An Evaluation Study of BigData Frameworks for Graph Processing

Benedikt Elser, Alberto Montresor
Università degli Studi di Trento, Italy
{elser|montresor}@disi.unitn.it

***Abstract*—When Google first introduced the Map/Reduce paradigm in 2004, no comparable system had been available to the general public. The situation has changed since then. The Map/Reduce paradigm has become increasingly popular and there is no shortage of Map/Reduce implementations in today's computing world. The predominant solution is currently Apache Hadoop, started by Yahoo. Besides employing custom Map/Reduce installations, customers of cloud services can now exploit ready-made made installations (e. g. the Elastic Map/Reduce System).**

In the mean time, other, second generation frameworks have started to appear. They either fine tune the Map/Reduce model for specific scenarios, or change the paradigm altogether, such as Google's Pregel. In this paper, we present a comparison between these second generation frameworks and the current de-facto standard Hadoop, by focusing on a specific scenario: large-scale graph analysis. We analyze the different means of fine-tuning those systems by exploiting their unique features. We base our analysis on the k-core decomposition problem, whose goal is to compute the centrality of each node in a given graph; we tested our implementation in a cluster of Amazon EC2 nodes with realistic datasets made publicly available by the SNAP project.

I. Introduction

A key myth of American folklore is the garage as birthplace of start-ups. Google is a famous example: the company scaled from a garage in Menlo Park to a search provider indexing more than 45 billion pages[1]. To minimize their initial hardware costs, they exploited clusters of commodity hardware instead of investing into large supercomputers. To develop novel parallel algorithms, engineers at Google soon realized that they needed both a convenient way of programming them and a framework to orchestrate their execution. These requirements resulted in the design of the Map/Reduce platform [7]. Used everywhere inside Google [6], Map/Reduce received a lot of attention from scientific and business computing since its first publication in 2004. The Map/Reduce model was later reimplemented in the open-source framework Apache Hadoop [25].

The availability of Map/Reduce frameworks and the abundance of cloud-provided resources (e. g. Amazon's EC2 [1]) made the analysis of large quantities of data available to everyone and started what is by now called the "big data" movement. Today, everyone can start a business based on the processing of large amounts of data, without being overwhelmed by the initial investment. Using these techniques, it is now possible to analyze datasets so large that could have never been processes without a supercomputer.

Initially, Map/Reduce has been proposed for very simple, embarrassingly parallel tasks – like log analysis – applied to very large and distributed datasets [7]. Later, an increasing number of papers tried to apply Map/Reduce to a larger set of problems, including machine learning [15], joining complex dataflows [4], [27], and analyzing very large-scale graphs [11].

Linked data and graphs structures, in particular, have acquired more and more importance in today's data world. In fact, graphs are omnipresent in the Internet and in our lives: as examples, take the connections between friends, the co-authorship relation originated from scientific papers, the dependencies between providers and suppliers in today's complex business world. Each of them can easily be reduced to a network of vertices and edges.

The usage of Map/Reduce beyond its original design has generated a large amount of criticisms against such an abuse [15]; this has inspired both the adoption of specific programming patterns that optimize the analysis of large linked data structures in Map/Reduce [13], and the creation of entirely new frameworks based on alternative programming models [17].

Contribution In this paper we present, to the best of our knowledge, the first evaluation study of modern big data frameworks (Map-Reduce [], Stratosphere [], Hama [], Giraph [], Graphlab []) in terms of their applicability to graph processing. We include their latest technological developments and test them in a real world setting based on an EC2 cluster. As a testing problem, we adopt the k-core decomposition of large graphs, whose goal is to compute the centrality of each node by identifying the maximal induced subgraphs including that node. We ported a distributed k-core algorithm [19] to each of the considered frameworks, optimizing it to exploit the special benefits of each of them. Furthermore, we compare the results obtained in a cluster with those obtained in a single multicore machine based on shared memory. Finally, we discuss the fault resilience of each framework. The rest of this paper is organized as follows: Section II introduces the k-core algorithm. Section III gives an overview of Map/Reduce and graph-centric programming models. Section IV describes current frameworks and their improvements over Map/Reduce. Section V discusses the most interesting results from our analysis. Section VI concludes with an outlook to future work.

[1] http://www.worldwidewebsize.com/

II. K-Core

The test algorithm we choose to implement is a distributed implementation the of *k-core decomposition* problem [19], whose goal is to determine the relative importance of a node by identifying the maximal induced subgraphs of a graph. k-core decomposition has been applied to a number of problems; for example it has been used to characterize social networks [22], to help in the visualization of complex graphs [2] or to determine the role of proteins in complex proteinomic networks [3].

