ReGraph: A Graph Processing Framework that Alternately Shrinks and Repartitions the Graph

Xue Li*[†]
Tsinghua University

Mingxing Zhang*[†]
Tsinghua University, Sangfor Technologies Inc.

Kang Chen[†][‡]
Tsinghua University

Yongwei Wu[†][‡]
Tsinghua University

ABSTRACT

"Think Like a Sub-Graph (TLASG)" is a philosophy proposed for guiding the design of graph-oriented programming models. As TLASG-based models allow information to flow freely inside a partition, they usually require much fewer iterations to converge when compared with "Think Like a Vertex (TLAV)"-based models.

In this paper, we further explore the idea of TLASG by enabling users to *1)* proactively repartition the graph; and *2)* efficiently scale down the problem's size. With these methods, our novel TLASG-based distributed graph processing system ReGraph requires even fewer iterations (typically ≤ 6) to converge, and hence achieves better performance (up to 45.4X) and scalability than existing TLAV and TLASG-based frameworks. Moreover, we show that these optimizations can be enabled without a large change in the programming model. We also implement our novel algorithm on top of Spark directly and compare it with other Spark-based implementation, which shows that our speedup is not bounded to our own platform.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Computing methodologies → Distributed algorithms**;

KEYWORDS

Graph Processing, Distributed System, Repartition

ACM Reference Format:
Xue Li, Mingxing Zhang, Kang Chen, and Yongwei Wu. 2018. ReGraph: A Graph Processing Framework that Alternately Shrinks and Repartitions the Graph. In *Proceedings of 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3205289.3205292

*X. Li and M. Zhang equally contributed to this work.
[†]Department of Computer Science and Technology, Graduate School at Shenzhen, Tsinghua National Laboratory for Information Science and Technology(TNLIST), Tsinghua University, Beijing 100084, China; Research Institute of Tsinghua University in Shenzhen, Guangdong 518057, China. M. Zhang is also with Sangfor Technologies Inc.
[‡]Corresponding author: Kang Chen (chenkang@tsinghua.edu.cn) and Yongwei Wu (wuyw@tsinghua.edu.cn).

1 INTRODUCTION

1.1 From Vertex to Graph

"Think Like a Vertex (TLAV)" [23] is the cornerstone of early large-scale graph processing systems [1, 14, 20, 22], and it's inherited by many works [6, 15, 25, 39, 40]. This philosophy leads to vertex-centric programming models that require users to implement programs from the perspective of a vertex rather than the whole graph. Typically, the execution of a user-defined vertex function involves: *1)* receiving messages from other vertices via incoming edges; *2)* updating the state of itself; and *3)* sending messages to other vertices via outgoing edges. In each iteration, this program kernel is executed on every vertex once if it is active, and a certain terminating condition is defined to stop the execution.

In contrary to the randomly accessible, "global" perspective of data employed by conventional shared-memory sequential graph algorithms, vertex-centric frameworks employ a local, vertex-oriented perspective of computation, encouraging practitioners to "think like a vertex". As a result, two different vertices/edges can be assigned to different workers (i.e., automatic scale-out) and can even be processed simultaneously if the intersection of their neighborhoods is empty (i.e., automatic parallelization). However, although TLAV leads to convenient programming models and has been proved to be useful for many algorithms, it does not always perform efficiently, because it is very short-sighted. Only the data of immediate neighbors can be read/updated in an execution so that the information is propagated just one hop at a time. Especially for large-diameter graphs, a TLAV-based framework may face the problem of slow convergence. Even with some optimizations such as the asynchronous execution or an advanced scheduler [24], a TLAV-based program is still short-sighted, which prohibits programmers to implement more efficient algorithms (such as disjoint-set based WCC).

To overcome this limitation, some works [29, 31, 38] propose an alternative philosophy named "Think Like a Sub-Graph (TLASG)". The main idea of TLASG is making the use of the fact that, for every node in the distributed graph processing cluster, a partition of the whole graph rather than only the neighborhood of a specific vertex/edge is available at a time. Thus, it is possible to reduce the number of iterations needed to propagate a piece of information from a source to a destination. For example, only one step is needed if these two vertices appear in the same graph partition.

As a summary, TLASG-based frameworks pass an entire sub-graph, rather than only a(n) vertex/edge, to the user-defined function, so that the program kernel can immediately exchange the data of two vertices/edges if they belong to the same sub-graph. This interface opens up the partition structure to users and allows information to flow freely inside a partition. As a result, it can usually converge much faster than a TLAV-based algorithm. At the same time, this model still provides a sufficiently high level of abstraction and is much easier to use than, for example, MPI. According to their evaluation, a TLASG-based framework can be up to 3.1X faster than a TLAV-based counterpart when running the connected component detection algorithm on hash-partitioned graphs.

1.2 Proactively Repartition

The main benefit of choosing TLASG over TLAV is that the information within a whole partition can be synthesized in a single iteration. But, if a static partition is used, the system may not be able to unleash its possibility fully. For example, existing graph partitioning algorithms are designed to reduce the network traffic (i.e., reduce the number of cross-partition edges or vertex replicas). But, for a TLASG-based algorithm, the quality of a partition is also affected by another metric, i.e., the number of sub-graphs it produces. Here, by using the number of sub-graphs, we refer the number of connected components after removing all the cross-partition edges. It is possible that the initial partition produces a large number of small sub-graphs, which significantly offsets the benefit of using TLASG-based algorithms. As one can imagine, even with TLASG, the data propagation is still one hop at a time but on the super graph[1] rather than the original graph.

Thus, it is natural to think that we may speed up the processing by proactively repartitioning the graph. Through continuously changing the partition of a vertex, it can meet more vertices and hence may lead to an even better convergence speed. But, the repartition procedure itself is expensive as it typically requires $O(|E|)$ communication cost. In common cases, this prohibitive cost of repartition overrides all the benefits it may have.

However, according to our investigation, it is possible to significantly reduce the repartition cost in a TLASG-based algorithm by reducing the graph size. The work [34] has proposed an innovative approach to reduce the scale of the problem through temporarily not loading the inactive vertices. Although it is based on an out-of-core environment that is different from ours, it does inspire us to dynamically capture the working set by eliminating parts of the graph that don't contribute. In our work, this can be achieved by deleting all the unnecessary edges after processing a sub-graph, as these edges' information is squeezed out and hence it is meaningless to retain them. The benefit of doing this is two-fold: *1).* It reduces the network cost, and this is actually a prerequisite of our repartition-centric algorithms since only by this way the overhead of repartition will not offset its benefits; *2).* It also reduces the computation cost of later supersteps, which can not be achieved by simply using a good static partitioning algorithm.

Moreover, the above alternately-shrink-repartition technique enables us to effectively **scale down** the problem. It is a well-known phenomenon that the execution time of a graph application typically will first decrease with the increasing number of workers. But, after passing a certain threshold, the execution time will increase if more workers are used, because the communication between workers is much more expensive than that within a worker. As a result, some works [18] try to scale down the size of the cluster for obtaining better performance. It is natural to integrate this mechanism in our framework, as we alternately decrease the number of edges (so that the needed number of workers decreases) and repartition the graph (so that the scale down procedure doesn't increase additional cost). In fact, our algorithm can scale down the problem size very fast. In most cases of our experiments, the size of the problem is reduced to under the capability of a single machine after running a few rounds (≤ 6).

[1] Super graph constitutes of super vertices. Each connected component is considered as a super vertex if it is still connected after removing all the cross-partition edges.

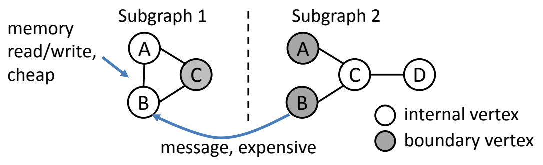

Figure 1: TLASG model in Giraph++.

