ETSCH: Partition-centric Graph Processing

Alessio Guerrieri
DISI
University of Trento
Email: a.guerrieri@unitn.it

Alberto Montresor
DISI
University of Trento
Email: alberto.montresor@unitn.it

Simone Centellegher
ICT Doctoral School
University of Trento
Email: s.centellegher@unitn.it

***Abstract*—This paper presents ETSCH[1] , a novel paradigm for processing large graphs. ETSCH departs from the vertex-based approach of BSP frameworks like PREGEL in two ways: first, the units of computation are not the vertices, but rather a collection of subgraphs, obtained by partitioning the input graph; second, the subgraphs are obtained through an edge-partitioning algorithm, in which edges, rather than vertices, are subdivided into disjoint subsets Global computations over the graph are then easily expressed using classical centralized algorithms executed on each of the partitions, with the only additional burden of specifying simple reconciliation procedures when vertices are replicated in multiple computing nodes. The ETSCH paradigm has been implemented both on top of existing frameworks like HADOOP and SPARK, and as a stand-alone service based on AKKA, a toolkit for building distributed message-driven applications. When considering problems like single-source shortest path and PageRank, our experiments show that solutions based on ETSCH/HADOOP and ETSCH/SPARK already outperform the standard solutions to the same problems in HADOOP and SPARK, respectively. But it is our AKKA implementation that really shines: the execution time on graphs with millions of edges falls down from from thousands of seconds (ETSCH/HADOOP) to tens of seconds (ETSCH/SPARK) to seconds (ETSCH/AKKA), while easily scaling to graphs with billions of edges. ETSCH/AKKA is also faster than other partition-centric frameworks like BLOGEL and GPS.**

I. INTRODUCTION

The "big data" phenomenon has created the need to manage and analyze larger and larger datasets. The scale of such datasets keeps increasing exponentially, moving from gigabytes to terabytes and now even to petabytes. While parallel (multi-CPU, multi-core) systems have been used to deal with this deluge of data, there are many cases in which distributed approaches are the only viable road. A distributed system is able to cope with potentially unlimited datasets, is more robust to hardware failures, is often cheaper and, with the emergence of distributed frameworks for data analysis, is also much easier to use than it was a decade ago.

Since many of these large datasets represents connected entities, there has been strong focus on frameworks that are specialized on graph computation. PREGEL [1], the most influential of these frameworks, has been designed following the "think like a vertex" philosophy: each vertex is considered as a single processing unit, that receives information from neighbor vertices, computes a very simple function and sends up-to-date information back; all these steps are performed in periodic rounds. Vertices are thus assigned to a collection of *workers*. Each worker takes care of scheduling the vertex programs that are assigned to it and handles the communication between vertices.

While this is a very simple and elegant paradigm, it does not take into consideration that many interesting graph properties could be more easily computed by considering larger groups of vertices and edges, beyond the adjacency list of a single node.

In this paper we make the case for a different approach based on two ideas: edge partitioning and subgraph computing. The partitioning process happens along the edges and not the vertices, meaning that each partition is actually a collection of edges; the vertices associated to those edges may be replicated between distinct partitions. Each of the subgraphs is assigned to a different worker and each worker computes a function over the *entire subgraph*. The results obtained in each separate subgraph may be later reconciled into a global view by considering the vertices shared between different workers as *communication channels*.

The contribution of this paper is ETSCH, the first distributed graph processing framework that combines the ideas expressed above: computation is associated to partitions rather than vertices, and partitions are edge-disjoint rather the vertex-disjoint. Each machine will be responsible for a collection of edges, while communicating with other machines through vertices that appear in multiple partitions. A simple programming model is defined to support such abstraction. Experiments performed in the Amazon EC2 cluster show that ETSCH is highly scalable and outperforms or is on par to the most important distributed graph processing frameworks.

The rest of the paper is structured as follows: Section II presents an overview of the existing vertex-centric and partition-centric distributed frameworks. Section III presents the architecture of ETSCH, together with a few sample algorithms implemented on top of it. The experimental results are presented in Section V. The paper is concluded in Section VI.

II. RELATED WORK

The MapReduce programming model [2] has been introduced by Google to facilitate the development and execution of algorithms on very large quantities of data. This model inherits the map and reduce functions from functional programming to create a simple and inherently parallelizable programming model. While the MapReduce programming model has been proposed by Google, the most common open-source implementation is Apache's HADOOP [3]. While one of the original examples of MapReduce application was PageRank [2],

[1] "Etsch" is the German name for the river Adige, that flows through Trento. Its German pronunciation is similar to the English pronunciation of the word "edge".