Informally, a k-core is obtained by recursively removing all nodes with degree smaller than k, until the degree of all remaining vertices is larger than or equal to k. Consider Figure 1 as an example. Clearly, by definition, each k-core includes the larger cores, i.e. the $(k+1)$-core, the $(k+2)$-core, etc. In the figure, the 3-core is nested in the 2- and 1-core. Apparently, larger values of k correspond to more central positions in the network.

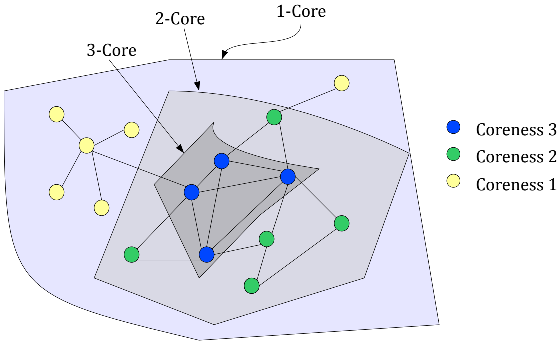

Figure 1. k-core decomposition for a sample graph [19].

Algorithm 1 illustrates the distributed algorithm; refer to [19] for an in-depth analysis. Initially, each node u sets its own k-core value, stored in *core* to its degree and sends it to all its neighbors ($neighbors_V(u)$). Upon receipt of a recent value k from node v, node u recomputes its own value and broadcasts it to its neighbors if it has changed. The *computeIndex*() method returns the largest value i, such that there are at least i entries equal or larger than i in *est*.

Filtering To minimize the interprocess communication we introduced an optimization to our k-core algorithm, referred as *filtering* in the rest of the paper. Instead of sending the changed value to every vertex in the neighborhood, we relax this constraint: neighbor vertices that have a smaller k-core value than the current computed one do not require an update. This is because the computation of the value is monotonically decreasing, starting at the $degree(u)$ of node u and stopping only when there is a fixed set of k nodes with the k-core value equal to or larger than k. That computation does not depend on the actual value of the neighbors coreness, as long as their value is higher. Even if vertices only know the initial value of a neighbor v, which is $degree(v)$, they will still correctly calculate their own value. As edge degree typically follows a power-law distribution, reducing communication from these nodes makes a huge difference (cf. section V-C4).

Algorithm 1: Distributed algorithm to compute the k-core decomposition; routine executed by node u.

```
on initialization do
    changed ← false;
    core ← $d(u)$;
    foreach $v \in neighbor_V(u)$ do  $est[v] \leftarrow \infty$;
    send $\langle u, core \rangle$ to $neighbor_V(u)$;

on receive $\langle v, k \rangle$ do
    if $k < est[v]$ then
        $est[v] \leftarrow k$;
        $t \leftarrow$ computeIndex($est, u, core$);
        if $t < core$ then
            $core \leftarrow t$;
            changed ← true;

repeat every $\delta$ time units (round duration)
    if changed then
        send $\langle u, core \rangle$ to $neighbor_V(u)$;
        changed ← false;
```

III. Background

Map/Reduce is a data-parallel batch system, able to process large volumes of data using a simple programming model. The main features of the framework include the queuing, distribution and parallelization of the computation across a cluster of machines in a resilient way. It parallelizes the user-provided implementations of the map, reducefunctions by partitioning the input data and feeding it to these functions. The framework also takes care of the shuffling and sorting of partial solutions, as well as distribution issues and error handling, such as restarting of jobs. Therefore, the programmer is merely required to express her algorithm as a sequence of these function calls. The mapfunction turns input data, which is not required to have any specific structure or relation, into a collection of key-value records. These records are aggregated by keys into bins; each bin is processed by one instance of the reducefunction. Initially proposed by Google [7], there are now several alternative implementations, the most prominent one being Apache Hadoop [25].

The algorithms proposed in the initial paper of Map/Reduce do not focus on linked data; instead, problems requiring a large amount of independent computations were discussed, such as the distributed `grep` problem. However, the framework can be applied to a variety of problems. For example, iterative graph algorithms can be expressed as a series of Map/Reduce runs, where each run produces an intermediate solution [5]. The same steps can be applied repeatedly, by using intermediate solutions as the input data, until the final solution is computed. The problem is that each run must include the complete information of the input graph as well, ignoring even basic data persistence potential. An overview of these and other problems related to the processing of graphs in Map/Reduce can be found in [16].