1.3 Our Contribution

Based on the above observation, we propose a repartition-centric distributed graph processing framework named ReGraph. Different from existing TLASG-based frameworks, users of ReGraph can proactively insert repartition procedures between iterations for scaling down the problem or re-balancing the workload. We conduct our experiments on an 8-node cluster, and as data in Section 5 shows, in many real-world cases, applications that follow this novel workflow can converge much faster than the traditional algorithms, and hence lead to a significant speedup and better scalability. Moreover, besides the reduction of total execution time, our capability of scaling down the problem also leads to another merit, i.e., we reduce the amount of computing resources used (calculated by summing up the execution time of each worker). This characteristic is important in a cloud/multi-tenant environment, in which the unused workers can be assigned to other tasks.

However, our intention is to show that this novel algorithmic design technique is both general and effective. It is proposed as an enhancement rather than a replacement of the traditional models. Thus, in order to assure that our technique can co-exist with existing ones, we made the interface of our C++-based framework almost the same as GraphX's [15]. Therefore, it's easy to implement our algorithms using Spark programs. In fact, we evaluate two sets of comparisons: *1)* one is the comparison between existing graph processing frameworks and ReGraph; and *2)* the other is between GraphX and Spark programs that follow our novel workflow. Both experiments have shown considerable speedup (up to 45.4X).

2 BACKGROUND

To the best of our knowledge, the first publication that proposes TLASG is Giraph++ [31]. Researchers from IBM found that the use of TLAV-based program forces costly messaging even between vertices in the same partition. They argue that each graph partition essentially represents a proper sub-graph of the original input graph, instead of a collection of unrelated vertices. Thus the framework should enable users to process a local sub-graph at a time.

As an illustration, Figure 1 presents a simple graph with only four vertices and a possible 1D partition. In this example, vertices

A and B (internal vertices) are assigned to worker 1, and the other two vertices are assigned to worker 2. As a result, although only vertices A and B are assigned to worker 1, it also maintains a copy of vertex C as a boundary vertex. During the processing procedure of Giraph++, users are enabled to freely exchange the information of two internal vertices if they are assigned to the same worker. This kind of exchanges can even happen multiple times in one superstep. As a comparison, updates to boundary vertices are exchanged via message passing, which only happens once at the end of each superstep. According to their evaluation, the graph-centric model leads to significant reduction of network messages and execution time per iteration, as well as fewer iterations needed for convergence.

GoFFish [29] and Blogel [38] also follow the TLASG philosophy. They further improve the programmability and performance by providing more flexibility to users and block-level communication. For example, GoFFish allows an arbitrary shared-memory graph algorithm to be used as a black box over each connected sub-graph.

3 MOTIVATING EXAMPLE

As mentioned in Section 1, the main contribution of this paper is a general algorithmic design technique that alternately shrinks and repartitions the graph. In this section, we demonstrate this workflow's capability by presenting a thorough example on calculating Weakly Connected Components (WCC) of a graph. Specifically, a weakly connected component is a maximal subgraph of a directed graph such that for every pair of vertices in the subgraph, there is an undirected path connecting them (i.e., the two vertices are connected after replacing all the directed edges with undirected edges). This application lies at the core of many data mining algorithms and is a fundamental subroutine in graph clustering. We first describe the traditional TLAV and TLASG based algorithms and point out their limitations. Then, we introduce our new repartition-centric algorithm and discuss the possible enhancements (e.g., the best repartitioning algorithm). Finally, we use several micro benchmarks to show the effectiveness of our method.

3.1 TLAV and TLASG Based Algorithms

In TLAV-based frameworks, the standard way of calculating WCC is label propagation. In this algorithm, each vertex maintains a property that represents its component label, which is its own vertex ID initially. In subsequent supersteps, a vertex will change its label if it receives a smaller ID and then it will propagate this smaller ID to all its neighbors. This method is both simple and scalable, but not necessarily efficient. As the label is propagated to direct neighbors at a time, it may require many rounds of messages to converge, especially for graphs that have large diameters.

To overcome this problem, TLASG-based algorithm exposes the whole graph partition of each worker to users. Therefore, at the beginning of each superstep, an arbitrary **shared-memory algorithm** can be used on every partition to immediately propagate the smallest vertex ID[2] of every local WCC to all the connected vertices contained in the same partition. Then, only the boundary vertex needs to send its locally computed component label to its corresponding internal vertex. Both algorithms finish when there is no further update. It is obvious that the TLASG-based algorithm

[2]Smallest ID among both the internal vertices and messages sent from other partitions.

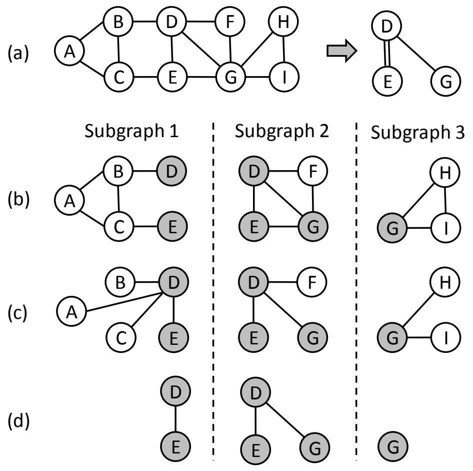

Figure 2: An example of graph shrinking.

can be much faster than the TLAV-based counterpart. But, as mentioned in Section 1, the effect of this optimization heavily depends on the quality of the partition. The possibility of TLASG may not be fully unleashed if the super graph produced by the initial partition still has a large diameter.

3.2 Repartition-centric Algorithm

3.2.1 Overview. To overcome the shortages of TLAV-based and TLASG-based algorithms, we propose a repartition-centric algorithm for WCC. As an overview, the whole procedure of this algorithm contains two phases:

1). The algorithm starts from a **scale-down phase** that efficiently scales down the problem's size (i.e., number of edges remained in the graph). As an illustration, Figure 2 (a) presents an example of the initial and ending state of a single process iteration (i.e., a superstep). As we can see, the number of edges is largely reduced from 13 to only 3. Typically, we will reduce the number of used workers while the size of the problem is decreased until all the remained edges can be contained in a single worker.

2). According to our evaluation, only a few supersteps are needed to shrink the original problem into a smaller one that can be handled by a single worker, which is the end of the scale-down phase. However, the results calculated on only this smaller problem may not reflect the whole results. Consequently, our algorithm also has a **back-propagation phase** that propagates the results back in just the reverse order of the former scale-down phase.

3.2.2 Scale-down Phase. Specifically, in each superstep of the scale-down phase, the edges of the graph are first equally partitioned into different workers (i.e., we use the 2D partitioning algorithm proposed by PowerGraph rather than the 1D methods used by Pregel and Giraph++). We ask users to set a threshold T that designates the minimum number of edges that should be contained in a worker. It depends on the memory size of each machine, and an alternative approach is automatically setting it to be the initial number of edges. Thus, when the number of edges decreases, the number of used workers is correspondingly reduced. Figure 2 (b) illustrates a possible result of the partition. In this figure, vertices represented by filled circles are **shared** vertices. Similar to boundary vertices defined in Giraph++, a vertex is shared if its edges are not all assigned to the same worker. As a result, there are two or more workers containing a replica of this vertex. In contrast, the other vertices represented by white circles are local vertices. We do not need to store a full copy of the graph in each node, thus the data of vertices are shuffled during repartitioning. But, since we scale down the problem size by removing unneeded vertices/edges, the data-copy cost of this shuffling is typically much less than an ordinary communication round of other frameworks.