MapReduce is not very efficient for graph analysis. PREGEL [1] was developed again by Google as an answer to these issues. In similar vein to the Bulk Synchronous Parallel (BSP) model [4], each iteration is composed of two phases, computation and communication, terminated by a single synchronization barrier. Each vertex is represented as a process, with knowledge of its own neighbors. In the first phase, the processes independently execute a number of computation steps and possibly issue messages to other processes. During the second phase, all the messages are sent across the network and delivered to the processes. The synchronization barrier makes sure that all vertices receive all messages sent to them during the current iteration before the start of the next one. During the computation phase each process updates its state by using the messages arrived during the previous iteration, sends messages to its neighbors and may vote to halt the computation.

SPARK [5] is a generic framework for large scale data processing based on *resilient distributed datasets* (RDD) that are transformed by applying functions such as map, reduce and join. If possible, the RDDs are kept in memory and only dumped to the distributed filesystem if needed, thus making it very efficient for iterative computation. GRAPHX [6], a specialized framework built over SPARK, efficiently stores and transforms very large graphs, while also offering a PREGEL-like interface for iterative graph algorithms.

A different approach is offered by the GRAPHLAB framework [7], with the asynchronous *Gather–Apply–Scatter* pattern (GAS). In the Gather phase each vertex receives messages from neighboring vertices which are aggregated in a single message that changes the local states in the Apply phase. Eventual changes are spread across outgoing edges in the Scatter phase. This approach has the distinct advantage of moving the computation from vertices to edges, thus allowing more flexibility and scalability in presence of high-degree vertices that can be split in different partitions.

This last feature has also been implemented in GPS [8], that introduced an optimization called large adjacency list partitioning (LALP), which partitions the outgoing edges of high degree vertices in different workers. GPS has also some support for global computation and global objects, which can help speed up the convergence for some graph algorithms.

Frameworks that are most similar to ETSCH are those that allow partition-centric computation, in which iterative programs are executed by each machine on the entire partition assigned to it. Both GIRAPH++ [9] and BLOGEL [10] allow users to execute their algorithms on the entire partition. GIRAPH++ then sends messages from boundary vertices (those that are connected to the outside of the partition) across different workers. GIRAPH++ also offers a third interface that allows changing the graph, useful for problem such as graph coarsening and summarization. BLOGEL supports different modes of execution. It executes vertex-centric algorithms, block-centric algorithms and even hybrid algorithms, in which all vertices execute before the entire block. These modes of execution allow BLOGEL to offer a very fast implementation of PageRank that first operates in block mode, initializing the PageRank of the vertices using only local information and the connection between blocks, and then execute the classical vertex- centric PageRank algorithm, which will converge faster than normal thanks to the better starting values.

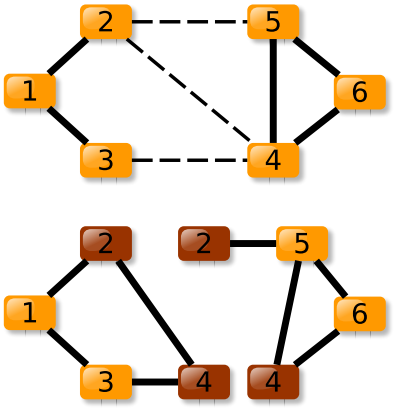


Fig. 1: Example of vertex and edge partitioning: on the top figure, vertices are partitioned and a few *cut edges* connect vertices belonging to different partitions. On the bottom figure, edges are partitioned and a few *frontier vertices* appear in more than one partition

III. ETSCH

Figure 1 shows the differences between partitioning the vertex set and partitioning the edge set (*vertex partitioning* and *edge partitioning*).

When a graph is subdivided using a vertex-partitioning algorithm, each subgraph has a number of external edges that connect vertices across partitions. These are called *cut edges* and are not really part of the subgraph, since the partition does not have knowledge of the other endpoint of the edge. The approach in this case is to consider vertices as computational entities that "send" messages to their neighbors, potentially across partitions using cut edges.

This is not the case with edge partitioning. Both vertices and edges of a local graph can be associated with local state. Edges are part of exactly one subgraph, therefore their states belong exactly to one partition. The same happens with vertices that are not replicated. The nodes that are replicated in different partitions are called *frontier edges*; their state need to be periodically reconciled.

Figure 2 shows the organization of ETSCH. First of all, the graph is decomposed into K partitions by an edge-partitioning algorithm. Each partition is assigned to a different *worker*, which executes the following steps:

1) The *initialization phase* is run once, by taking the subgraph representing the partition as input and initializing the local state of vertices and edges.
2) Once completed the initialization, each subgraph state is fed to the *local computation phase*, that runs an independent instance of a sequential algorithm that updates the local state of the subgraph.
3) The *aggregation phase* logically follows the local computation: for each frontier vertex, the framework collects the distinct states of all replicas and computes a new state, that is then copied into the replicas.