It is important to notice that by design, Map/Reduce is optimized for reliability, not for execution time. Each node can leave the cluster at any time, without job failures, which is due to a very conservative approach to redundancy. For example, after each computation step, such as map, the intermediate result is written back to a distributed file systems, to be easily recoverable. Furthermore, holding state on a node is discouraged, which on the one hand eases the handling of node failures, but on the other hand requires each computation to provide all required data as input. Therefore, normally each iteration is dominated by the data-transfer costs.

A. Mitigations

Typical graph analysis problems are characterized by small outputs (a set of $\langle vertex\_id, result\_type \rangle$ tuples, or even a single value) with respect to the size of the input (which is normally a huge graph). Thus, repeatedly taking the graph as input and storing large intermediate results has a dramatic impact of a Map/Reduce job's execution time. Therefore, patterns applicable to graph algorithms have been proposed [13], in order to reduce the total number of messages sent.

One frequently employed technique is *grouping*: instead of sending data directly to a vertex, a grouping function merges multiple vertices into a single "pool" vertex. In the subsequent reduce, all pooled vertices will be computed together. In contrast to standard Map/Reduce, the computation will be able to access all already computed values from the pool, which reduces the need of communication between nodes.

To mitigate the amount of graph structure information that is sent across the network, the *schimmy pattern* has been proposed. The idea is that the relevant structure information is cached on each node in the cluster and accessed by the user reducecode via shared memory. To cache only relevant information, the framework function that maps vertex ids to cluster nodes must be appropriately customized.

Both methods, among others, have been analyzed in [13]. While grouping did not yield a significant difference in run-time, the authors found the schimmy pattern to reduce the run-time significantly.

However, both methods have been criticized in [21] for destroying the resilience that comes with Map/Reduce jobs.

B. A novel paradigm

Google's success stems from their search product, which relies on an iterative graph algorithm that ranks pages based on their importance called *PageRank*. However, by design, the Map/Reduce system is stateless, hence all information about both nodes and their current rank, as well as their edges needs to be transferred in the cluster over and over. Despite being a suboptimal choice for graph computation, the programmer's focus on a single local action that is automatically scaled to a large dataset by the framework was appealing.

Hence, Pregel [17] – a graph-parallel system based on the Bulk Synchronous Parallel [26] (BSP) model – was proposed. Pregel is a *vertex-centric* approach: a programmer implements an iteration step (superstep) of a single stateful vertex in the chosen algorithm. During each iteration, the vertex program can change its state, send messages to its neighbors or change its edges. Each iteration receives the previous iteration's messages as input. If a termination criterion is met, each vertex can vote to halt the execution and suspend itself. By suspending it basically yields its CPU time to the active nodes. Upon consensus about termination, the framework stops the execution.

By restricting indirect state changes to message exchanges, the authors argue that *the system is inherently free of deadlocks and data races* [17], while at the same time being simple and well-performing.

IV. Second generation Framework

This section introduces frameworks that, while loosely based on the Map/Reduce paradigm, significantly reduce the communication overhead imposed by the original proposal.

A. Parallelization Contracts (PACT)

Map/Reduce systems are limited to execute a batch job determined at compile time, as the data flow is fixed. In contrast, existing, well-known technologies have featured sophisticated query optimizers capable to dynamically adjust a data flow since a long time. These components transform the user's input into an optimized execution plan. The optimization may include rearranging the sequence of operators, based on their algebraic semantics.

The same is applicable to a Map/Reduce job [10]; however, a more formal definition of the programming model's operators is required. In the Stratosphere framework, so-called "parallelization contracts" define the semantics of an operator. The optimizer that consumes this information is the PACT compiler. A user implements the second order function of an operator, which is encapsulated inside a well-defined first order function. The encapsulation determines the input and output of the function. For example, a match-join requires two inputs and has one output. These encapsulations are referred as *input/output contracts*, respectively. Based on these contracts, the compiler is able to infer possible optimization strategies. Furthermore, the developer may give hints to the compiler via Java annotations. These hints include, for example, the expected size of the input / output, the ratio of input to output records or whether the input is sorted. Using that information, the PACT compiler will do an initial optimization pass before submitting the job to Nephele, the Stratosphere's run-time system.

1) Additional Operators: One criticism about Map/Reduce is the lack of an efficient JOIN operator [4]. While it is possible to implement that kind of operators on top of Map/Reduce, Stratosphere takes another approach, by natively providing such operation – in multiple forms. Stratosphere comes with MATCH, CROSS and COGROUP operators; others can be supplied by the programmer. These operators allow sophisticated implementations of graphs algorithms.

