V-Combiner: Speeding-up Iterative Graph Processing on a Shared-Memory Platform with Vertex Merging

Azin Heidarshenas
heidars2@illinois.edu
University of Illinois at Urbana-Champaign

Serif Yesil
syesil2@illinois.edu
University of Illinois at Urbana-Champaign

Dimitrios Skarlatos
skarlat2@illinois.edu
University of Illinois at Urbana-Champaign

Sasa Misailovic
misailo@illinois.edu
University of Illinois at Urbana-Champaign

Adam Morrison
mad@cs.tau.ac.il
Tel Aviv University

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

An iterative graph algorithm applies a vertex update operation to all vertices in a graph in every iteration. For large graphs, this computation is costly. However, in practice, not all the updates contribute equally to the end result and, in fact, an exact result may not be needed. In this work, we leverage these insights to speed-up iterative graph algorithms. We propose a mechanism to identify the less important vertices and omit computations for them.

Our scheme, called *V-Combiner*, is a deterministic, fast, and application-transparent technique to construct an approximate graph to enable faster execution. The main idea behind V-Combiner is to merge certain vertices into *hubs*, which are vertices that have many connections and contribute heavily to the end result of the algorithm. We also propose an inexpensive *correction* step to recover the contribution of the merged vertices to get higher accuracy.

We evaluate V-Combiner on 4 different applications and 5 datasets. For 44-threaded runs, V-Combiner achieves an average end-to-end speedup of 1.25× over the conventional system, with an accuracy of 91.8%. It also shows a better performance-accuracy trade-off than the existing *sparsification* and *k-core* techniques.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

Graph processing, Shared-memory platforms, Approximations

ACM Reference Format:
Azin Heidarshenas, Serif Yesil, Dimitrios Skarlatos, Sasa Misailovic, Adam Morrison, and Josep Torrellas. 2020. V-Combiner: Speeding-up Iterative Graph Processing on a Shared-Memory Platform with Vertex Merging. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392739

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392739

1 INTRODUCTION

Many graph processing applications used in the machine learning, social network, computational biology, and financial system domains are inherently iterative. Examples include Belief Propagation [18, 19], Community Detection [45], Page Rank [34], and Hyperlink-Induced Topic Search [22]. They typically have a property that gets propagated across vertices, and a metric for convergence to a solution, which determines their time complexity.

Iterative algorithms are inherently costly for large graphs. For example, running Page Rank on billion-vertex graphs can take minutes in a 24-core shared-memory system [25]. In many environments, this latency is unacceptable. On the other hand, many applications can trade a small amount of accuracy for improved performance [15, 32]. For example, in Page Rank, the user is often interested in a very small subset of the computation results [10, 43] — almost all queries only require to know the top-ranked pages. Further, vertex classification algorithms such as Community Detection and Belief Propagation can tolerate small errors as long as the final inferred vertex labels remain correct.

Providing high accuracy in approximate iterative graph computations is challenging, as the error introduced in one vertex can propagate to its neighbors (and eventually to all other reachable vertices), and gets accumulated across iterations. For this reason, approximations used in non-iterative graph algorithms such as Single-Source Shortest Path (SSSP) [41] or Triangle Counting [14] are not suitable for iterative graph algorithms.

Previous works for iterative graph algorithms [12, 25, 33, 38] apply program approximation techniques such as loop perforation [39] and task skipping [36] to the input graph [25, 33] or to the program [12, 38]. These techniques speed-up the program. However, by dropping random vertices or connections in the graph, or skipping their computation as the program runs, these techniques introduce non-determinism, which makes debugging difficult and may cause variability in the outputs of the application.

Graph summarization techniques [28] generate a simpler form of the graph for faster processing and/or more efficient storage. The most popular of these techniques include sketching [1, 7, 30, 37], sparsification [3, 15] and k-core decomposition [13, 21, 35]. Sketching techniques summarize the graph information into a data structure that can be queried in the future to provide a fast approximate answer after a few simple computations. However, sketching techniques have a large overhead and, therefore, cannot be used for performance-intensive tasks. The sparsification and k-core decomposition techniques have a much smaller overhead. However, the

resulting graphs have performance and accuracy limitations for certain applications. As we will see, these two techniques are unable to achieve both high speedup and high accuracy.