Step (2) and (3) are executed iteratively, until the desired goal

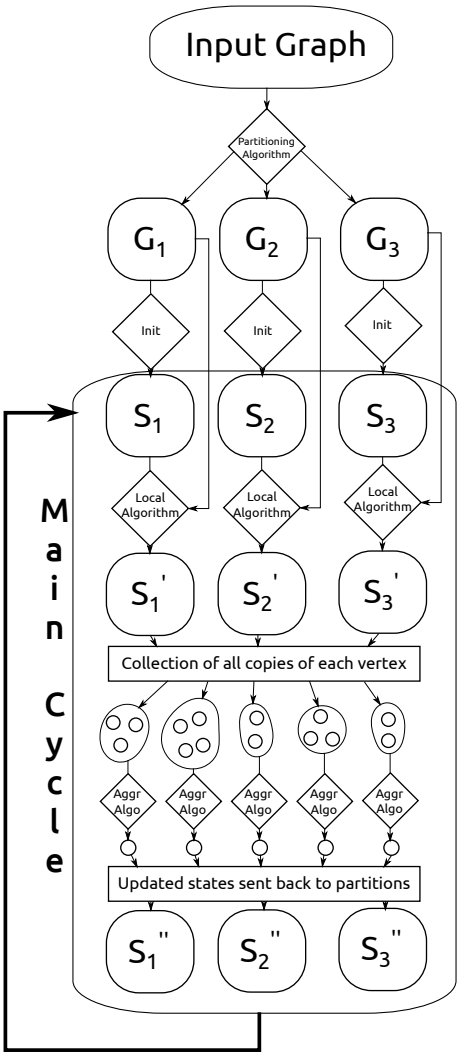


Fig. 2: Illustrative schema of ETSCH

is reached and the distributed algorithm has completed its goal.

In order to use ETSCH, three functions corresponding to the three phases must be implemented. Functions init() and localComputation() take a subgraph as input and perform their computation on it. aggregation() takes an array of replica state (whose type is defined by user) and should return a single state that reconcile those contained in the array. The framework takes care of calling init() and localComputation() in each of the worker, and provides them with a subgraph to be computed; it then collects the replicated states from the replicas, calls aggregation() on them and then copy the aggregated state back to the replicas.

A. Application examples

Algorithms 1 and 2 provide a couple of application examples; the former shows how to compute the distances of vertices from a source vertex, while the latter shows how to identify the connected components of a graph using ETSCH.

For the problem of distance computation (Algorithm 1), each vertex is associated with a state containing just the distance variable *dist*. Initially, all vertices are initialized to $+\infty$, apart from the *source* vertex which is initialized to 0. In the local computation phase, the vertices distances are updated by executing the Bellman-Ford algorithm until no changes are made. In the aggregation phase, replicated states of vertices are represented as a vector of distances, from which the minimum distance is taken.

Algorithm 1: Distance computation

```
function init(SubGraph G_i)
    foreach v ∈ G_i.V do
        if v = source then
            v.dist = 0;
        else
            v.dist = ∞;

function localComputation(SubGraph G_i)
    changed = true;
    while changed do
        changed = false;
        foreach e ∈ G_i.E do
            s = e.sourceVertex;
            d = e.destVertex;
            if s.dist + 1 < d.dist then
                d.dist = s.dist + 1;
                changed = true;
            else
                if d.dist + 1 < s.dist then
                    s.dist = d.dist + 1;
                    changed = true;

function Distance aggregation(Distance[] D)
    return min(D);
```

This approach decreases substantially the number of iterations needed by the framework to complete its execution. If the shortest path between a node and the *source* passes through K partitions, that node will have the correct distance from *source* after only K iterations. Since the length of a path is an upper bound to the number of partitions traversed by that path and a vertex-centric algorithm can only move by one edge during each iterations, our approach can reduce the amount of iterations needed to converge. Comparing the number of iterations needed by the vertex-centric approach against our partition-centric approach, we measured a 30% decrease on small-world graphs and over 95% decrease on road network with large diameter.