Both the left side and the right side of Figure 2 represent a k-core computation. The main difference is that on the right, we used a MATCH operator from the Stratosphere framework to join a vertices neighbors (input A) with its k-core value (input V). One might argue that JOIN is a complex operation; however, when joining sorted data, its cost is usually negligible because big data algorithms are generally IO-bound rather than CPU-bound. By providing the neighbor data as a separate input, in each iteration only the $\langle vid, core\rangle$ tuple is transferred from node O to node I, displacing the Schimmy pattern. In contrast, the Map/Reduce implementation requires a complete transfer the graph structure during each iteration. We compared both approaches in Section V-C2 to identify compelling differences.

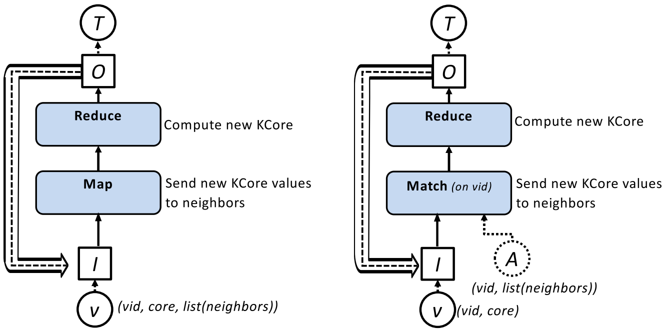

Figure 2. k-core decomposition Left: Map/Reduce, Right: Match/Reduce. *Notation: I: Input, O: Output, T: Final Output, A: Graph Structure, V: Vertex Data, Dashed Arrow: Iteration Body.*

2) Incremental Iterations: One particular feature of the Stratosphere framework is the built-in support for workset algorithms. This technique requires to split the algorithm data into a *solution set* S and a *working set* W with the goal of adding dynamic programming features to Map/Reduce. The working set contains only those nodes that need to be recomputed in an iteration, and is supposedly smaller than S, which contains all graph nodes. Graph algorithms that work only on their neighbor nodes, such as PageRank or Connected Components, exhibit such behavior. After an iteration S is updated and a new working set is computed. Therefore, fewer data is repeatedly sent across the network, reducing the computation time significantly. However, support for worksets has not yet surfaced in the current Stratosphere implementation.

An even more efficient solution is to consider every single computation as a "microstep" and update the solution set immediately. In that case a partial solution reflects all latest changes, which leads to finer-grained parallelism.

B. *Pregel*

There are two predominant distributed graph processing frameworks, that implement Pregel: Hama [23] and Giraph [24]. Both are built on top of Hadoop and are maintained by the Apache project. Hama provides two distinct collections of APIs: one pure BSP model for message passing and collective communication, and one vertex-centric model. Giraph focuses instead on the interoperability between other Apache projects, such as direct I/O from HBase and is backed up by companies, such as Yahoo and Facebook. The BSP APIs of both projects let the developer implement the functions that are executed by a single graph node. The node may send messages to other nodes that will be consumed during the next superstep. Furthermore, a node may vote to halt the execution.

Although both frameworks are implemented on top of Hadoop, they are fundamentally different from the matrix computation of Map/Reduce. At run-time the frameworks build up state for every single node and provide methods for sending messages to its neighbors. The programmer can concentrate on the actual algorithm she implements in contrast to implementing a map phase to perform the message sending, and setting up a reduce phase that needs to transfer the graph structure as input for the next map.

C. *GraphLab*

GraphLab evolved from its first iteration, a framework for CPU-rich shared-memory systems [15], to a distributed version, applicable to clusters of machines [14]. Just like the Pregel proposal, the system is graph-parallel; however, it employs different concepts related to dynamic programming techniques, synchronization constraints and vertex-to-node mapping.

While Pregel uses message passing as communication primitive, in GraphLab each vertex can access the state of adjacent vertices. GraphLab also lacks synchronized supersteps: vertex programs are executed asynchronously, while the sync task runs continuously in the background. One gain of direct access and asynchrony is that nodes always access the most recent state of their neighbors. Another gain of this asynchronous execution is that a single slow running vertex program can no longer stall the system. In Pregel, the interval between two supersteps is defined by the run time of the vertex with the largest neighborhood. As the number of edges is usually power-law distributed, there will be few but spacious vertices in the graph. This is why GraphLab goes a step further, by splitting vertices with enormous neighborhoods across different machines and synchronizing them.

