instantiating the random and the ranking-function protocols discussed above. Then, in Section 4.3 we evaluate the scalability provided by our implementation of Telos.

All the experiments have been conducted on a cluster running Ubuntu Linux 12.04 consisting of 4 nodes (1 master and 3 slaves), each equipped with 16 Gbytes of RAM and 8-cores. The nodes are interconnected via a 1 Gbit Ethernet network.

4.1. A Proof of Concept Implementation. The aim of this experiment is to verify the correctness of Telos when dealing with a layered composition of protocols. We built a two layered structure composed of a random protocol layer and a ranking protocol layer. We generated a set of points in 3D space and we assigned to each node a point. The points are generated in order to form a torus. Then, the neighborhood of each node is randomly initialized and the execution started. By using the ranking protocol, we iteratively connect nodes that are close to each other considering the euclidean distance. At the end of the execution, each node is connected to the most similar nodes according to the euclidean distance forming a torus-shaped graph.

We tested the implementation on a graph made of $20K$ vertices. The results in Figure 4.1 show the evolution of the graph in different iterations. Note, we plot the graphs using Gephi[6] and the default layout for placing the nodes. Due to this, although the point assigned to a vertex does not change it may happen that the same point is in different positions in the figures. The topology recalls the shape of a torus already at super-step 10. At super-step 20 no edges connect "distant" vertices in the torus. This experiment shows that, if properly instrumented, Telos can correctly manage multiple layers to organize network overlays according to user-defined ranking functions.

In the following, to have a deeper understanding of Telos, we present a pseudo-code implementation of the protocols of this experiment. The pseudo-codes are inspired by the original Scala[38] implementation. Code Fragment 1 presents the implementation of the random protocol. The idea is to randomly generate a number upon request on the method `getVertexId()`. The generated numbers must be in the range $[startRange, endRange]$. Where, in general, *startRange* is equal to 0 and *endRange* is equal to the number of nodes in the graph. The main issue is that each machine executing the protocol must be able to call such method. To do this, in the initialization (`init` method) we create two broadcast variables initialized with the values representing the required range. The RandomContext contains the logic to generate a random number and it is initialized with the two broadcast variables. In this experiment we use the convention to set the ids of the nodes in the range $[1, 20000]$ without missing values. Finally, the random protocol is initialized with such values and the ranking protocol makes use of the method `getVertexId()` to select a node identifier in order to communicate with a different and random node in each iteration.

Instead, code Fragment 2 presents the pseudo-code of the implementation of the ranking protocol. The implementations of `beforeSuperstep`, `afterSuperstep`, `init` and `createInitMessage` are not given because not required for this kind of Protocol. Specifically, the RankingProtocol extends the Protocol interface described in Table 3.1. We instruct the framework to start this protocol at the beginning of the computation with `getStartStep` equals to 1 (Line 3) and to run in each step of the computation with `getStepDelay` (Line 4) equals to 1. The `createContext` method (Line 6) is called at the beginning of the computation to initialize the state on each vertex. The data returned can be any custom type to allow the definition of custom data structures to record all the data needed by the protocols.

The ranking protocol uses the custom defined *RankingContext* to save the coordinate position of a point represented by the vertex and the neighbourhood of the vertex. Recall that Telos calls, for the active protocols, in each super-step the `compute()` method for each active vertex (Line 19) providing both a vertex context *ctx*

Code Fragment 1: `RandomProtocol extends Protocol`

```
class RandomProtocol extends Protocol
{
   def getStartStep() = return 1
   def getStepDelay() = return 1

   def init(sc:SparkContext) =  {
       startRangeBroadcast = sc.broadcast(startRange)
       endRangeBroadcast = sc.broadcast(endRange)
   }

   def compute[RandomContext](ctx:RandomContext,
        mList:List[Message]):(RandomContext, Message) = {
       return (ctx, List())
   }

   def createContext[RandomContext](row:Array[String]):
       RandomContext = {
        return new RandomContext(startRangeBroadcast,
                                 endRangeBroadcast)
   }
}

class RandomContext(val startRngBrd: Broadcast[Long],
          val endRngBrd:Broadcast[Long]) extends VertexContext {
   def getVertexId(): Long = {
       return (random.nextDouble() * (endRngBrd.value -
              startRngBrd.value)).toLong+startRngBrd.value
   }
}
```