The algorithm for computing the connected components uses a variation of Dijkstra's Algorithm (Algorithm 2). Each vertex is associated with a connected component identifier *id*, which is generated randomly for each vertex. The local computation phase epidemically spread the smallest component identifier by passing it through the local edges, until all vertices have been reached. In the aggregation phase the smallest identifier is selected from all the replicas and returned as their connected component identifier. Eventually, each connected component will be identified by a single value, which is

Algorithm 2: Connected components computation

```
function init(SubGraph G_i)
    foreach v ∈ G_i.V do
        v.id = random();

function localComputation(SubGraph G_i)
    PQ = new PriorityQueue⟨Vertex⟩();
    foreach v ∈ g_i.V do
        PQ.add(v);
    while not PQ.() do
        q = PQ.pop();
        foreach v ∈ q.neighbors do
            if v.id > q.id then
                v.id = q.id;
                PQ.update(v);

function aggregation(ID[] D)
    return min(D);
```

Algorithm 3: Gather-Apply-Scatter

```
function localComputation(SubGraph G_i)
    foreach (u, v) ∈ g_i.E do
        D_{u,v} = scatter(D_u, D_{(u,v)}, D_v)
    foreach u ∈ G_i.V do
        foreach v ∈ u.neighbors do
            u.Accum =
            sum(u.Accum, gather(D_u, D_{(u,v)}, D_v));

function aggregation(Accum[] a, D curstate)
    Accum cur = nil;
    foreach i ∈ a do
        cur = sum(cur, i);
    return apply(cur, curstate);
```

the smallest identifier randomly generated in each connected component.

Both are basic problems that can be used as building blocks for other, more complex computations. For example, the problem of distance computation is needed to compute properties like betweenness centrality [11]. It is also possible to implement Luby's maximal independent set algorithm [12] in ETSCH, by spreading the random values in the local phase and choosing if a vertex must be added to the set in the aggregation phase.

B. Applicability of Etsch

As a general guideline, problems that need several iterations to complete in a vertex-centric framework are the ones that can gain the most from a partition-centric framework such as ETSCH. In other problems, such as computing the number of triangles in a graph or solving PageRank, most algorithms need only local computation and therefore a vertex-centric interface can allow simpler algorithms. For these reasons we implemented the *gather–apply–scatter* model on top of ETSCH, as shown in Algorithm 3. The *scatter* phase is executed in the local phase and the messages sent inside the partitions can already be *gathered* and collected. The ETSCH aggregation phase will collect the messages sent to that node from different partitions and *apply* it to the state of the node.

This additional module will allow users to run a program written in GAS or following ETSCH programming model, while following the same partitioning. Some of the experiments presented in Section V has been obtained by running the standard PageRank algorithm on top of this *gather–apply–scatter* module (see [13] for the pseudocode).

C. Partitioning schemes

The quality of the partitioning chosen for the execution of ETSCH can have a huge impact on the efficiency of the system and deserves some careful consideration. The most important metric to consider to evaluate the quality of such partition is the number of frontier nodes, the vertices that appear in more than one partition and thus cause the creation of replicas and the need of the aggregation phase. A smaller amount of frontier nodes affects the execution of ETSCH in different ways:

- Workers need less memory to store all the replicas that refers to the copies of the frontier vertices
- ETSCH sends less messages to reconciliate the replicas of each node
- The aggregation phase of ETSCH is shorter, since there are less nodes on which it has to execute.

While having less frontier nodes is always better, there is a trade-off between the time spent to improve the partitioning and the time needed by ETSCH to complete its execution. It is clearly not a good idea to run a complex, slow partitioning algorithm if the algorithm to be run is not very time-consuming, or if the system will spend more time in partitioning the graph than in analyzing it.

There are many very fast hash-based random partitioning algorithms that can be used when the quality of the partitioning is not very important, but they tend to create extremely disconnected partitions and a huge number of frontier nodes. ETSCH can still be executed on these partitions but the advantages of a partition-based framework might disappear.

A better compromise is POWERGRAPH's random vertex cut [13], a greedy algorithm that computes a good edge partitioning by following few simple rules for each edge. If there is a need of a more connected partition, DFEP [14] is a diffusion-based algorithm that, while slower, will allow ETSCH to run more efficiently. JA-BE-JA [15] needs even more iterations to converge, but computes extremely balanced partitions with good connectedness.

In many cases, the graph comes with additional information about its vertices and edges. A web graph can contain the URL of the pages, a road network can contain information about the coordinates or the type of connections, collaboration networks can contain data about the people involved. This information can be exploited to obtain high quality partitions very quickly. In our experiments in Section V that involve web data, we used a variant of the URL partitioning algorithm offered by

BLOGEL [10] that groups together all out-edges of each web domain and then assign greedily each group to the less loaded partition. We found that the fraction of frontier nodes in web graphs can be very few (around 2% in our experiments) and therefore the amount of bandwidth needed is very small. We estimate less than 7Mb of data was sent around the network for each iteration of PageRank of ETSCH on the ARABIC graph, which contains over 20 million nodes.