The functionality of GraphLab is encapsulated in the *gather-apply-scatter* (GAS) pattern, which must be implemented by the programmer. In the first phase, executed in parallel on the edges of each node, information from neighbors (e.g., their k-core values) is *gathered.* Access to vertices is read-only, as required by the parallel execution of that phase. The second phase, called *apply*, is executed atomically, allowing changes to the data structures, for example the computation and update of a node's new k-core value. The final *scatter* phase, that is again executed in parallel on the node's edges, can be used for signaling neighbor nodes or updating edge data.

V. Comparison of distributed frameworks

Fine tuning both our implementations and the compared frameworks is critical for the execution time. We therefore start this section by discuss the details of both issues, before presenting the most important results from our study.

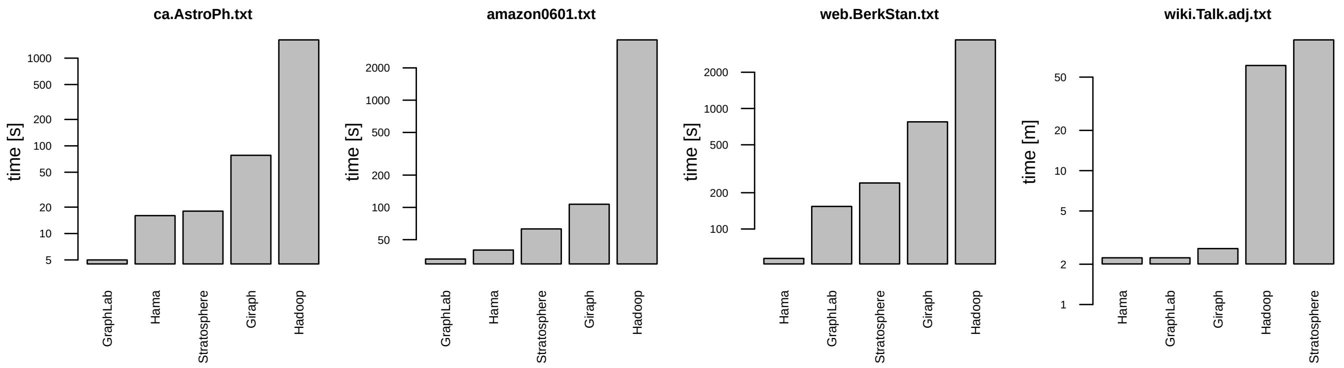

Figure 3. The minimum runtimes of all frameworks for different graphs.

A. Implementation

Hadoop: In order to define a baseline, we implemented a standard Map/Reduce k-core algorithm and applied the schimmy and grouping patterns. We employed the built-in support for iterations, which executes map and reduce in a loop until convergence is reached. The mapfunction reads unsorted tuples of vertex id, k-core value and a list of neighbors. Using the id as key, it sends the k-core value to both every neighbor and the vertex itself. Binned by each vertices key, the reducefunction receives a list of k-core values and computes a new score. It sends that value back into the mapfunction until the iterations stop. When applying the grouping optimization, the key for the reducefunction is computed by a hash function. Therefore, a single reducer instance computes the k-core of multiple vertices and outputs one message for each vertex. Using the schimmy pattern, only the vertex id and the k-core value is sent from the reducer to the mapper, the remaining operation is the same as previously described.

Stratosphere: We did a straight port of the Hadoop code to Stratosphere. However, as the PACT implementation of iterations has yet to surface, we implemented it at the Nephele Data Flow engine level. Furthermore we employed the *match* operator, as illustrated in Figure 2. This way, we can split the graph data and structure information into two inputs and cache the latter, which achieves a similar effect to the Schimmy pattern.

Recently Stratosphere gained Nephele support for workset iterations [8]. Its goal is to recompute only the nodes whose input values have changed. Based on that functionality we implemented two solutions. The first is to emit just enough nodes, to successfully recompute nodes, that have at least one recently changed neighbor. The second is a emulation of a Pregel implementation, that saves a node's state information in the solution set and passes messages along the working set (cf. Figure) 4. Because of the stateless nature of the underlying framework, all state needs to be passed among the Nephele execution nodes, hence both solutions represent a trade off. In the first case, this means increased complexity in the data flow and a slightly higher message exchange compared to Pregel. In the second case, the whole node state is passed around, leading to increased message size. While we implemented both solutions, the Pregel solution yield better results and is introduced in the following.