and the messages received by the vertex *mList*. In this case, the vertex makes an *intra-vertex* communication to access the context of a different protocol on the same vertex and retrieve the *RandomContext*. Such context is used to choose randomly a counterpart to communicate with and exchange information. In addition, for each message received the vertex prepares a response message. In the last part of the method the vertex orders all the data received and keeps the vertices that are closer to the assigned 3D point. The value k is a custom parameter and in this experiment it is initialized to 5. The value returned by the `compute` method is the new vertex context and a list of messages to be dispatched by Telos to the specified vertices.

4.2. Case Study: Balanced Graph Partitioning. JA-BE-JA [40] is a distributed algorithm for computing the balanced k-way graph partitioning problem and it works as follows. Initially, it creates k logical partitions on the graph by randomly assigning colours to each vertex, with the number of colours equals to k. Then, each vertex attempts to swap its own colour with another vertex according to the most dominant colour among its neighbours, by: (i) selecting another vertex either from its neighbourhood or from a random sample, and (ii) considering the "utility" of performing the colour swapping operation. If the colour swapping would decrease the number of edges between vertices characterised by a different colour, then the two vertices swap their colour; otherwise, they keep their own colours.

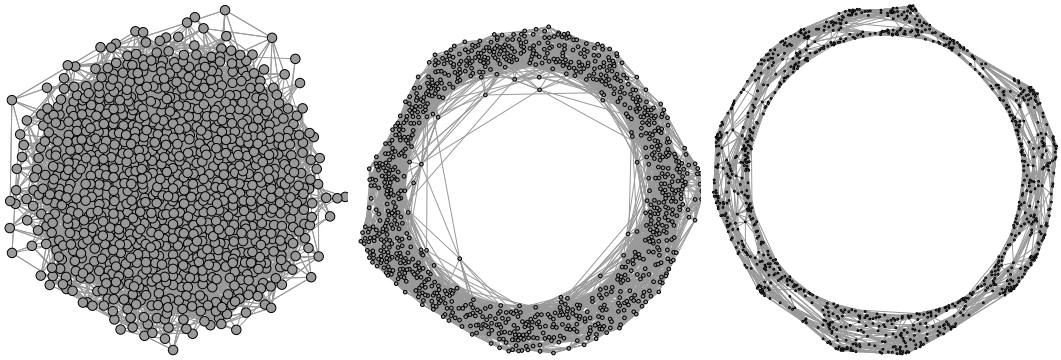
In a previous work [8] we presented an implementation of JA-BE-JA in Apache Spark outlining the adaptations that have been required in order to efficiently adapt the algorithm to match a BSP-like structure. In its original formulation, JA-BE-JA assumes that each vertex has complete access to the context of its neighbours and also their neighbourhood. To enforce this assumption, we initially introduced specific messages to retrieve

Code Fragment 2: `RankingProtocol extends Protocol`

```
1  class RankingFunction extends Protocol
2  {
3     def getStartStep() = return 1
4     def getStepDelay() = return 1
5
6     def createContext[RankingContext](row: Array[String]) :
7     RankingContext = {
8         var context = new RankingContext
9         context.x = row(0)
10        context.y = row(1)
11
12        for (i <- 2 until row.length)
13        {
14            context.neighbour.add(row(i))
15        }
16        return context
17    }
18
19    def compute[RankingContext](ctx:RankingContext,
20                mList:List[Message]):(RankingContext,Message) = {
21        var message = List()
22        val randomContext = ctx.accessProtocol("RandomProtocol")
23        val rcpt = randomContext.getVertexId()
24        val buffer = ordByEucDis(rcpt, ctx.neighbour)
25        message = message ++ new Message(rcpt, getTopK(buffer))
26
27        var newData = List()
28        foreach(msg : mList)
29        {
30            newData = newData ++ msg.content
31            val buffer =
32               ordByEucDis(msg.sender,ctx.neighbour++msg.content)
33
34            message =
35               message ++ new Message(msg.sender,getTopK(buffer))
36        }
37
38        val myBuf = ordByEucDis(ctx.self,ctx.neighbour++newData)
39        return (new RankingContext(ctx.self, myBuf),message)
40    }
41 }
```