Our Work. In this paper, we present *V-Combiner*, a general, deterministic technique for speeding-up iterative graph-processing applications, while maintaining high-accuracy results. V-Combiner is motivated by the observation that not all vertices contribute equally in iterative graph processing algorithms. Therefore, the key technical challenge is identifying and removing the vertices with a small contribution, and still maintaining the graph properties that dictate the accuracy of the graph algorithm.

During pre-processing, V-Combiner *merges* some vertices into their neighboring *hubs*, which are vertices with many connections. It then executes the unmodified original algorithm on the reduced graph, speeding-up execution. Finally, it corrects the final result of the program to account for the effect of the merged vertices.

V-Combiner has the following characteristics:

- **Application Transparency.** Unlike previous work that requires changing the implementation of the graph algorithm [12, 38], in V-Combiner the application is completely unaware of the merging step applied to the input graph. The application treats the approximate graph in the same way as the original graph. This allows the programmer to develop graph algorithms as before.
- **Low Overhead, High Performance, and High Accuracy.** V-Combiner has simpler pre-processing and lower overhead than previous work such as k-core [13, 21, 35], GraphTuner [33] and Input Reduction [25]. V-Combiner speeds up the execution by creating a smaller approximate graph that needs fewer updates. It retains high accuracy by maintaining important graph properties and using a final correction step.
- **Deterministic Behavior.** V-Combiner uses a deterministic merging mechanism that produces identical approximate graphs every time. All previous algorithms except k-core are non-deterministic [12, 15, 25, 33, 38]. Compared to k-core, V-Combiner algorithm maximizes the connectivity of the reduced graph.

Results. We evaluate V-Combiner on a 44-core shared-memory machine running four iterative applications: Belief Propagation, Community Detection, Hyperlink-Induced Topic Search, and Page Rank. We execute each application with 5 real-world data sets. Our main results are:

- Considering the algorithm-time only, V-Combiner speeds-up the computation by an average of 1.55× over the exact baseline algorithm at an accuracy level above 90%, compared to average speedups of 1.46× with sparsification and 1.36× with k-core.
- Considering the end-to-end execution time, which includes one-time overheads, the average speedup of V-Combiner over the exact baseline algorithm is 1.25×, with an accuracy of 91.8%. The performance of V-Combiner is equal to sparsification and significantly better than k-core. Unlike sparsification and k-core, it can successfully meet the accuracy bound on all the benchmarks.
- A trade-off exploration shows that V-Combiner can produce a better set of points in the performance-accuracy trade-off space than the other algorithms in the region above 90% accuracy.
- V-Combiner obtains better load balancing, preserves the average length of the paths, produces deterministic results (unlike sparsification), and performs less work at high accuracy due to high connectivity (unlike k-core).

Contributions. The main contributions of this paper are:

- We analyze and compare existing techniques for approximating iterative graph algorithms.
- We develop V-Combiner, a novel general approximation technique to improve the performance of iterative graph algorithms with acceptable accuracy losses.
- We evaluate V-Combiner on a shared-memory machine and compare it to state-of-the-art approaches for approximate graph processing.

2 BACKGROUND

A graph G consists of a set of vertices or vertices (V) and edges (E). A graph can be directed or undirected. In directed graphs, an edge can be incoming or outgoing. In undirected graphs, edges do not have a direction. Therefore, each edge can be treated as two logical edges: an incoming and an outgoing one. We refer to the incoming edges of a vertex as *in-edges*, and to outgoing edges as *out-edges*. The vertices at the other end of in-edges are the *in-neighbors* of a vertex. Similarly, the end points of out-edges are the *out-neighbors*. The number of in-edges of a vertex is its *in-degree*, and the number of out-edges is its *out-degree*. For undirected graphs, the degree of a vertex is equal to the number of physical edges of the vertex, and is equal to the vertex's in-degree and to the vertex's out-degree. A *path* in the graph is a sequence of edges that connects a series of distinct vertices.

2.1 Iterative Graph Algorithms

Algorithm 1 shows the template of an iterative graph algorithm. The algorithm is composed of multiple iterations. In an iteration, the algorithm goes over all of the vertices in the graph (Line 3). For each vertex, it applies an *update function*, which gathers information from the neighbors of the vertex (Lines 5-7) — and potentially from the edges. The value of each neighbor vertex is first multiplied by W, which is a vertex-specific constant. For example, in Page Rank, W for neighbor vertex v is equal to $\frac{1}{out-degree[v]}$. The result is then aggregated, using an operand that varies across applications (e.g., addition or multiplication) (Line-7). After that, the update function refines the result stored for the vertex using C_1 and C_2 (Line 8). For Page Rank, C_1 and C_2 are commonly set to 0.85 and 0.15. In every iteration, a *change* variable (Line 9) accumulates the difference in the values of each vertex before and after the update function. The iterations stop when change is less than a threshold — i.e. the convergence criterion is met (Lines 10-11).