After partitioning, an arbitrary shared-memory algorithm can be used to calculate the local connectivity relationship of each partition, such as DFS, BFS, and the disjoint-set union algorithm (which we use) [13]. With this local connectivity information, we can now **reconstruct** the whole graph partition into a new subgraph that *1)* preserves the **same** connectivity characteristic of the original graph partition; but *2)* has the **minimum** number of edges. Theoretically, the optimal way of doing such reconstructing is transforming the original partition into a forest. As Figure 2 (c) shows, each graph partition is rebuilt into a forest[3] by *1)* deleting all the original edges; and *2)* adding one edge for each vertex that points to the **center** (whose ID is the component label) of its corresponding local WCC. In other words, after the transformation, every local WCC is represented by a star-like tree, and all the vertices of a local WCC are still connected. We will explain why the star shape is most suitable for our use case in Section 3.3. Essentially, this procedure remarkably reduces the graph's size. A subgraph with $|E'|$ edges and $|V'|$ vertices will only hold "$|V'|-$ (number of local WCCs)" edges after the transformation.

Although the above procedure can largely decrease the number of edges, its theoretical limitation is reducing the number of edges to "$|V|-$ (number of WCCs)" edges, where $|V|$ is the number of vertices in the original graph. In fact, we can further reduce the graph's size by removing all the local vertices (as shown in Figure 2 (d)). Because all the edges of a local vertex are in the same worker, we can assure that these edges will not incur any further merging of two local WCCs in later supersteps. In other words, the deleting of local vertices will not affect the connectivity characteristics of the original graph. Therefore, we can delete a local vertex, only recording that its final WCC should be the same as its local WCC's center's. Note that since we delete all local vertices when reconstructing subgraphs, we can only choose a shared vertex (if exists) as the center of a local WCC, instead of choosing the vertex with the smallest ID (as TLAV or TLASG-based algorithms).

After the above procedure, the original graph (the left one in Figure 2 (a)) is transformed to a new one (the right one in Figure 2 (a)) that has much fewer edges. Then, we can scale down the cluster's size, repartition the new graph (move all edges left to the same node as only a single node remains in this example), and start computation for the next iteration. This iterative procedure finishes when only one node left.

3.2.3 Back-propagation Phase. After scaling down to only one worker, we can easily decide the final component of every **remained** vertex. However, such result is only a subset of the whole result, because many vertices are deleted for being local vertices in

[3] The number of trees contained in the result forest should be equal to the number of local WCCs in the original graph partition. As a result, only a single tree is contained in each forest produced in our example.

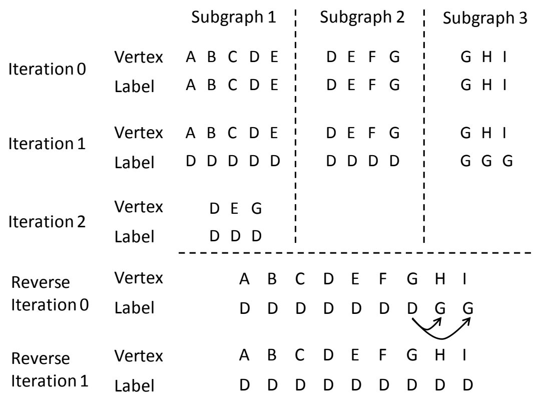

Figure 3: Execution procedure of a WCC example.

a certain superstep. As a result, we still need a back-propagation phase that constructs the final results by propagating information in just the reverse order of the former scale-down phase.

As illustrated in Figure 3, each vertex maintains a label demonstrating the component it belongs to, which is its own ID at the beginning (iteration 0). And at the end (iteration 2), we can easily find that all the remained vertices (D, E, and G) belong to the same WCC so that they are assigned with the same label (D). The problem here is that, in iteration 1, vertices A, B, and C are deleted because they are local vertices of graph partition 1, so that these vertices' final labels cannot be inferred in iteration 2. Similarly, vertex F, H, and I are also deleted because they are local vertices of graph partition 2, 3, 3, respectively. Therefore, before deleting a local vertex, we record the information about which vertex is its corresponding local WCC's center, and we can assure that the final label of a local vertex must be the same as this center's. As an example, the final label of A, F, I must be the same as the final label of D, D, G, respectively. With such information, we can simply construct the final results by back propagating the final label information in a backward direction (Figure 3). Since this back-propagation phase is unique in our algorithm, one may think that it is complex and hence increases the programming overhead of users. However, as we will show in Section 4.4 with pseudocode, these steps can be implemented very straightforwardly by using the well-known "join" operation, which is already very familiar to users.

3.3 Partitioning Algorithm

Although the quality of the initial partition does not have a decisive impact on the performance of our algorithm since we continuously repartition the graph, the partitioning algorithm still plays a vital role. In traditional graph frameworks, there are only two main metrics that are used to evaluate a partitioning algorithm: *1)* the skewness; and *2)* the amount of communication cost that it produces. But, in our algorithm, we need to choose an algorithm that is *1)* lightweight and *2)* produces a partition that we can delete as many edges as possible. The first requirement is because we need to use this partitioning algorithm for multiple times rather than only producing an initial partition, and hence only the heuristic-based simple partitioning algorithms are practical. As for the second requirement, the more edges that we can delete, the less cost are the following repartition and computation steps.

Table 1: Execution time and iterations needed of WCC.

	TLAV		TLASG		Repartition	
Dataset	Time	Iter	Time	Iter	Time	Iter
Pokec	6.249s	10	35.06s	9	2.110s	2
LiveJ	10.18s	12	74.31s	10	4.279s	3
Dimacs	2565s	6262	926.0s	18	21.93s	4

Actually, after each iteration, one edge remains for every shared vertex if it is not chosen to be center of its local WCC. So, the number of result edges after a shrinking step is "(number of shared vertices) - (number of local WCCs)". Since the second variable of this formula is hard to control, our goal is to minimize the first variable, i.e., reduce the number of shared vertices as much as possible.

Coincidentally, an existing partitioning algorithm hybrid-cut [6] is reported to be able to satisfy our requirement, in PowerLyra system that proposes hybrid-cut, the communication cost is also proportional to the number of shared vertices. As a result, although hybrid-cut is only designed for reducing communication cost, it is also able to maximize the number of edges that will be deleted for us. However, the effectiveness of hybrid-cut is achieved by treating high-degree and low-degree vertices differently, which is useful only when the degree distribution of the graph is very skewed. This is not a problem for its original use case because most real-world graphs follow the power-law distribution. But, in order to use hybrid-cut in our algorithm, we must assure that the same property remains for all the graphs that are reconstructed by every shrinking step. So, we reconstruct the graph into a forest of **star-like** trees, which has the most skewed degree distribution.

3.4 Micro Benchmark

In this section, we use some micro benchmarks to validate that our algorithm can significantly reduce the computation cost and hence lead to a better performance. More comprehensive evaluations are presented in Section 5. We evaluate all of the three WCC algorithms we have introduced on three different real-world datasets. Specifically, we implement our repartition-centric algorithm in C++ and MPICH2 and fix the threshold T to 5 million edges. We also test PowerLyra as a representative of TLAV-based frameworks, since PowerLyra is reported to be faster than other TLAV-based frameworks such as PowerGraph, GraphLab, and Giraph. As for TLASG-based frameworks, we choose Giraph++. We don't test Blogel because although it provides the most efficient TLASG-based WCC algorithm, this implementation depends on an expensive preprocessing procedure to partition the graph (e.g., more than 20s for the Pokec graph), which spends much more time than the entire execution of our algorithm.