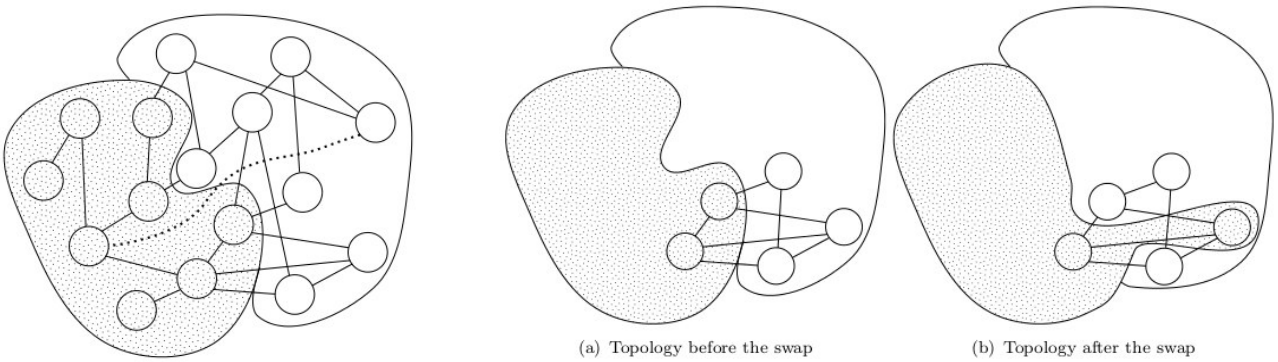
the neighbourhood of any vertex. However, we noticed that forcing a strong consistency of such information slows down the performance too much. As a consequence, we provided an alternative implementation (called GP-SPARK) that introduces a degree of approximation to accelerate the computation, in which vertices piggy-back their neighbourhood information in other messages. This mechanism causes the vertices to apply the local heuristics on possibly stale data, but increases the performance of the original approach providing a comparable quality of results with respect to the original version.

The original GP-SPARK works with a two layered architecture composed as follows: (i) the *colour swapping*



(a) Superstep 0 (b) Superstep 10 (c) Superstep 20

FIG. 4.1. *Evolution of the torus overlay at different Supersteps*



(a) Topology before the swap (b) Topology after the swap

FIG. 4.2. *Swap in the stain metaphor of the JA-BE-JA topology*

protocol, that attempts the colour swapping targeting either a vertex from the local neighbourhood or from the random sampling, and (ii) the random sampling protocol that provides some random vertices to the colour swapping protocol. Here, we present GP-TELOS, an improved version of GP-SPARK that introduces a new layer with the *border ranking protocol* aimed to boost the quality of results. The objective is to give to JA-BE-JA the possibility to select from a better set of vertices when attempting colour swapping.

Recall, the layered approach exploits orchestration between several protocols relying over the intra-communication facility. GP-TELOS adds a new layer that interacts with the colour swapping one. First of all, the colour swapping's layer is initialized with the input graph topology. Then, we have a random layer to provide to each vertex with some long range links that are created through the retrieve of a small sample of random nodes for each vertex. This layer is refreshed with new vertices in each iteration.

As shown in Figure 4.2.a, the colour swapping layer could be thought as a set of stains which interact with each other *moving* like fluids. In this metaphor, the long range links fold the stains in a N dimensional space, where N is the number of links maintained in the random layer. Such links allow the interaction between areas of the stains that are not topologically near each other in the color swapping layer. The JA-BE-JA algorithm orchestrates those stains in order to minimize the edge-cut. Whenever a swap happens is like a stain flows rolling to the neighbouring stain, as shown in Figure 4.2.b.

In GP-TELOS we introduce an additional layer of the *ranking* kind. This relies over the definition of a ranking function. The idea is to abstract the property of *borderness* of each vertex in the color swapping layer. Whenever 2 vertices are on the borders the swap could take place. Figuratively the vertices cross the boundaries of the stain.