Algorithm 1: Iterative graph algorithm.

```
input : Graph, Threshold, MaxIter, W, C1, C2
output: values
1   for i ← 1 to MaxIter do
2       change ← 0;
3       for u in Graph.Vertices do
4           old ← values[u];
5           gatheredVal ← 0 or values[u];
6           for v in Graph.Neighbors(u) do
7               gatheredVal ← gatheredVal ⊕ (W × values[v]);
8           values[u] ← C1 × gatheredVal + C2;
9           change ← change + | old − values[u] |;
10      if change < threshold then
11          break;
```

The update function creates a notion of information propagation, which can be weights or probabilities. Information propagation generally follows the edges. In undirected graphs, the information propagates in both directions of the edges, whereas in directed graphs, it can propagate in the forward or reverse direction of the edges. Overall, different algorithms differ in the definition of the update function (which is also associated with the direction of edges) and the convergence criterion. In this work, we select four well-known algorithms with different characteristics.

Belief Propagation (BP): BP performs inference on a graphical model [18]. BP calculates a random variable (i.e., the belief) for each vertex, which indicates its most probable state. In BP, the update operation works in both directions, and so propagates the information.

Community Detection (CD): CD finds the community structure of a network [23, 26, 42, 45]. It uses the label propagation method. It takes a directed graph with a set of labeled vertices as input, and propagates the known community IDs to the rest of the vertices in the graph. The update operations happen only in the direction of edges.

Hyperlink-Induced Topic Search (HITS): HITS takes as input a directed graph (e.g., a social or web network) and identifies the most relevant (*aka* authoritative) users/pages given a specific topic [2, 22]. It uses the graph structure to obtain two scores for each vertex, namely the *authority* and *hub*. The authority score of a vertex is computed using the hub scores of its in-neighbors, while its hub score is computed using the authority scores of its out-neighbors. The update operations happen both in the direction of edges to calculate authority scores, and in the reverse direction to compute hub scores.

Page Rank (PR): PR [34] computes the ranks of all the vertices based on the graph structure. The input can be any directed graph, such as a social or web graph. At every iteration, the Page Rank score of a vertex is computed by summing up the Page Ranks of its in-neighbors divided by their out-degrees. The update operations happen only in the direction of edges.

Other Graph Algorithms. Our focus is on iterative graph algorithms whose time complexity is determined by their convergence or number of iterations. A number of machine learning algorithms on graphs also belong to this category. However, our observations and techniques do not apply to Triangle Counting, Single Source Shortest Path (SSSP), Betweenness Centrality, and similar problems because their time complexity is independent of the number of iterations. Moreover, it is more challenging to provide high accuracy for iterative algorithms, as the error in one vertex gets propagated and accumulated to all the other reachable vertices across iterations. Hence, approximations proposed for algorithms such as SSSP [41] and Triangle Counting [14] are not suitable for iterative graph algorithms. Finally, the running times of non-iterative algorithms are relatively low compared to the iterative ones. Hence, there is less opportunity to accelerate them using approximations.

2.2 Graph Pre-processing

A graph application has two main parts: pre-processing and computation. Existing work often ignores the pre-processing time and only focuses on optimizing the computation. However, there exists a trade-off between pre-processing time and algorithm execution time [29]. Thus, it is important to account for the pre-processing time when optimizing computation time. Figure 1 shows that the pre-processing time is a significant fraction of the total execution time for some of our applications. These results are measured on a machine with 44 threads (Section 7.1).

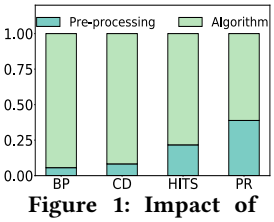


Figure 1: Impact of pre-processing time.

3 OBSERVATIONS

V-Combiner is based on several observations we make on graphs.

3.1 Not All Vertices Contribute Equally

In Algorithm 1, the number of updates generated at a vertex is equal to the number of neighbors it has. For directed graphs, it is equal to the in-degree if updates happen in the direction of edges, or to the out-degree if they happen in the reverse direction. Therefore, vertices with higher degrees have a higher impact on how information is propagated throughout the graph.