Table 1 shows the performance evaluated by running these three algorithms on 8 workers. As we can see, the TLASG-based algorithm typically requires fewer iterations than the TLAV-based algorithm, and our repartition-centric algorithm requires even fewer. Take Dimacs, a real-world road graph that has an extremely large diameter, as an example, the TLASG-based algorithm remarkably reduces the number of iterations from 6262 to 18, and our algorithm can further reduce this number to 4. As for performance, the iteration reduction achieved by Giraph++ only leads to a better performance on Dimacs mainly because Giraph++ is implemented in JAVA. In contrast, our repartition-centric algorithm always achieves the best performance, which is about 2.4-42.2 times faster than the best of TLAV and TLASG. In addition, we also re-implemented the TLASG-based algorithm in C++ and MPICH2 for a fair comparison, and our algorithm still shows a considerable speedup (1.5X-9.3X).

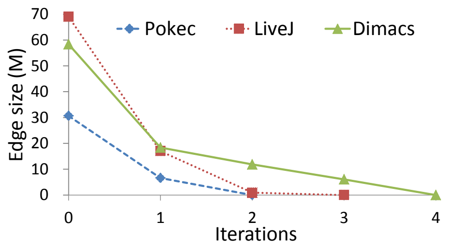

Figure 4: Number of remained edges.

To further demonstrate the scaling-down capability of our algorithm, we count the number of remained edges for each iteration. As shown in Figure 4, the number of edges is decreased by 68.5%-78.5% in the first iteration. Moreover, only a few more iterations are needed to reduce the number of edges to less than the T we set.

4 REGRAPH

In order to facilitate users to design and implement repartition-centric algorithms, we built a new graph processing framework named ReGraph. In this section, we first describe the typical workflow of ReGraph. Then, we introduce the data and programming model of it, which are very similar to GraphX's [15]. Finally, we use an example to demonstrate the usages of ReGraph.

4.1 Overview

Figure 5 presents the standard workflow of a ReGraph program. For each iteration, the program will first repartition the graph. Then, it will process the sub-graphs concurrently and properly delete the unnecessary edges. Take WCC as an example, *1)* its processing phase is a local procedure that finds local WCCs on each graph partition; and *2)* the following shrinking phase refers to the graph transformation procedure (i.e., from sub-graph to forest).

Note that at the beginning of each iteration, the graph is first repartitioned to obtain a more balanced workload and shuffle the belonging of vertices. During this procedure, we require that each working node should receive at least T edges so that the size of the cluster is scaled down to no more than $current\_edge\_size/T$ nodes if necessary. Typically, the distributed iterative execution will terminate after all the remaining edges can be handled by a single worker. After that, more processing may be needed to complete the algorithm. As for WCC, a back-propagation procedure is executed.

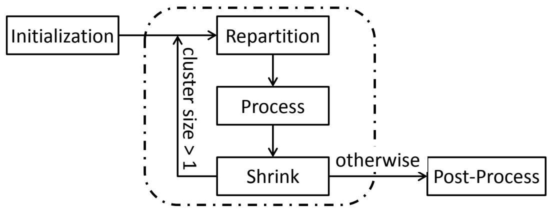

Figure 5: Main workflow of ReGraph.

Table 2: Data model and programming model of ReGraph.

Data
$KVC < K, V > \{N,\ hash\_table < K, V > data\}$
$SGC < E > \{N,\ vector < Edge < E >> edge\_set\}$
Core API of *KVC*
get(K) $->V$
set($K,\ V$) $->void$
sync(
$merger : (V,\ V) -> V$
) $->KVC$
join(
$KVC*$,
$joiner : (K,\ V,\ V) -> K,\ V$
) $->KVC$
Core API of *SGC*
mapGraph(
$vector < KVC* >$,
$updater : (vector < Edge >\&,\ vector < KVC* >) -> SGC$
) $->SGC$
partition(
int,
$vector < KVC* >$,
$partitioner : (Edge*,\ vector < KVC* >) -> int$
) $->SGC$

4.2 Data Model

As we have mentioned previously, in order to lower users' porting cost, we define the data and programming model of ReGraph in a way that much similar to GraphX. Specifically, there are only two kinds of data structures are provided, as illustrated in Table 2.

The first one is *KVC* (Key-Value Collection), which is a collection of key-value pairs that are distributed among the **used** workers. Each used worker will maintain a subset of the whole set of key-value pairs. *KVC* is very similar to the "PairRDD" defined by Spark. But, in ReGraph, *1)* key-value pairs assigned to each worker are maintained in a hash table rather than a simple vector, and it enables us to efficiently implement the *get* and *put* operations; *2)* a parameter N is designated to specify the number of used workers. When the size of the cluster is scaled down, one can reduce this parameter and then a repartitioning procedure is invoked to reduce the number of used workers.

The second high-level data structure is *SGC* (Sub-Graph Collection), which is a collection of edges that are also distributed among the N used workers. Similar to the "EdgeRDD" provided by GraphX, ReGraph uses 2D partitioning so that each worker is simply assigned with a disjoint set of edges. As we will show by examples presented in Section 4.4, *KVC* is usually used to store vertex data (use vertex ID as keys and the corresponding properties as values) while *SGC* is typically used to store edge data.

4.3 Programming Model

ReGraph provides only a limited set of operations for accessing the data defined in *KVC* and *SGC*. In this section, we will describe the main APIs one by one, and then, in the next section, we will demonstrate their usages with an example.

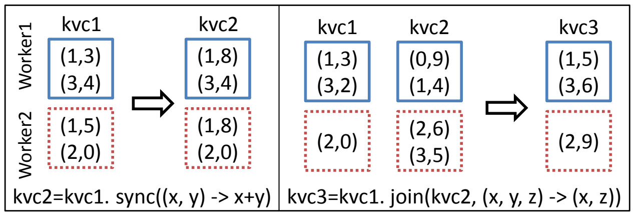

Figure 6: An illustration of KVC's operations.

4.3.1 Operating Key-Value Collection. As Table 2 shows, ReGraph allows users to directly access a single key-value pair contained in *KVC* via the *get*() and *set*() APIs, which are both **local** operations. Thus, a single key can actually appear multiple times in the same *KVC*, as long as different pairs are maintained by different workers. For example, although pairs *(key, v1)* and *(key, v2)* have the same key, they can simultaneously exist in the same *KVC x*, if *(key, v1)* is stored on worker *i*, *(key, v2)* is stored on worker *j*, and $i \neq j$. In this case, *x.get(key)* will return *v1* if it is executed on worker *i* and will return *v2* if it is executed on worker *j*.

In order to synthesize values on different workers, we also provide two other APIs, namely *sync* and *join*, as illustrated in Figure 6. Both APIs are global operations that will incur network communication. Specifically, *sync*() takes a user-defined *merger* to combine values of the same key on different workers for a single *KVC*. For example, if a specific *key* is stored on three different workers 1, 2, 3 and the corresponding pairs are $(key, v1)$, $(key, v2)$, and $(key, v3)$ respectively. After performing a *sync*() operation, each of these three workers will still contain a key-value pair whose key is still *key*; but the value of these three copies will be the same v' that equals to $merger(v1, merger(v2, v3))$. One should note that: *1)* for every worker, a key-value pair (key, v') will exist in the resulting dataset if and only if there is a pair (key, v) originally; and *2)* the *merger* function is required to satisfy the commutative and associative law, so that the order of performing *merger* function does not matter (e.g., $merger(v1, merger(v2, v3)) == merger(merge(v3, v1), v2)$).

The other operation *join*() takes two *KVC*s as input to compute a new one. It requires that the second input *KVC* has no data for the same key on different workers (or at least have the same value). Another requirement is that the first *KVC*'s datatype for value is the same with the second *KVC*'s datatype for the key. With these constraints, the *join* operation performs in the following manner: for each pair (a, b) in the first input *KVC*, it will find the corresponding pair (b, c) in the second input *KVC*, and then outputs a (key, v) into the generated *KVC* where (key, v) equals to $joiner(a, b, c)$. Note that this operation is kind of different from the one used in Spark for better convenience, though the function of our *join* could be implemented using the traditional *join* operation.