IV. Implementation details

This Section describes the three different implementations of ETSCH that are compared in Section V. Each of these implementations can give some insight in the advantages and disadvantages of the system that has been used to develop them.

A. Hadoop

ETSCH has first been implemented as a Map-Reduce job on top of Apache HADOOP. HADOOP has arguably the largest user base between all big-data analysis frameworks, and its implementation, while not very efficient, is very stable without any unpredictable behavior.

Because of the limitations of the Map-Reduce model, two iterations of HADOOP must be executed for each super-step of ETSCH: one Map-Reduce program for the local computation phase and one for the aggregation phase. In each of these steps, the entirety of the graph must be loaded from scratch from the distributed file system, thus incurring in significant overhead.

The termination condition is controlled by the driver program, which checks the appropriate HADOOP counters to see if ETSCH's phases have modified the state of any node in the graph.

B. Spark

The second implementation has been developed in Apache SPARK/GRAPHX, a faster and more flexible alternative than HADOOP. What made GRAPHX especially intriguing is the fact that it uses edge partitions and therefore is already close to our model.

In particular GRAPHX's implementation of the graph offers a *mapTriplets* function that executes an user defined function on the entirety of a partition. For our scope we extended that function, enveloping the partition in a custom class that keeps track of changes in the states of the nodes. The entirety of the ETSCH pipeline is then implemented as a sequence of API calls, with accumulators used to count state changes in the entirety of the system.

For this version of ETSCH, the implementation had been written in Scala, to test the efficiency of SPARK in its most publicized language. This choice made initial coding much faster, but also created difficulties in debugging because of the immutability of Scala's objects. This characteristics also caused our implementation to waste time in converting data from immutable to mutable objects to make them accessible and editable by the user defined functions.

C. Akka

AKKA [16] is a Java framework for parallel and distributed Actor-based programming. AKKA offers the possibility to create generic *actors*, stored using very few memory, that can react asynchronously to incoming, user-defined messages. While the absence of graph-specific services meant more work to implement ETSCH, it allowed us to avoid using work-arounds to overcome the other frameworks' limitations.

Our ETSCH implementation creates one single actor for each worker and one actor as a master. Currently the master actor functions only as a check-in point for the actors to discover the system and does not do any computation. The worker actors store their partitions and execute the algorithms defined on top of ETSCH programming model.

At startup, each worker reads its input partition independently. Since in the data there is no information about which nodes have replicas and in which partitions they reside, the workers needs to spend a communication phase and exchange the list of nodes. Each frontier node is then assigned to a random partition, which will receive updates to its state from the other partitions and eventually execute the aggregation program.

During the execution phase, each worker/partition executes the local computation phase independently and, once it has finished, sends a message to each of the other workers with the list of changed states. Once a worker has received updates from all workers, it will move to the aggregation phase.

Our current implementation does not yet use AKKA's services for obtaining fault-tolerance and load-balancing and assumes failure-free executions. While we plan to extend our framework to deal with failures in a future work, we underline that even the largest datasets tested in this section are analyzed in less than one minute, and thus it is perfectly feasible to re-start the entire job in case of failures.

Internally, each partition is stored as a single sorted edge list with pointers to the outgoing edges of each node. This approach allowed a more efficient representation of the graph in memory, without the large overhead of efficiency caused by the use of Java's standard collection. The graph is effectively stored in a low-level array of longs thus decreasing the impact of memory issues on the execution.

V. Results

ETSCH has been implemented on top of three different frameworks for distributed computation, HADOOP, SPARK and AKKA. This section compares the three implementations of ETSCH against native vertex-centric algorithm in the respective frameworks. A final comparison shows that our AKKA implementation of ETSCH is orders of magnitude faster than the other implementation and obtains generally better results with respect to competitor frameworks such as BLOGEL and GPS.

All these experiments have been executed using Amazon AWS. The machines used in the experiments are `m3.large`, equipped with High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors (2 virtual cores) and 7.5 GB of RAM.

Name	$\|V\|$	$\|E\|$	$\varnothing$	d_{max}
DBLP	317,080	1,049,866	21	343
YOUTUBE	1,134,890	2,987,624	20	28754
AMAZON	400,727	2,349,869	18	9905
ROADNET-PA	1,087,562	1,541,514	784	9
LIVEJOURNAL	3,997,962	34,681,189	16	14815

TABLE I: Datasets used with ETSCH/HADOOP and ETSCH/SPARK

Name	$\|V\|$	$\|E\|$	$\varnothing$	deg_{max}
INDOCHINA	7,414,866	194,109,311	28.12	6985
UK-2002	18,520,486	298,113,762	21.59	2450
ARABIC	22,744,080	639,999,458	22.39	9905
UK-2005	39,459,925	936,364,282	23.19	5213