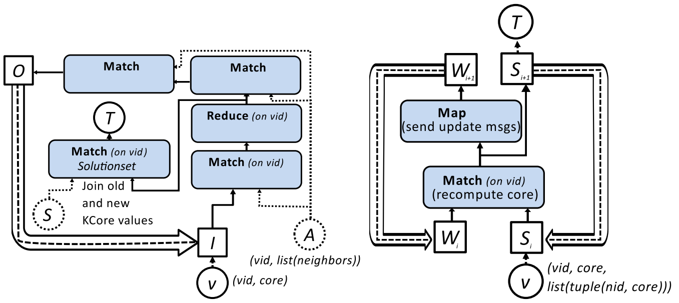

Figure 4. Dynamic programming inspired k-core decomposition in Stratosphere. Left: Map/Reduce using worksets. Right: Using a Pregel emulation

Giraph, Hama: Our Pregel implementations closely resemble the pseudo-code in Algorithm 1. Initially a vertex sets its k-core value to its degree on startup. When a vertex receives a message, it recomputes its value. The process repeats upon computation of a changed value, otherwise the framework suspends the execution of that node. As an optimization, we implemented a shared hash table, that serves the most recent computation results to processes on the same node. In this way, we emulate the "microstep" functionality of Stratosphere. Furthermore we worked with combiners to reduce the number of messages sent across the network and added filtering to our the code. The API differences between Hama and Giraph are marginal, hence both implementations are nearly identical.

GraphLab: Just like the Pregel implementation, we closely implemented our proposed pseudo-code using the GAS pattern: In the *gather* function, we collect the vertex id of each neighbor, along with its k-core value. To collect that data in a list, we reimplemented the `operator+=` function, as GraphLab assumes commutative associative operations of primitive data types as default. The new k-core value is computed in the *apply* function, while neighbors were notified in the *scatter* function. We were able to run this code both with the lowest consistency guarantees, thus the high parallelism.

Despite GraphLab's preference for the GAS pattern, we implemented also a message based version using GraphLab's

Dataset	Nodes	Edges
ca.AstroPh	18,772	396,160
ca.CondMat	23,133	186,936
Amazon0601	403,399	2,443,412
web-BerkStan	685,235	6,649,474
com.Youtube	1,134,890	2,987,624
wiki-Talk	2,394,390	4,659,569
com.Orkut	3,072,441	117,185,083

Table I
DATASETS USED IN THIS STUDY

built-in messaging framework. We also added the filtering technique to both versions of the code.

B. Testbed

To test our implementations, we set up an AWS cluster, running the 1.0.3 release of Apache Hadoop and HDFS together with the 0.6.0 release of Apache Hama and the recently released Giraph 1.0. We installed GraphLab 2.1.4414 and the latest Stratosphere checkin from the Stratosphere-iterations git branch, which is `fd4f` at the time of writing. Experiments were performed using up to 32 Amazon's M1 MEDIUM nodes with 2 EC2 Compute Units and 3.75 GB of RAM each. The cluster size was scaled from 2 nodes up to 32, each framework was configured to distribute computation evenly.

We carried out comparison measurements on a recent 16 CPU machine with 10 GB of RAM and Sata-SSDs. The software was similar to the EC2 cluster setup.

The data sets were taken from the public available collections of the SNAP [9] project. Table V-B summarizes them.

C. Results

For the sake of brevity, we introduce only the most interesting results. We cite the individual publications where appropriate. Unless explicitly noted, the cluster size was set to 12.

1) A five-way baseline: Overall we found the performance of Map/Reduce based approaches inferior to graph-centric ones (cf. Figure 3). Hadoop marks the bottom of the table in each of our comparisons. The results are from 2 times up to 66 times slower. We did not find significant improvements by using the group or schimmy patterns [12]. On the other hand, GraphLab was – by far – the fastest of all evaluated frameworks. Under all circumstances, the performance of at least one graph-centric framework superseded the other approaches. Therefore, from an execution time point of view, we clearly recommend relying on a graph-centric framework.

A closer analysis of Figure 3 reveals that data sets with less than a hundred thousand nodes play in the hands of Stratosphere. Its lack of a wiring phase, as in Hama, where nodes have to be connected to their neighbors, and the lack of Hadoop's serialization makes it the fastest of the Java-based frameworks for smaller data sets. However, for larger data sets such as the Amazon0601 or Youtube ones, the picture is different. Here GraphLab and Hama exhibit faster performance. In these cases, our Pregel Stratosphere code was able to yield comparable results. In contrast, the Map/Reduce and MATCH-based Stratosphere implementations, which are not shown here, were far off by a factor of at least 4. Therefore, in that scenario, while vertex centric frameworks take the lead, Stratosphere is at least comparable.

Applying the grouping or schimmy patterns to Hama did not yield any improvement (cf. [20]). The inferior performance of Stratosphere might be explained by the same insights: the density of the network might have hit a suboptimal code path.