4.3.2 Operating Sub-Graph Collection. As ReGraph is a TLASG-based framework, its only core API that processes graph data is a *mapGraph*() operation that exposes the whole graph partition to users. This operation takes an old *SGC* and a set of *KVC*s as input to calculate a new *SGC*. Within its scope, users are allowed to define arbitrary shared-memory algorithms that can *1)* add or delete edges to/from the **local** graph partition by modifying the *edge_set*; and *2)* read and modify properties stored in the input *KVC*s via the *get* and *set* operations. The only requirement is that all these operations are performed locally.

Algorithm 1 Program for WCC.

```
Data
    $SGC < void > g$;
    $KVC < VID, EID > local\_degree, degree$;
    $KVC < VID, VID > center, result, unknown$;
    $stack < KVC < VID, VID >> s$;
    $vector < Edge < void >> new\_edge\_set$;
    $vector < KVC* > kvcs = \{\ldots\}$;//pointers to all above KVCs
Functions
    $count\_degree()$:
        foreach $(u, v)$ in $edge\_set$
            $local\_degree.set(u, local\_degree.get(u) + 1)$;
            $local\_degree.set(v, local\_degree.get(v) + 1)$;
        return $g(edge\_set)$;
    $local\_WCC()$:
        $sequentialWCC()$; //run a sequential WCC algorithm
        $unknown, new\_edge\_set = \{\}$;
        foreach $(v, c)$ in $center$
            if $!islocal(v)$
                $new\_edge\_set.push(Edge(v, c))$;
            else if $!islocal(c)$
                $unknown.set(v, c)$;
            else
                $result.set(v, c)$;
        return $new\_g(new\_edge\_set)$;
Computation for each iteration
        $g = g.mapGraph(\{\&local\_degree\}, count\_degree)$;
        $degree = local\_degree.sync((a, b) -> a + b)$;
        $g = g.partition(N, \{\&degree\}, hybrid\_cut)$;
        $g = g.mapGraph(\{\&local\_degree\}, count\_degree)$;
        $degree = local\_degree.sync((a, b) -> a + b)$;
        $g = g.mapGraph(kvcs, local\_WCC)$;
        $s.push(unknown)$;
Post-process
        while $!s.empty()$
            $unknown = s.pop()$;
            $unknown = unknown.join(result, (a, b, c) -> (a, c))$;
            $result = \{result, unknown\}$;
        return $result$;
```

In fact, there are only two kinds of communication that may happen in ReGraph. The first is caused by global *KVC* operations (i.e., *sync* and *join*), which are usually used to synchronize vertex data. This procedure is similar to the communication pattern of existing TLASG-based frameworks. For example, as we have described in Section 2, in the end of each superstep, Giraph++ needs to synchronize the vertex property. In ReGraph, it can be implemented by a single *sync* operation on the corresponding property *KVC*.

Moreover, as ReGraph tries to proactively repartition the graph, it also provides another API *partition*() that will repartition the graph's edge set. The first parameter of this function is an integer N, which indicates that the edges contained in this *SGC* should be repartitioned into N parts. The third parameter of this function is a user-defined function called *partitioner*. The execution of *partitioner* will consume an edge and output a hash code h, which indicates that this edge should be assigned to worker $h\%N$. As the decision of this assignment may require the access to some vertex properties, these properties can be passed to the *partition*() function by using its second parameter, which is a vector of *KVC*s.

4.4 Example

To illustrate the usages of ReGraph's API, we present the implementation of our WCC algorithm (Algorithm 1).

Specifically, at the beginning of each iteration, a local function is called to count the degree of each vertex in the local graph partition (via a *mapGraph* operation). Then, the global degree of each vertex can be calculated from local degree through using a *sync*() function. This degree information is then used for repartitioning the graph with hybrid-cut [6]. After repartitioning, local degree and global degree should be calculated once again for deciding whether a vertex contained in the partition is a local or shared vertex. With all the above information, the *local_WCC*() is executed concurrently on every worker to process and shrink the graph partition that it holds. Within this function, any shared-memory WCC algorithms could be utilized. To shrink the graph, we process each vertex v and the center c of the corresponding component: *1)* if v is shared, an edge (v, c) should be added to the new edge set; *2)* if v is local but c is shared, record in *unknown* that the final label of v is the same as the final label of c; *3)* otherwise, c is the final label for v, which can be added to the result directly.

After all iterations finish, a back-propagation procedure is required to get the final labels for those vertices added to *unknown*. In each reverse iteration, *unknown* is first joined with *result* to decide the final labels for vertices in it, which are then added to *result*.

Table 3: A collection of real-world graphs.

Dataset	Vertices	Edges	Description
Pokec [19]	1.63M	30.6M	Pokec social network
LiveJ [19]	4.85M	69.0M	LiveJournal social network
Dimacs [10]	23.9M	58.3M	Full USA road network
UK-2002 [3–5]	18.5M	298M	Web graph of the .uk domain
Twitter [17]	41.7M	1.47B	Twitter social network

5 EVALUATION

In this section, we present the evaluation results of ReGraph. Specifically, we choose four representative graph applications (WCC, MIS, MCST, and TC) and redesign the original algorithm into a repartition-centric manner. Each of them represents a general category of graph algorithms and is frequently used as a subroutine of more complex algorithms, which demonstrates that our general algorithmic design technique has a large range of applicability.

To further demonstrate the effectiveness of ReGraph, we implement the above algorithms and compare them with state-of-the-art systems. As these four applications are not always supported by all the existing systems, we always try every existing system and report the comparison between ReGraph and the best of others (the system that achieves the best average performance). Moreover, we also present an example of implementing repartition-centric algorithms on top of Spark, and compare it with GraphX, which is also a Spark-based graph processing system. Results show that our technique is not bounded to our system. Users can take advantage of it even though they do not want to change the platform.

5.1 Evaluation Setup

All our experiments are conducted on an 8-node Intel(R) Xeon(R) CPU E5-2640 based system. All nodes are connected with a 1Gb ethernet and each node has 8 cores running at 2.50 GHz. In order to perform the comparison, the latest version of PowerLyra, GPS, Giraph++ and GraphX are installed on the cluster. We use a collection of real-world graphs, the basic characteristics of each dataset are illustrated in Table 3. All of these graphs except Dimacs are social graphs that satisfy the power-law degree distribution.

Instead, Dimacs is the full USA road network which has a large diameter. For unweighted graphs, random weights ($\in (0, 1)$) are added to each of the edges if necessary.

5.2 Applications

In this section, we briefly introduce the four applications and their implementations in both ReGraph and other systems.

5.2.1 WCC. The design of our repartition-centric WCC algorithm is already presented in Section 3. As a comparison, we will also report the performance of PowerLyra's built-in TLAV-based WCC application, because although Giraph++ provides a TLASG-based WCC application that requires much fewer iterations than PowerLyra, our evaluation results show that its performance is typically still lower than PowerLyra (except on Dimacs).

5.2.2 MIS. Maximal Independent Set (MIS) is another important and widely-used graph application, whose output is an arbitrary maximal independent set of the input graph. In graph theory, a set of vertices constitutes an independent set if and only if any two of the vertices that contained in it do not have an edge connecting them. A maximal independent set S is then a set of vertices that *1)* constitutes an independent set; and *2)* there does not exist another independent set S' that is a proper superset of S (i.e., $S' \supset S$). Different from WCC, MIS is hard to be implemented in a message-passing model and hence is not provided by most existing graph processing systems.To the best of our knowledge, GPS is the only graph processing system that contains a distributed MIS implementation, which is based on Luby's classic parallel algorithm [21]. GPS is an open-source Pregel implementation from Stanford Infolab, which was reported to be 12X faster than Giraph [25].