TABLE II: Datasets used with ETSCH/AKKA

A. Datasets

In our experiments, we used different datasets to test the performance of our framework on different problems. The five graphs that are used in our implementation of ETSCH in SPARK and HADOOP are presented in Table I. DBLP is the co-authorship network from the DBLP archive, YOUTUBE is the friendship graph between the users of the service while AMAZON is a co-purchasing network of the products sold by the website. Finally, ROADNET-PA is a large-diameter graph representing a road network and LIVEJOURNAL is a large social network graph with a small diameter. All these networks have been taken from the SNAP graph library [17] and cleaned for our use, making directed edges undirected and removing disconnected components. These datasets do not contain any additional information outside of the structure of the graph and have been partitioned using DFEP [14].

The larger datasets used in AKKA, presented in Table II, have been downloaded from the Laboratory of Web Algorithmics of the University of Milan[2]. Such datasets are compressed via LLP [18] and WebGraph [19] and contain additional information about the domain of the web page. We partitioned them using a variant of BLOGEL's URL partitioner based on edges rather than vertices.

B. Hadoop

To test the practical advantages of ETSCH we first prepared a HADOOP implementation of the framework in which the user can define the three functions as defined in Section III. We compared this approach against running a baseline vertex-based implementation of the shortest path algorithm on the unpartitioned graph, in which every vertex sends messages in the Map phase and receive them in the Reduce phase. Figure 3 shows that our approach is much more efficient when the number of processing nodes is small, since the partitions are larger and paths are more easily compressed. When the number of partitions grows, the baseline approach gets closer to ETSCH, but the latter is still more efficient. This implementation, while not very efficient with later ones, is very stable without any unpredictable behavior.

[2]http://law.di.unimi.it

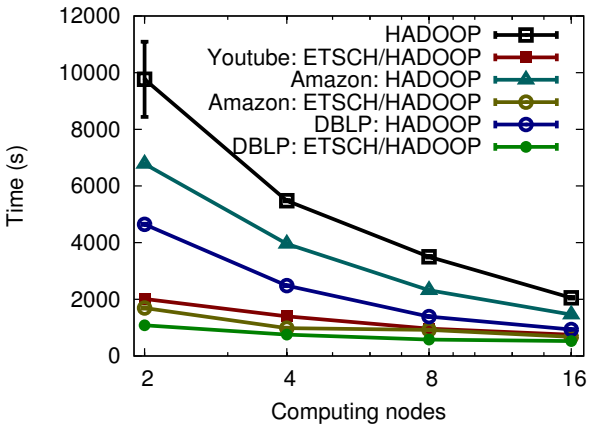


Fig. 3: Running time of single source shortest path algorithm in HADOOP comparing a standard baseline algorithm and ETSCH, with `m3.large` machines on EC2.

C. Spark

As the experimental results show, SPARK is much more efficient but is still unstable and we encountered strange behaviors when the memory available is smaller than optimal. This is expected since SPARK makes a larger use of the memory than HADOOP.

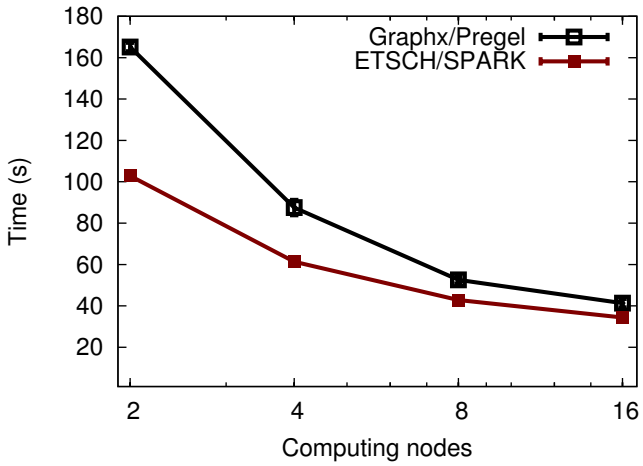
As before, we compared the single-source shortest path algorithm implemented on ETSCH with the standard PREGEL approach, following the example presented in the GRAPHX programming guide. Given the large speedup obtained by moving from HADOOP to SPARK, we were able to use the last two datasets from Table I as well.

We show the experimental results in Figure 4. In the LIVEJOURNAL graph we can see that our approach is faster, but the greater the number of partitions the smaller is the speedup caused by processing each partition as a subgraph. The ROADNET-PA graph shows that, as expected, when the diameter is huge ETSCH is extremely efficient. While the PREGEL version needs hundreds of iteration to complete, ETSCH finishes in only few iterations thus decreasing the synchronization overhead.