2) To Map or to Match or to Pregel: The comparison of the more traditional approach of sending all data via a single input (map) versus having two inputs, one for the graph data and another for the computation data (MATCH), results in favor for the latter, for all tested graph data (cf. Table II). Going one step further and recomputing only the necessary nodes reveals even more gains.

Type	File	time [m]
ca.CondMat.txt	Match/Reduce	0.23
ca.CondMat.txt	Pregel	0.23
ca.CondMat.txt	Map/Reduce	0.25
com.youtube.ungraph.txt	Pregel	9.98
com.youtube.ungraph.txt	Match/Reduce	16.78
com.youtube.ungraph.txt	Map/Reduce	34.40
web.BerkStan.txt	Pregel	14.12
web.BerkStan.txt	Match/Reduce	24.70
web.BerkStan.txt	Map/Reduce	298.35

Table II
MAP VS. MATCH VS. PREGEL BASED KCORE IN STRATOSPHERE

For small datasets, matching does not yield a noticeable improvement. At the same time, the performance does not get worse due to the joining of both tables. For the Youtube dataset with ~ 1 million nodes and the Orkut data with ~ 3 million nodes, the benefit is between $1/3$ and $1/2$ of the computation time in a cluster of size 12. As previously mentioned, the Berkeley networks are associated with particularly bad performance in Stratosphere. One cause might be the join over nodes with a maximum degree of $84\,230$. However, the results still confirm our finding: Even in this scenario, matching is three times as fast. These results show that the addition of a match operator was indeed a valuable contribution, which both speeds up the computation and obsoletes the various proposals for join implementations for Map/Reduce.

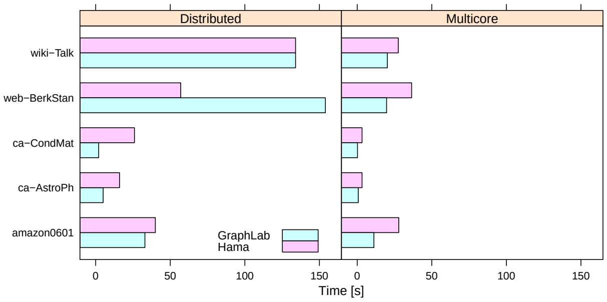

Figure 5. Hama and GraphLab in both a distributed and multicore setting.

3) Multicore and Shared Memory: To reconfirm our findings from the distributed scenario, we reproduced our measurements on a multicore machine. In Figure 5 we compare both scenarios using the analyzed vertex centric frameworks. The results for the Map/Reduce frameworks did not match the speed of their counterparts and have been left out. The first thing to notice is that GraphLab apparently comes from a multicore background and is still optimized for that. In that scenario, Hama performs worse than GraphLab by at least 20% for every tested data set. The gain shows up especially using large data sets in the multicore scenario, which is an important benefit.

In the distributed scenario, the gain is negligible for a huge data set like Wiki-talk, and even reversed for the Berkeley data. As previously mentioned, dense data set seems to exploit a shortcoming in the implementation of distributed GraphLab. For a complete analysis of GraphLab on a multicore machine see [18].

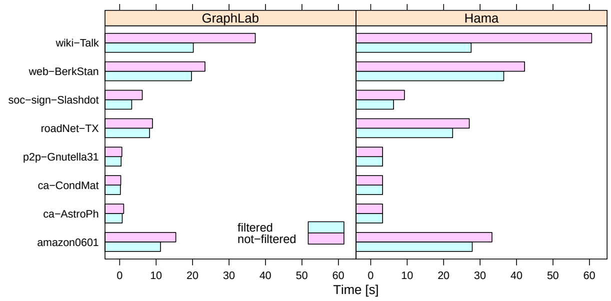

Figure 6. Evaluation of the filtering proposal using GraphLab and Hama.

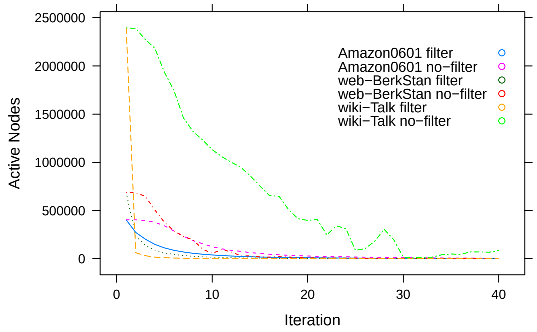

Figure 7. Active Nodes with and without filtering.