However, MIS can be solved very easily in our framework. For each iteration, we just need process all local vertices one by one in each partition. If a local vertex has no neighbors in the MIS, we can safely add it to the final result and then all its neighbors (no matter local vertices or shared vertices) will never appear in the MIS. In other words, an edge will remain in an iteration if and only if both of its vertices are shared vertices and have no neighbors in the MIS.

5.2.3 MCST. Minimum Cost Spanning Tree (MCST) is an application that calculates a spanning tree of a connected, undirected, weighted graph. This tree should connect all the vertices of the graph with the minimum total weight of its edges. For unconnected graphs, we calculate an MCST for every connected component, i.e., a minimum spanning forest. MCST is an important graph application that is used directly in the design of networks and invoked as a subroutine in many other algorithms, such as [7, 9, 30]. As a result, sequential MCST algorithms have been well studied, such as Kruskal's algorithm [16], which we use. However, distributed MCST is usually very complex. As far as we know, GPS is the only graph processing system that provides a distributed MCST implementation, thus we use GPS as a baseline.

In contrast, rather than using the complex algorithm (e.g., the parallel Boruvka algorithm [8]), MCST can be solved extremely simply in ReGraph. It can be easily proved that, if an edge is not used in the MCST of a sub-graph, it is also not needed in the generation of the whole graph's MCST. With this property, we can efficiently shrink the size of the problem by deleting unneeded edges. This

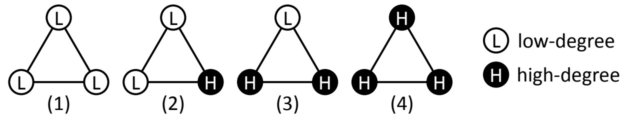

Figure 7: The four kinds of triangles.

technique enables our algorithm to work on extremely large graphs. After the shrinking procedure, we will scale down the cluster size if necessary, repartition the graph and continue computation, just the same as WCC's or MIS's workflow. When there is only one worker remained, all edges in the local MCST will be added to the result.

5.2.4 TC. Triangle Counting (TC) is a basic problem that is used as a subroutine of many important social network analysis algorithms [2, 11, 35, 36]. This application counts the number of triangles in an undirected graph, where a triangle is formed by three vertices and edges between each pair of them. We use an implementation of TC in PowerLyra to compare with our work. Actually, there are two versions of implementations of TC in PowerLyra. We choose the optimized one that implements the "hash-table" version of "edge-iterator" algorithm described in [27] to be the baseline.

In our system, we propose a novel repartition-centric TC algorithm. Inspired by hybrid-cut, we divide the vertices into two categories (high-degree and low-degree) and process them differently. Then, all the possible triangles can be divided into four categories (Figure 7) and only the first three types (have at least one low-degree vertex) will be counted in each iteration. Specifically, in an iteration, an edge is processed if and only if: *A)* it is between two low-degree vertices; or *B)* it connects a low-degree vertex and a high-degree vertex. For a type A edge, we allocate it according to the vertex with the smaller ID. While when allocating a type B edge, the low-degree vertex's ID is considered. As a result, each low-degree vertex is stored in exactly one partition with two sets containing these two kinds of edges respectively: *1)* one set contains all its high-degree neighbors (type B edges); and *2)* the other contains all its low-degree neighbors whose ID are greater than itself (type A edges). As for the edges between high-degree vertices, a hash-based random partition method is used.

In each iteration, we count the number of triangles containing **at least one** low-degree vertex (the first three types of triangles in Figure 7). The first two kinds of triangles could be counted by counting the intersections of two low-degree vertices' neighbor sets (sets containing low-degree neighbors for the first kind of triangle, and sets containing high-degree neighbors for the second kind). While for the third kind, we can enumerate two vertices of a low-degree vertex's high-degree neighbors and check if the edge between them exists. In the end of an iteration, the graph is shrunk by removing all the low-degree vertices. After the shrinking, many of the original high-degree vertices become low-degree vertices and then the above procedure is executed once again. The algorithm ends only when all the remained edges can be held in one machine.

Essentially, the advantage of our algorithm can be explained by the famous inequality of

$$(a_0 + a_1 + ... + a_n)^2 \geq a_0^2 + a_1^2 + ...a_n^2$$

Through processing only low-degree vertices in each iteration, we gradually transform high-degree vertices into low-degree vertices. The aggregating complexity is similar to the transformation from

Table 4: Execution time (in second/iteration). Since GPS's MIS implementation is a stochastic algorithm, we run it several times for each case and present the average execution time as well as the number of iterations.

		Pokec		LiveJ		Dimacs		UK-2002		Twitter	
WCC	workers	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra
	8	2.110/2	6.249/10	4.279/3	10.18/12	21.93/4	2565/6262	12.77/3	65.92/29	40.23/3	60.35/15
	64	2.110/2	7.284/10	3.627/3	11.18/12	23.70/4	2520/6262	15.66/4	68.99/29	21.05/3	67.92/15
MIS	workers	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS
	8	3.647/3	140.3/51.1	6.333/3	167.5/65.3	9.746/2	175.8/59.1	22.38/3	284.4/87.3	84.50/3	468.7/69.4
	64	3.336/3	151.4/51.1	6.535/3	188.6/65.3	11.11/2	186.5/59.1	20.20/3	253.1/87.3	46.20/3	337.5/69.4
MCST	workers	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS	ReGraph	GPS
	8	3.607/2	39.55/14	8.419/3	138.7/27	10.67/2	442.5/137	30.98/5	341.4/86	181.0/5	635.0/36
	64	3.632/2	38.91/14	9.615/4	75.69/27	11.20/2	413.4/137	34.11/6	280.3/86	129.2/6	252.7/36
TC	workers	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra	ReGraph	PowerLyra
	8	4.387/3	3.181/1	8.730/3	6.418/1	8.967/1	9.834/1	19.05/3	31.14/1	-	-
	64	1.809/3	4.034/1	3.925/3	6.680/1	1.472/1	5.009/1	8.018/3	29.70/1	-	-

the left part of the above inequality to the right part (where a_i represents the decreasing in the degree of iteration i), which guarantees the complexity of our method is much smaller than PowerLyra's.

5.3 Overall Performance

Table 4 demonstrates the results on overall execution time of each application on five datasets, except TC, which fails on Twitter due to exhausted memory (all of our system and other systems fail). For each case, we conduct the execution on ReGraph and every other existing system if it supports this algorithm. The comparison between ReGraph and the existing system that achieves the best average performance (PowerLyra for WCC and TC, and GPS for MIS and MCST) is reported. In order to be fair, the initial pre-processing (e.g., the first partitioning) time of every system is not included. In contrast, for ReGraph, the reported time includes all the cost of following repartitioning and the post-processing.

As we can see, our system can achieve a significant speedup over existing systems, especially when a large number of iterations are needed in their execution. ReGraph outperforms PowerLyra by up to 117.0X and 3.7X on WCC and TC respectively, and the speedup to GPS on MIS and MCST can be up to 45.4X and 41.5X. The reasons of why ReGraph can outperform these existing frameworks are two-fold: *1)* our repartition-centric model needs significantly fewer iterations for convergence; *2)* we adopt a shrink strategy thus the graph size is dramatically decreased while processing, which leads to a much lower computation and communication cost in the following supersteps. A closer look shows that the speedup on TC is caused by the second reason, though more iterations are used, while the speedup on other applications is caused by both reasons.