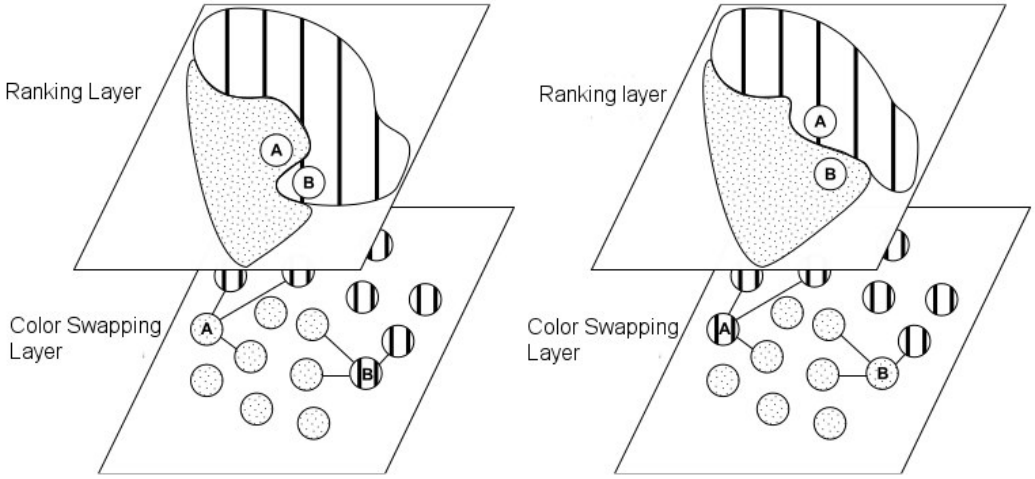


FIG. 4.3. *Example of an exchange colour using the ranking layer over the real JA-BE-JA coloured graph.*

TABLE 4.1
Edge-cut value for GP-TELOS *and* GP-SPARK *with the three datasets*

K	3elt		Vibrobox		Facebook	
	GP-TELOS	GP-SPARK	GP-TELOS	GP-SPARK	GP-TELOS	GP-SPARK
2	750	**1,433 (+91%)**	14,812	22,244 (+50%)	75,690	80,971 (+7%)
4	1,810	2,903 (+60%)	30,432	40,358 (+33%)	147,991	157,282 (+6%)
8	3,048	4,473 (+47%)	43,728	56,954 (+30%)	256,902	**245,682 (-4%)**
16	4,191	6,344 (+51%)	54,339	75,051 (+38%)	348,494	353,061 (+1%)
32	5,241	8,491 (+62%)	67,787	95,858 (+41%)	415,315	457,257 (+10%)
64	6,419	10,622 (+65%)	88,953	116,149 (+31%)	520,391	552,714 (+6%)

The *ranking function* between two nodes u and v is defined as follow:

$$H(z) = \left| \frac{|\{q|q \in neighbor(z) \wedge color(q) \neq color(z)\}|}{|neighbour(z)|} \right| \tag{4.1}$$

$$rankingFunction(u,v) = \begin{cases} +\infty, & \text{if } color(u) = color(v) \\ |H(u) - H(v)|, & \text{otherwise} \end{cases} \tag{4.2}$$

Note, two peers u and v are more similar as the $rankingFunction(u,v) \to 0$. The function ranks similar two peers u and v that have the same amount of neighbours of different colour and u and v are coloured differently. As a consequence, peers that are in the middle of a partition are similar to the ones that are in the middle of another partition (i.e. a node u has all the neighbours of its very same color, $H(u) = 1$) and, more useful for our purposes, vertices which are on the borders of a partition are similar to others that are also on the same kind of border. For instance, a blue vertex having 5 neighbours with 3 coloured red would be ranked high from a red vertex having 5 neighbours with 3 coloured blue. As a result, in a figurative way as shown in Figure 4.3, the stains flow between each other like a liquid. In this way, the ranking layer keeps in the neighbourhood of each vertex other vertices having high similarity according to *rankingFunction*. The aim is to retrieve a better solution starting from the actual topology to another more accurate for the purpose. Moreover, such an abstraction allows to find better candidates to exchange the colour with and to filter and organize the vertices among their *borderness* property.