4) Filtering: Another contribution of this study is an improved algorithm for the k-core decomposition. It minimizes the message transfer among nodes, and thus the number of active nodes per iteration. This dynamic programming technique should have a measurable impact. Figure 6 analyzes the proposal on the multicore machine using a variety of data sets and both vertex centric frameworks. The impact is apparently stronger for GraphLab; however, it is measurable also for Hama. This might be related to implementation-specific differences in the frameworks.

For the huge Wiki-talk dataset, the filter cuts the execution time nearly by half. The plot showing the active nodes in Figure 7 explains such difference. Instead of the high number of active nodes, which only slowly fade into a stable state after around 30 iterations, the activity rapidly diminishes in less than 5 iterations. This effect, even if not as strong, is the same for all data sets.

5) Fault Resilience: A frequently cited reason for Hadoop's inferior performance is the built-in checkpointing after each step, which eases recovery, but hampers performance [5]. However, with the relative youth of its successors, it is currently the only framework that can actually recover from faults. Both distributed GraphLab and Stratosphere detect lost nodes quickly, however cannot recover from errors. Both display an error code on the console. Only Stratosphere has a built-in recovery mechanism, which is at the time of writing unable to handle iterations properly. Hence that situation might change soon. Hama's BSPMaster, on the other hand, waits for all processes to finish but does not appear to have timely heartbeat notification built-in.

These findings lead us to the recommendation that under the assumption of high probability of node failure and extreme long running jobs, a conservative choice of frameworks is suitable.

VI. Conclusion

In this paper we presented a thorough study of modern big data frameworks and their application to graph algorithms. We selected the k-core algorithm and adapted it to each platform. We executed the algorithm both in a distributed cloud settings as well as on CPU rich multicore machines. Our findings confirm the validity of these improved frameworks. Each of them improves over Hadoop in terms of execution time. However, from a programming paradigm point of view, a vertex centric framework is the better fit for graph related problems. Not only is the ease of programming an important point, as the focus can be close to the sketched algorithm. The performance of vertex-centric frameworks is not matched by Map/Reduce-based frameworks, even by improved ones, such as Stratosphere. Therefore, we clearly recommend one of the Pregel-inspired frameworks for graph processing, although the resilience of those frameworks has room for improvements.

Compared to Hadoop, Stratosphere enhanced the situation enormously, thanks to their flexible programming contracts concept. For graph algorithms, the MATCH operator as a mean to split input into worksets and graph input yields clear performance gains. An important announced, but still missing feature of Stratosphere will be the PACT compiler support for worksets. With that feature available, Stratosphere might be able to catch up to vertex centric frameworks, using highly optimized dataflows; however, the ease of programming might still be an issue. We therefore recommend a re-evaluation at that point.

Another outcome of this work is an improved dynamic programming algorithm, which greatly improves the runtime of k-core. Future work on graph algorithms might consider the filtering concept as a possible source for improvements.

Acknowledgment

This work has been partially supported by the EIT Activity “Europa” n. 12115 and by an *AWS in Education Research Grant Award.*

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://ec2.amazon.com/.

[2] J. Alvarez-Hamelin, L. Dall Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. *Advances in neural information processing systems*, 18:41, 2006.

[3] G. D. Bader and C. W. Hogue. Analyzing yeast protein-protein interaction data obtained from different sources. *Nature biotechnology*, 20(10):991–997, Oct. 2002.

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[5] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, July–Aug. 2009.

[6] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150, San Francisco, CA, December 2004.

[8] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, July 2012.

[9] http://snap.stanford.edu/index.html. Stanford network analysis project.

[10] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, July 2012.

[11] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[12] I. Leonardi. k–core decomposition of large graphs with hadoop. Master's thesis, Università degli Studi di Trento, March 2011.

[13] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[14] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. 06 2010.

[16] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[17] 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*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[18] L. Masera. Progettazione e valutazione di algoritmi su grafi in graphlab. Master's thesis, Università degli Studi di Trento, January 2013.

[19] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 207–208, New York, NY, USA, 2011. ACM.

[20] L. E. P. Morales. Large scale distributed graph processing: A comparative analysis. Master's thesis, Università degli Studi di Trento, March 2012.

[21] M. Schwarzkopf, D. G. Murray, and S. Hand. The seven deadly sins of cloud computing research. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 1–1, Berkeley, CA, USA, 2012. USENIX Association.

[22] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269 – 287, 1983.

[23] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society.

[24] The Apache Software Foundation. Apache Giraph. http://giraph.apache.org/.

[25] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/.

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

[27] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.