5.4 Total CPU Execution Time

Besides the overall execution time, our repartition-centric algorithms' capability of scaling down the problem size provides more benefits. Specifically, in a cloud (i.e., pay-per-use) or a multi-tenant environment, after we scale down the problem size and decide to use fewer workers, the unused workers can be assigned to other tasks for better resource utilization.

As a result, we also compare the total resources consumption between ReGraph and other systems by measuring the sum of each

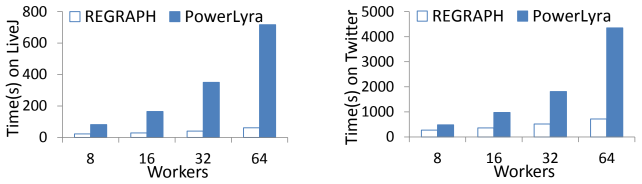

Figure 8: Total CPU time for WCC on LiveJ and Twitter.

worker's execution time. For existing systems that do not scale down the cluster, this number is actually the product of overall execution time and the cluster size. In contrast, this number for ReGraph is calculated by summing up the execution time of each worker. As Figure 8 illustrates, the reduction on this metric is even larger than the speedup on overall execution time (up to 11.6X). Typically, smaller the graph is, or more workers there are in the cluster, we save more consumed resources.

5.5 Time Breakdown and Scalability

To further analyze the performance of ReGraph, we conduct a piecewise breakdown analysis (Figure 9) that mainly divides the overall execution time into three categories: *1).* time to compute, which is mainly spent on local processing; *2).* time to communicate, which is caused by repartition and information exchange between workers; *3).* time to do other things, such as prepare data to be sent. We can find that with the increase of the initial number of workers, the overall execution time is decreased as well as the computation time. Simultaneously, the communication time accounts to more and more percentages of total time, which is a common pattern that limits the scalability of all kinds of distributed graph processing systems. However, since we will scale down the cluster size to a proper size during execution, the increase of communication time in ReGraph is less than other systems.

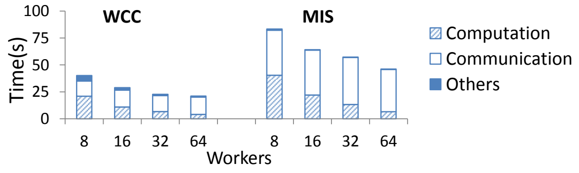

Figure 9: Time breakdown for WCC and MIS on Twitter.

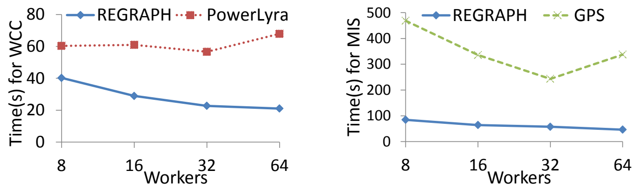

Figure 10: Scalability of WCC and MIS on Twitter.

Actually, increasing number of nodes bring quicker computation as well as two negative effects to ReGraph: *1)* more nodes lead to more communication; and *2)* since the number of vertex replicas increases, for most algorithms, our shrink strategy could delete fewer edges, thus leads to more computation (possibly more iterations). ReGraph doesn't scale well on MCST mainly because of the second reason. Though these potential limitations exist, since our model needs far fewer iterations and automatically scales down the cluster size, thus reduces the total network traffic, ReGraph scales better than other existing systems in most situations, as Figure 10 shows. We believe that for even larger graphs used in industry, our system will be able to scale to hundreds of nodes.

5.6 Implementation in Spark

Table 5: Execution time for Spark-based implementations.

	Pokec	LiveJ	Dimacs	UK-2002	Twitter
GraphX	49.48s	88.57s	>36000s	219.8s	848.8s
Our Model	20.10s	47.44s	127.2s	135.1s	464.7s

As we have discussed in Section 1.3, our repartition-based model is an enhancement rather than a replacement of the traditional models. To prove that it can co-exist with existing frameworks, we implement our repartition-centric WCC algorithm on top of Spark and compare it with GraphX. In fact, this implementation is extremely similar to Algorithm 1 except using Spark RDD instead of our data structures. Table 5 shows the performance evaluated by running both our Spark-based implementation and GraphX on 8 workers, and our model shows a considerable speedup over GraphX.

In fact, the implementation of our repartition-centric WCC algorithm based on Spark is about 5.8-11.6 times slower than the one implemented in ReGraph (based on C++ and MPICH2). This is mainly caused by two reasons: *1).* The object model of JVM causes more overhead on Spark, which leads to a lower performance [37]. This is also the main reason of why GraphX is slower than PowerLyra, although they both adopt the TLAV-based model and need the same number of iterations for convergence; *2).* The semantics of *join* operation is kind of different between in Spark and in ReGraph. In the back-propagation phase, a single *join* operation is needed every time in ReGraph. In contrast, besides *join*, there are also several *map* operations are needed to implement the same semantics in Spark, thus lead to more overhead.

5.7 Discussion

5.7.1 Applicability. The most important limitation of ReGraph is that not all graph processing algorithms can benefit from our proposed shrinking and repartition techniques. For some applications, it's impossible or difficult to find an effective and lightweight shrink strategy. However, as demonstrated by the above sections, we have already found a lot of graph applications that can benefit from our method. Since all of these algorithms prevalently use basic operations, many advanced graph analysis applications are implemented based on them. For example, WCC and MIS are used as sub-routines in many graph clustering algorithms, and TC is also a common sub-routine of many community analysis algorithms.

Nevertheless, it is still important to understand precisely whether an algorithm can or cannot benefit from ReGraph. To this end, we list several prerequisites. Formally, our method is applicable if and only if the graph application can be defined as an iterative graph application $G_{i+1} = f(G_i)$ (where $G_i = (V_i, E_i)$ is a data graph) that: 1) has multiple iterations; 2) the size of graph is shrunk after an iteration, i.e., $|V_i| + |E_i| > |V_{i+1}| + |E_{i+1}|$; and 3) the final result is not changed after shrinking.

1). The alternately-shrink-repartition technique leads to faster convergence and less computation cost of later supersteps. Obviously, it works only when the algorithm is iterative, i.e., the algorithm consists of multiple iterations. For example, the TC algorithm in [27] which always takes only one round could not fit in our work, thus we optimize an alternative iterative algorithm in Section 5.2.4.

2). The second prerequisite is needed for reducing the overhead of repartitioning (thus the overhead will not offset its benefits) and the cost of later computation. Unfortunately, some algorithms don't meet this requirement. PageRank is a typical example because it needs all edges to update every vertex's state in each superstep so that the technique doesn't work for it.

3). The third prerequisite guarantees the correctness of enabling our technique. If the changes in graph structure of each iteration don't affect the final result, the algorithm will always end up with a correct answer, which can be proved by mathematical induction. Under this prerequisite, the best approach of shrinking the graph is to delete as many edges/vertices as possible.

Although the above list of prerequisites seems tight, fortunately, there are in fact plenty of graph algorithms can benefit from our framework. The reason is that many graph applications can be solved by **greedy algorithms** that can immediately decide whether an edge is needed or not with only the local information. That is to say, this kind of algorithms can make the locally optimal choice at each stage with only the given sub-graph of each node, and then the other unneeded vertices/edges can be safely removed for shrinking the graph. It is easy to see that most of the algorithms we evaluated (WCC/MIS/MCST) are typically solved by sequential greedy algorithms. There are also many algorithms that solve different kinds of applications fit into this category, such as graph coloring [26], unweighted maximal matchings, maximum weighted matching, minimum edge cover, minimum cut [18] and so on. Another kind of graph applications that can take advantage of our framework are problems that can be partitioned into a series of disjoint sub-tasks that each sub-task related to only a sub-graph of the original graph. For example, the Triangle Counting problem can be solved by counting triangles related to a small portion of the vertices (low-degree vertices in our algorithm) in each step, which shows that the applicability of our method is not even limited by the applicability of greedy algorithms.