We compared the performance of GP-TELOS and GP-SPARK by means of the following metrics: (i) *edge-cut*: the number of edges that cross the boundaries of each subgraph. This metrics gives an estimation about

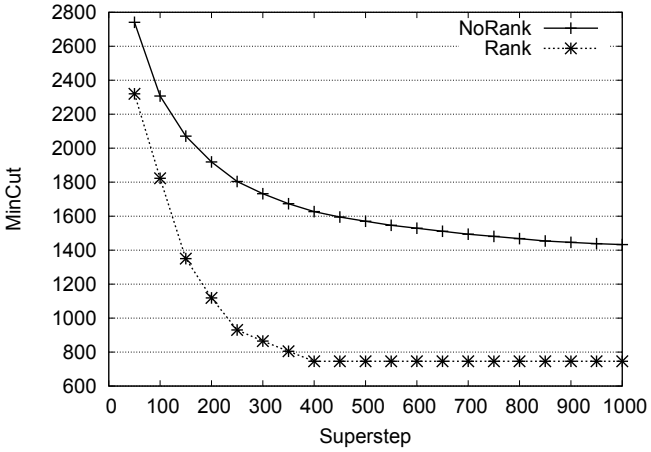


FIG. 4.4. *Convergence of edge-cut over supersteps*

the quality of the cut, with lower values corresponding to a better cut, and (ii) *convergence*: the number of supersteps required to achieve a substantially definitive edge-cut result. Our aim is to show that Telos does not affect the performance in term of supersteps to find a solution with respect to the GP-SPARK implementation.

The experiments have been conducted on two datasets taken from the Walshaw archive [47] (3elt and vibrobox) and from the Facebook social network[2]. Figure 4.4 shows the convergence time in term of supersteps for the 3elt dataset with $K = 2$. Results with other datasets and different values of K are not included due to space constraints but they exhibit similar trends. The results show evidence that convergence is similar between GP-SPARK and GP-TELOS, as in both cases they achieve an almost-definitive edge-cut around the 400th superstep. In particular, after the 400th superstep GP-TELOS is stable, whereas GP-SPARK is converging but it is improving the result in every step marginally. Also, GP-TELOS yields a much better quality of results, achieving half the edge-cut of GP-SPARK.

Table 4.1 presents the edge-cut obtained by GP-TELOS and GP-SPARK, averaging the results of 5 runs. We executed multiple runs by varying the number of the graph partitions with the values $K = \{2, 4, 8, 16, 32, 64\}$. It can be noticed that GP-TELOS obtains a better edge-cut in all the datasets and in all the configurations but with the Facebook dataset and 8 partitions. However also in this configuration GP-TELOS provides an edge-cut similar (just 4% less) to the one in the GP-SPARK version. Overall, the GP-TELOS version provides a value between the 47% and the 91% better in the 3elt dataset and always better that the 30% in the Vibrobox dataset. These results suggest that the new layer helps improving the results, and a layered vertex-centric approach can be used to carry out graph processing computations.

4.3. Scalability. The aim of this experiment is to give a preliminary evaluation of Telos framework in managing large input graphs varying the number of cores involved in the computation. To this end, we built two Erdos-Renyi random graphs (1M vertices 5M edges the first, 500K vertices 1.5M edges the second) generated with the Snap library [23]. It is worth to point out that the current implementation of Telos is prototypical, thus our interest is more related to the viability of the approach than to the pure performance. To give this evaluation, we run the torus overlay experiment described in the previous section on both graphs, measuring the convergence time when a different amount of computing cores is used. We performed our experiments using both these graphs, varying the number of cores in the following range: $\{8, 16, 24\}$. For the sake of the presentation, values are normalised, independently for each graph, in the range [1 ; 100], with 100 being the highest execution time. In addition, each value is the average of 3 independent runs. The results are presented in Figure 4.5. With the graph composed of 500K vertices, our approach achieves only a little scalability and no benefit is obtained when the total amount of computing elements is greater than 16. Instead, when considering the larger graph our approach is able to achieve a decent scalability using up to 24 cores.