Typically, in real-world social and web graphs, a significant portion of the vertices have small degrees. For example, for the large Twitter [27] and PLD [31] graph datasets, we find that, on average, 97.4% of the vertices are connected to at most 100 other vertices. In contrast, 0.00005% of the vertices are connected to at least 100,000 vertices. They are typically identified as *hubs*, and substantially impact how information is propagated.

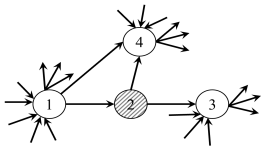


Figure 2: Illustration of different vertex connectivity.

Figure 2 shows an example subgraph. It has three vertices that are highly connected (①,③, and ④), and one that is not (②). The former have more impact on information flow.

3.2 A Subset of Results Is All That Is Needed

In many real-life applications of graph processing, the application computes over many vertices, but the user is only interested in a very small subset of the results. The subset of most popular vertices is enough in many scenarios. For example, a company may want to identify its most influential customers, or a researcher may want to find the top authoritative researchers in some domain in the DBLP network [43] or the top authoritative pages in a hyperlink environment [22]. In fact, the Page Rank algorithm in [10] returns only the top ranked vertices. Therefore, it should often be acceptable for an approximate implementation of the algorithm to return the same top ranked vertices even though the ranking of the less popular vertices may be different.

3.3 Applications Can Tolerate Small Errors

In applications such as Belief Propagation and Community Detection, the goal is to infer the broad category of each vertex, rather than to compute an exact value for each vertex. Each vertex has a vector of values, where each element of the vector corresponds to the probability of the vertex belonging to a specific category. The highest probability indicates the category of the vertex. As such, small errors in the probability values can be tolerated as long as the inferred vertex category remains the same.

3.4 Removing Some Vertices Can Be Effective

Removing some vertices or edges reduces the number and cost of update operations. If the high-level structure of the graph is preserved, the computation on the reduced graph could be a close

approximation of that on the original graph. Our intuition is that, since *hub* vertices shape most of the information flow in the graph, vertices from their neighborhood that have a small degree can be *merged* into the hubs, and the overall flow of information in the graph be only marginally affected. Merging a vertex into a hub involves removing the vertex and adding all (or a subset) of its edges to the hub vertex.

4 LIMITATIONS OF CURRENT TECHNIQUES

For the types of algorithms that we consider, there are two main existing techniques that prune graphs. They are *Sparsification* [3, 15] and *K-core* [13, 21, 35]. These techniques have a few limitations, which act as a motivation for our work. In this section, we discuss these limitations. Later, in Section 7.1, we improve the two techniques and, in Section 8, evaluate them against V-Combiner.

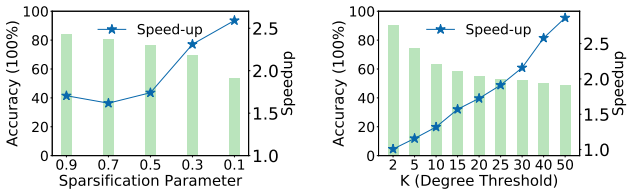
Sparsification selects some edges in the graph using a probabilisic method and prunes them from the graph. It does not prune any vertices. The pruned edges are chosen based on a sparsification parameter, the shape of the graph, and some properties of the edges. The pruning can be made more or less aggressive. Details are provided in Section 7.1.

Since only edges are removed and not vertices, the algorithm still has to loop over all the vertices. In addition, by removing edges, it typically increases the length of the paths between vertices. As a result, the number of iterations that a graph algorithm requires to converge typically increases. For these reasons, it may be difficult to obtain a high speedup with a high algorithm accuracy.

K-core takes a graph and removes as many vertices as needed so that the remaining vertices have a degree equal or greater than k. The result is the *k-core* graph. This technique prunes both vertices and edges. Higher values of k imply that more vertices and edges are dropped.

Since both edges and vertices are removed, this technique can deliver higher speedups. It reduces the work without increasing the number of iterations until convergence. However, the downside is that, as k increases, more and more of the vertices remaining in the graph become disconnected. As the graph loses connectivity, the accuracy of the algorithm suffers.

As an illustration, Figures 3a and 3b show representative scenarios. They correspond to runs of Page Rank (*PR*) using sparsification (Figure 3a) and Community Detection (*CD*) using k-core (Figure 3b). Section 7 describes the input graph used (*PLD*), the parameters used, and the 44-core machine running the experiments. The figures show, for different levels of pruning (X axis, where the level of pruning increases toward the right side), the accuracy as defined in Section 7 (left Y axis) and the speedup relative to the execution using the full graph (right Y axis).