Moreover, to validate that even in the situation that the proposed technique is not applicable, our work is still meaningful, we implement the PageRank algorithm provided by Giraph++ in ReGraph,

Table 6: Execution time for PageRank (10 iterations).

	Pokec	LiveJ	Dimacs	UK-2002	Twitter
PowerLyra	6.260s	12.18s	37.65s	37.57s	113.8s
Giraph++	96.43s	264.0s	1128s	1123s	9802s
ReGraph	2.490s	7.331s	30.96s	24.90s	85.66s

Table 7: Code size for five applications (in line).

	WCC	MIS	MCST	TC	PageRank
TLAV	208	385	1565	678	279
ReGraph	357	295	253	307	168

and compare its performance with TLAV-based and TLASG-based systems. As Table 6 demonstrates, ReGraph could be as fast as other works all the time. In fact, it could achieve a small speedup over TLAV-based system PowerLyra, since local communication is cheaper. And it's much faster than Giraph++ because of our C++ and MPI based implementation, though they use the same algorithm. In conclusion, ReGraph provides an alternative to insert repartition and shrink procedures during processing, as well as keeps the ability to implement traditional TLASG-based algorithms efficiently.

5.7.2 Programming Complexity. The fundamental programming difficulty of the other TLAV/TLASG frameworks is that they need to coordinate the information of different graph parts that hosted on different machines. Although a TLASG API enables users to design smarter algorithms for a single sub-graph, it does not avoid the complexity of coordinating different sub-graphs that hosted on different machines. In contrast, our method always scales down the size of the problem to a single machine. As a result, our algorithms of solving MIS/MCST/TC are not very different from the single-machine algorithms, which are much simpler than the parallel algorithms (e.g., the parallel Boruvka algorithm for solving MCST) used by the other systems.

To further demonstrate the programming complexity of our work, we count the code size for each application we implemented in ReGraph and compare it with the representative TLAV-based system (PowerLyra or GPS). Although the TLAV model is proved to be convenient for programming, results in Table 7 show that the programming model of our framework enables users to implement their own repartition-centric algorithms very efficiently.

6 RELATED WORK

Dynamically Capturing the Working Set Dynamically capturing the working set by eliminating parts of the graph that remain inactive (or that don't contribute) is a promising direction to reduce the scale of the problem and has been explored in [34] by temporarily not loading the inactive vertices. However, this work is proposed based on an out-of-core environment that the whole graph is stored in the local machine, thus it's quite different with our work in many important aspects, as demonstrated by Table 8. Moreover, we can combine the methods of these two works for extending the scope of applications of our framework. This will be the future work of us.

Besides, a general algorithmic design technique *filtering* has been proposed by [18] to solve graph problems in MapReduce,

Table 8: The similarities and distinctions between Dynamic Partitions [34] and ReGraph.

	Dynamic Partitions [34]	ReGraph
Dropping edges	dynamic	dynamic
Reducing computing resources	allowed	allowed
Re-addition of dropped edges	allowed	disallowed
Repartitioning	not required	required
Applicability	asynchronous	Section 5.7.1
Execution environment	out-of-core	distributed
Programming model	TLAV	TLASG

which is similar to our shrink strategy. However, it mainly focuses on theoretical analysis and algorithms that can be implemented with only simple distributed Map and Reduce primitives, so its applicability is much smaller than ours.

Distributed Graph Systems There are many distributed graph processing systems have been proposed to process large graphs. Pregel is the earliest system that proposed the TLAV-based model, which leads to several inherited systems [1, 14, 20, 25, 40]. To overcome the limitation of TLAV-based frameworks, the TLASG-based model was proposed by Giraph++, and inherited by some works [29, 38]. GRAPE [12] is also a parallel graph system but differs from prior works in its ability to parallelize existing sequential graph algorithms as a whole. Though these systems are different from each other in terms of programming models or implementations, they are still different from our work. In ReGraph, repartition procedures can be inserted proactively between iterations to scale down the problem or re-balance the workload. Moreover, we support the reduction of cluster size during execution, thus leads to less resource consumed.

Besides, there are also some existing works [28, 32, 33] are proposed to support dynamic graphs, which is not currently supported in our framework. But, we think that the principle of frequently repartitioning the graph to maintain the quality of partition may be even more important for dynamic graphs. Therefore, this will be our future work.

7 CONCLUSION

This paper advocates the use of repartition-centric graph algorithms. Through deleting the unnecessary edges, we show that the increased overhead of proactively repartitioning the graph can be worthwhile. Moreover, to facilitate the implementation of repartition-centric graph algorithms, we design and implement ReGraph, which is a novel graph processing system that provides a set of Spark-fashion APIs. Our evaluation results demonstrate that ReGraph requires fewer iterations to converge and hence can achieve a better performance.

ACKNOWLEDGMENTS

We thank our shepherd Prof. Keval Vora and the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280, U1435216, 61402198), National Basic Research (973) Program of China (2014CB340402).

REFERENCES

[1] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[2] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. 2011. Tolerating the community detection resolution limit with edge weighting. *Physical Review E* 83, 5 (2011), 056119.

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM, New York, NY, USA, 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM, New York, NY, USA, 595–601.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, New York, NY, USA, 1:1–1:15.

[7] Nicos Christofides. 1976. *Worst-case analysis of a new heuristic for the travelling salesman problem.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group.

[8] Sun Chung and Anne Condon. 1996. Parallel implementation of Bouvka's minimum spanning tree algorithm. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE, 302–308.

[9] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. 1994. The complexity of multiterminal cuts. *SIAM J. Comput.* 23, 4 (1994), 864–894.

[10] Dimacs. 2005. The Center for Discrete Mathematics and Theoretical Computer Science. (2005). http://www.dis.uniroma1.it/challenge9/download.shtml

[11] Jean-Pierre Eckmann and Elisha Moses. 2002. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences* 99, 9 (2002), 5825–5829.

[12] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, New York, NY, USA, 495–510.

[13] Harold N Gabow and Robert Endre Tarjan. 1985. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences* 30, 2 (1985), 209–221.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 17–30.

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 599–613.

[16] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600.

[18] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, New York, NY, USA, 85–94.

[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[21] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing* 15, 4 (1986), 1036–1053.

[22] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 135–146.

[23] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 25:1–25:39.

[24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, New York, NY, USA, 456–471.

[25] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, New York, NY, USA, 22:1–22:12.

[26] Semih Salihoglu and Jennifer Widom. 2014. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment* 7, 7 (2014), 577–588.

[27] Thomas Schank. 2007. Algorithmic aspects of triangle-based network analysis. *Phd in computer science, University Karlsruhe* 3 (2007).

[28] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, New York, NY, USA, 417–430.

[29] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*. Springer International Publishing, Cham, 451–462.

[30] Kenneth J Supowit, David A Plaisted, and Edward M Reingold. 1980. Heuristics for weighted perfect matching. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 398–419.

[31] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[32] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 32.

[33] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 237–251.

[34] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 507–522.

[35] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of small-world networks. *nature* 393, 6684 (1998), 440.

[36] Howard T Welser, Eric Gleave, Danyel Fisher, and Marc Smith. 2007. Visualizing the signatures of social roles in online discussion groups. *Journal of social structure* 8, 2 (2007), 1–32.

[37] Reynold Xin and Josh Rosen. 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. (2015). https://databricks.com/blog/2015/04/28.

[38] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[39] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 608–621.

[40] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 301–316.