[2] http://socialnetworks.mpi-sws.org/

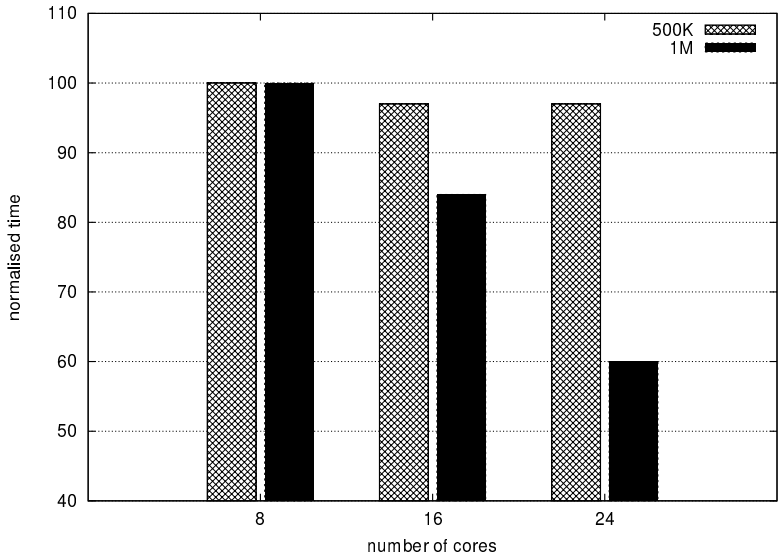


FIG. 4.5. *Scalability as a function of the number of cores*

5. Conclusion. This paper presents the approach adopted by Telos for programming of computations on large graphs. The aim of Telos is to propose an alternative to plain vertex-centric computations. The idea of Telos is based on defining overlay networks collectively built and managed by the nodes of the graph. Each overlay is specialised for a peculiar aim. The overlay-based approach takes inspiration from approaches that have proved to be robust and efficient in massively distributed systems that, somehow, recalls the structure of large graphs. We provided a detailed description of the concepts at the basis of Telos and its architecture, as well. We presented the Telos API, which has been designed to ease the programming effort required to program applications dealing with large graphs. In this way programmers can focus on the algorithm logic and to leave all the other burdens to the framework. We conducted an experimental analysis to validate the feasibility of the approach and showed its scalability with respect to the computational resources exploited. The experiments demonstrated that dynamic topologies can be effectively exploited during the computation. We believe that the ability of supporting multiple dynamic layers can be useful in many contexts, as for example in Peer-to-Peer applications [10, 39, 11] or for classical graph analysis problems, such as connected components [28, 25] and centrality measures [27]. As a future work we plan to conduct a comprehensive analysis of the performance of the framework and a comparison with other approaches targeting large graph computation. We also plan to implement Telos on top of different distributed frameworks (e.g., FastFlow [3], Akka [1]) to assess the performance of our proposed approach regardless the performances provided by Apache Spark.

REFERENCES

[1] Akka framework. `http://www.akka.io/`.
[2] M. ALDINUCCI, M. DANELUTTO, AND P. DAZZI. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2007.
[3] M. ALDINUCCI, M. DANELUTTO, P. KILPATRICK, AND M. TORQUATI. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2011.
[4] B. BACCI, M. DANELUTTO, S. ORLANDO, S. PELAGATTI, AND M. VANNESCHI. P3l: A structured high-level parallel language, and its structured support. *Concurrency: practice and experience*, 7(3):225–255, 1995.
[5] R. BARAGLIA, P. DAZZI, M. MORDACCHINI, L. RICCI, AND L. ALESSI. Group: A gossip based building community protocol. In *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 496–507. Springer Berlin Heidelberg, 2011.
[6] M. BASTIAN, S. HEYMANN, M. JACOMY, ET AL. Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 2009.
[7] E. CARLINI, M. COPPOLA, P. DAZZI, D. LAFORENZA, S. MARTINELLI, AND L. RICCI. Service and resource discovery supports over p2p overlays. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*,