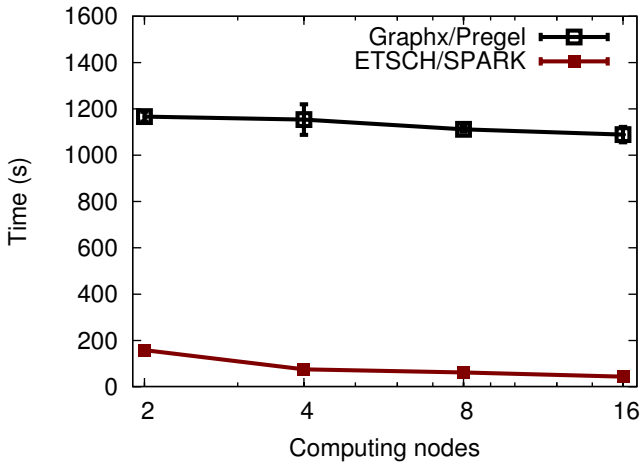
D. Akka

AKKA's implementation has been tested on the larger web graphs, therefore in the rest of the section we focus on the PageRank problem and we compare ETSCH, BLOGEL and GPS against it. We believe that the advantages of a partition-centric approach in problems such as single-source shortest path have been already shown by the previous sections and by results in [10] and [9]. For ETSCH, we implemented the standard PageRank algorithm on top of the Gather-Apply-Scatter module presented in Section III.

Figure 5 shows the scalability of our approach, running PageRank on the 4 large datasets in Table II on different numbers of `m3.large` machines. ETSCH can analyze even the largest, 1–billion edges dataset with only 4 machines, but the addition of more machines allow the system to use the memory more efficiently and thus significantly speed up the



(a) LIVEJOURNAL



(b) ROADNET-PA

Fig. 4: Comparison between ETSCH and the standard PREGEL implementation in SPARK/GRAPHX

Dataset	ETSCH/ HADOOP	ETSCH/ SPARK	ETSCH/ AKKA
DBLP	755s	7.8s	1.75s
YOUTUBE	1400s	11.1s	2.45s
AMAZON	984s	12.2s	2.33s
ROADNET-PA	NA	75.1s	1.43s
LIVEJOURNAL	NA	61.3s	8.78s

TABLE III: Comparing different frameworks for ETSCH, using 4 machines.

Dataset	ETSCH/AKKA	BLOGEL	GPS
INDOCHINA	$0.94s \pm 0.01s$	$4.51s \pm 0.01s$	$10.37s \pm 0.17s$
UK-2002	$2.33s \pm 0.05s$	$3.44s \pm 0.02s$	$8.11s \pm 0.18s$
ARABIC	$5.12s \pm 0.05s$	$8.49s \pm 0.04s$	$15.42s \pm 0.09s$
UK-2005	$11.90s \pm 0.18s$	$10.91s \pm 0.08s$	$23.51s \pm 0.05s$

TABLE IV: Comparison of running time of a single PageRank iteration in ETSCH, BLOGEL, GPS (8 `m3.large` machines)

computation. Using 8 machines on the UK2005 causes a 2.37 speedup with respect to the same experiment with 4 machines. Investigating this result indicates that when the number of edges is too big, the system is slowed down by Java's garbage collector. Moving away from Java's collections and using ad-hoc, array-based data structure might be a solution that we will explore in future works.

Even with these memory issues, our AKKA implementation is order of magnitude faster than implementations on other frameworks, as seen in Figure III

E. Comparison with Blogel and GPS

Comparison against other frameworks is shown in Table IV. Given that in most datasets we were not able to run BLOGEL and GPS on only 4 machines, we decided to run the following experiments using 8 `m3.large` machines in EC2. We executed BLOGEL's URL partitioner on the datasets and then ran both BLOGEL and GPS on the same partitioned graph. Since GPS implements only a round-robin scheme, we exploited the BLOGEL URL partitioning scheme and assigned node identifiers so that GPS's round-robin distribution reflects the URL partitioning. For each of them we measure the running time of the vertex-centric PageRank part of computation, discarding the setup phase, and we divided it by the number of iterations needed to converge to measure the average running time per iteration.

Aside from UK-2005, where some memory issues with the Java's garbage collector make ETSCH slightly slower than BLOGEL, our approach is significantly faster. An interesting behavior can be noticed in the processing of INDOCHINA. Despite the smaller size of that dataset with respect to UK-2002, both BLOGEL and GPS take more time to complete a PageRank iteration. We investigated the problem and we found that the reason is that the partitions created by BLOGEL are balanced with respect to the number of vertices, but not with respect to the number of edges. This is a limitation of vertex-partitioning schemes: since we are dealing with graphs with a power-law degree distribution, the number of edges inside the partitions can be wildly different even if the number of the nodes stays the same. In the case of INDOCHINA, BLOGEL created a partition that has 4X more edges with respect the other partitions, thus slowing the system significantly.