(a) Sparsification with PR (b) K-core with CD

Figure 3: Illustration of sparsification and k-core.

We observe that, as the extent of pruning increases, the speedup increases, but the accuracy decreases substantially. For these applications and input graph, we see that even with a small degree of pruning (left side of X axis), the accuracy is already 90% or lower and, for k-core, the speedup is minuscule.

5 VERTEX MERGING WITH V-COMBINER

Based on the previous discussion, an effective approach to reduce the graph size needs to involve removing both edges and vertices. In addition, as we remove them, we have to be careful to eliminate (or at least reduce to a minimum) the possibility of disconnecting parts of the graph. Consequently, our proposal is to merge some vertices into nearby hubs, adding some of the edges of the removed vertices to the hub. The result is both accuracy and speedups.

Our technique is called *V-Combiner*. V-Combiner proceeds in two steps. First, it runs a *vertex merging* algorithm during graph preprocessing time, to identify vertices that have few connections and can be merged into their neighboring hubs. Then, after the application completes execution, it runs a correction or *recovery* algorithm to quickly obtain acceptable values for the merged vertices.

Figure 4 shows the end-to-end execution timeline for both the conventional exact approach and V-Combiner. The exact approach includes pre-processing time (i.e., when the graph is built) and compute time (i.e., when the graph algorithm runs). The V-Combiner approach contains pre-processing time (which includes vertex merging and building the graph), compute time (i.e., when the graph algorithm runs), and post-processing time (i.e., when the recovery takes place). The shaded boxes are V-Combiner-specific steps.

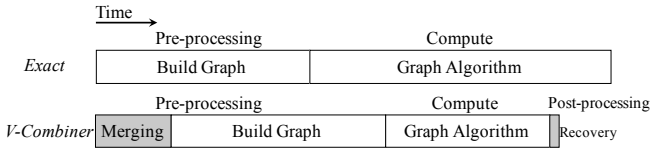


Figure 4: End-to-end execution steps of a graph algorithm.

The performance benefits of V-Combiner come from the reduced algorithm time due to the reduced number of vertices and edges. More specifically, the graph algorithm can converge with fewer iterations (because of the reduced vertices) and fewer update operations in each iteration (because of the reduced edges). Ideally, the reduction in algorithm time should more than offset the additional overheads of the pre- and post-processing steps.

Support for Different Graphs and Algorithms. V-Combiner can be applied to both directed and undirected graphs with minimal changes. Moreover, it can be used for a variety of graph algorithms with different types of update propagation. For example, in Page Rank, updates are always propagated from in-neighbors, whereas in HITS [22], updates are propagated both from in- and out-neighbors. Based on the update propagation, we classify algorithms into *one-way* propagation and *two-way* propagation. For undirected graphs, the propagation is naturally two-way. When merging, V-Combiner takes into account the direction of update propagation used by the algorithm, and tries to maintain the connectivity of the graph. The goal is for the updates to continue to propagate through the edges even after certain vertices are removed.

5.1 Merging Approach

During the merging step, V-Combiner proceeds to identify a subset of the high-degree vertices that we call *Supernodes*. Then, for each supernode, it considers the vertices in its input neighborhood (or in its output neighborhood, if the updates are propagated from out-neighbors). If these vertices have low in- and out-degrees, we say that they are *Subnodes* of the supernode. The assumption is that the contribution of such vertices to the final result of the algorithm is likely to be small. Hence, the subnodes are merged into the supernode, effectively being pruned away. The supernode takes some of the edges of the merged subnodes, as explained in Section 5.2.

The resulting graph has a set of supernodes with now higher degrees, connected with what we call *Regular* vertices. The resulting graph has a similar structure to the original one, but with many fewer vertices and edges.

We now give the precise definitions of supernodes, subnodes, and regular vertices in a given graph G. In the definitions, we use three thresholds called α, β, and π. The α and β thresholds are the lower and upper thresholds, respectively, for the degree of a vertex that qualifies as a supernode. The π threshold is the upper threshold for the degree of a vertex that qualifies as a subnode. We describe how to obtain such thresholds in Section 5.6.