pages 1–8. IEEE, 2009.
[8] E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci. Balanced graph partitioning with apache spark. In *Euro-Par 2014: Parallel Processing Workshops*, pages 129–140. Springer, 2014.
[9] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. Distributed graph processing: an approach based on overlay composition. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1912–1917. ACM, 2016.
[10] E. Carlini, P. Dazzi, M. Mordacchini, and L. Ricci. Toward community-driven interest management for distributed virtual environment. In *Euro-Par 2013: Parallel Processing Workshops*, pages 363–373. Springer, 2014.
[11] E. Carlini, A. Lulli, and L. Ricci. dragon: Multidimensional range queries on distributed aggregation trees. *Future Generation Computer Systems*, 55:101–115, 2016.
[12] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.
[13] M. D. P. Dazzi. A java/jini framework supporting stream parallel computations. 2005.
[14] P. Dazzi, P. Felber, L. Leonini, M. Mordacchini, R. Perego, M. Rajman, and É. Rivière. Peer-to-peer clustering of web-browsing users. *Proc. LSDS-IR*, pages 71–78, 2009.
[15] P. Dazzi, M. Mordacchini, and F. Baglini. Experiences with complex user profiles for approximate p2p community matching. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 53–58. IEEE, 2011.
[16] N. Doekemeijer and A. L. Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.
[17] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. *IEEE/ACM Trans. Netw.*, 14(3):479–491, June 2006.
[18] N. Javed and F. Loulergue. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In Y. Dou, R. Gruber, and J. Joller, editors, *Advanced Parallel Processing Technologies*, volume 5737 of *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin Heidelberg, 2009.
[19] D. Jeffrey and G. Sanjay. Mapreduce: Simplified data processing on large clusters. *Commununication ACM*, 1, 2008.
[20] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.
[21] V. Kalavri, V. Vlassov, and S. Haridi. High-level programming abstractions for distributed graph processing. *arXiv preprint arXiv:1607.02646*, 2016.
[22] H. Kavalionak and A. Montresor. P2p and cloud: a marriage of convenience for replica management. In *Self-Organizing Systems*, pages 60–71. Springer, 2012.
[23] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.
[24] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.
[25] A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci. Fast connected components computation in large graphs by vertex pruning. *IEEE Transactions on Parallel & Distributed Systems*, to appear.
[26] A. Lulli, P. Dazzi, L. Ricci, and E. Carlini. A multi-layer framework for graph processing via overlay composition. In *European Conference on Parallel Processing*, pages 515–527. Springer, 2015.
[27] A. Lulli, L. Ricci, E. Carlini, and P. Dazzi. Distributed current flow betweenness centrality. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 71–80. IEEE, 2015.
[28] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. Cracker: Crumbling large graphs into connected components. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 574–581. IEEE, 2015.
[29] 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*, pages 135–146. ACM, 2010.
[30] D. Margo and M. Seltzer. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment*, 8(12):1478–1489, 2015.
[31] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. *arXiv preprint arXiv:1404.3861*, 2014.
[32] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proc. of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.
[33] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, and D. Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.
[34] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Survey*, in press, 2014.
[35] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064. IEEE, 2015.
[36] M. Mordacchini, P. Dazzi, G. Tolomei, R. Baraglia, F. Silvestri, and S. Orlando. Challenges in designing an interest-based distributed aggregation of users in p2p systems. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*, pages 1–8. IEEE, 2009.
[37] U. Müller-Funk, U. Thonemann, and G. Vossen. The münster skeleton library muesli–a comprehensive overview. 2009.
[38] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification, 2004.
[39] A. H. Payberah, H. Kavalionak, A. Montresor, J. Dowling, and S. Haridi. Lightweight gossip-based distribution

estimation. In *Communications (ICC), 2013 IEEE International Conference on*, pages 3439–3443. IEEE, 2013.

[40] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60. IEEE, 2013.

[41] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[42] 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. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[43] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[44] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[45] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[46] S. Voulgaris and M. van Steen. Vicinity: A pinch of randomness brings out the structure. In *Middleware 2013*, pages 21–40. Springer, 2013.

[47] C. Walshaw. The graph partitioning archive, 2002.

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

[49] 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 *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Edited by: Frédéric Loulergue
Received: September 17, 2016
Accepted: March 11, 2017