GIRAPH++'s implementation of PageRank has been executed on UK-2002 and UK-2005 by the authors in their paper. Their reported results (respectively 4 and 13 seconds) are worse than what both BLOGEL and ETSCH/AKKA obtain; furthermore, note that they used more resources (10 quad-core machines with 32G of ram against 8 `m3.large` machines).

VI. FUTURE WORK

As future work, we plan to study the ETSCH framework both from a theoretical and a practical point of view. We plan to investigate which type of graph problems are solvable in ETSCH and which ones need a completely different framework. For some problems, the classical solutions could be easily

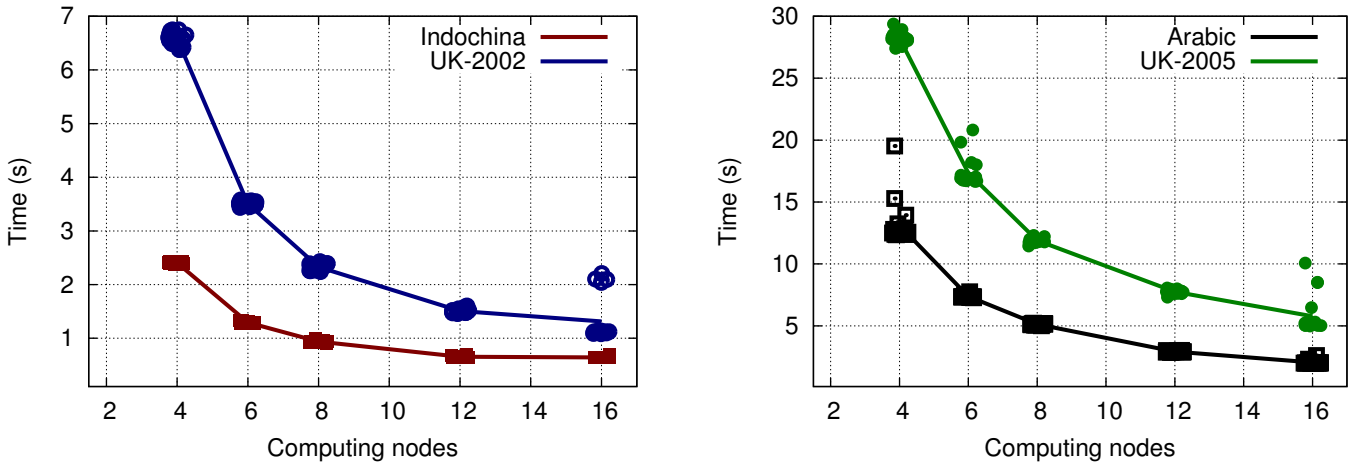


Fig. 5: Scalability of ETSCH/AKKA on the larger datasets, using `m3.large` machines on EC2. For each experiment we measure the average time for a complete iteration of PageRank

translated into ETSCH, while for others novel algorithms could be needed. On the technical side we plan to add fault tolerance and to the system, by exploiting AKKA's features, and make it usable by general users. We also want to add the partitioning algorithm to the system, to allow users to download, partition and analyze a large graph with a single command. An even more exciting project would be to extend ETSCH programming model to avoid the need for synchronization. In this extended frameworks both replicas and partitions would be able to execute in random order, thus hopefully allowing less waiting time and more efficient analysis of the graph.

REFERENCES

[1] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of the 2010 international Conference on Management of data*. ACM, 2010, pp. 135–146.

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

[3] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," http://lucene.apache.org/hadoop, 2005.

[4] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[5] 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*. USENIX Association, 2012, pp. 2–2.

[6] 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*. ACM, 2013, p. 2.

[7] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[8] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[9] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[10] 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*, vol. 7, no. 14, pp. 1981–1992, 2014.

[11] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[12] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[14] A. Guerrieri and A. Montresor, "DFEP: Distributed funding-based edge partitioning," in *Proc. of the 21st Conference on Parallel Processing (Euro-Par'15)*, ser. Lecture Notes in Computer Science. Springer, 2015.

[15] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "Ja-be-ja: A distributed algorithm for balanced graph partitioning," in *Proc. of 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'13)*. IEEE, 2013, pp. 51–60.

[16] "Akka," http://www.akka.io.

[17] J. Leskovec, "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, 2011.

[18] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International World Wide Web (WWW'11)*. ACM Press, 2011.

[19] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the 13th International World Wide Web Conference (WWW'04)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.