Definition 1: Supernodes (V_{sup}) are vertices with an in-degree higher than α and lower than β,

$$V_{sup} = \{\ v \mid v \in G.V\ \wedge\ InDegree(v) > \alpha\ \wedge\ InDegree(v) < \beta\ \}.$$

Vertices with in-degree higher than or equal to β are not considered supernodes. This is because we do not want them to increase their degree further by taking-in edges from merged subnodes. Vertices with very high degree can cause load imbalance and slow down the execution. Hence, these vertices remain unchanged in the graph.

For two-way algorithms in directed graphs, supernodes are vertices where the sum of in-degree and out-degree is higher than α and lower than β. Further, a supernode is an *in-supernode* if its in-degree is higher than or equal to its out-degree, and an *out-supernode* if the opposite is true.

For undirected graphs, supernodes are vertices with a degree between α and β.

Definition 2: Subnodes (V_{sub}) are vertices that have an in-degree lower than π (where $\pi < \alpha$), an out-degree lower than π, and at least one output that connects them to a supernode,

$$V_{sub} = \{\ v\ \mid\ v \in G.V \wedge InDegree(v) < \pi \wedge OutDegree(v) < \pi \wedge \exists w \in V_{sup}\ .\ w \in OutNeigh(v)\ \}.$$

A vertex with an in-degree or an out-degree higher than π is not a subnode because it is too important to be merged into a nearby supernode. Also, it is possible that a subnode is connected to two supernodes; in this case, it can be merged into either with a deterministic algorithm.

For two-way algorithms in directed graphs, subnodes are vertices where the sum of in-degree and out-degree is less than π, and have at least one edge that connects them to a supernode. Further, a subnode is an *in-subnode* if it has at least one output that connects it to an in-supernode, and is an *out-subnode* if it has at least one input that connects it to an out-supernode. A subnode can be both an in-subnode and an out-subnode.

For undirected graphs, subnodes are vertices with a degree lower than π and at least one edge that connects them to a supernode.

Definition 3: Regular vertices (V_{reg}) are the remaining vertices,

$$V_{reg} = G.V - (V_{sub}\ \cup\ V_{sup}).$$

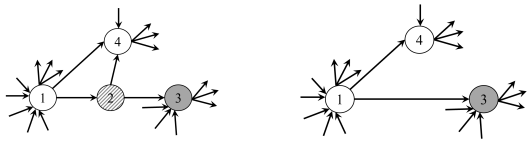
V-Combiner leaves those vertices intact.

5.2 Vertex Merging Algorithm

V-Combiner performs vertex merging to generate the approximate graph. The algorithm involves merging each subnode into a neighboring supernode. In the following, without loss of generality, we describe the vertex merging algorithm assuming one-way graph algorithms in directed graphs where updates are propagated from in-neighbors. In Section 5.5, we describe scenarios with other types of directed and undirected graphs.

Merging a subnode into a supernode involves removing the subnode and altering the edges as follows. First, the input edges of the subnode now become input edges of the supernode. This operation may create duplicated edges — i.e., multiple edges connecting the same input vertex to the same output vertex; such duplication will be later eliminated when the graph is built. Second, the output edges of the subnode are dropped. V-Combiner chooses this design over a possible alternative where the output edges of the subnode become output edges of the supernode. Such alternative is undesirable because it could create a new, previously-nonexistent path.

Figure 5 shows an example of vertex merging. Figure 5(a) shows an original subgraph, where V-Combiner wants to merge subnode ② into supernode ③. Figure 5(b) shows the approximate graph after merging.



(a) Original **(b) Approximate**

Figure 5: Vertex merging in V-Combiner.

As V-Combiner merges subnode ② into ③, it transforms edge ① → ② into ① → ③. Next, V-Combiner discards edge ② → ④. Had V-Combiner chosen to transform it into an edge ③ → ④, it would be creating a new, previously nonexisting path between ③ and ④. Hence, V-Combiner does not create such an edge.

Algorithm Description. Algorithm 2 shows the pseudo code of the merging algorithm in V-Combiner. The algorithm takes as input the number of vertices, the list of edges, and the in-degrees and out-degrees of the vertices. The output of the algorithm is the new in- and out-degrees of the vertices, and an array called *superNodes*. This array has as many entries as vertices in the original graph. If a vertex has become a subnode, its entry in *superNodes* has the ID of its supernode; if a vertex has become a supernode, its entry in *superNodes* has its own ID; for the remaining vertices, the entry in *superNodes* holds -1. The output of the algorithm will be later used in the step that builds the graph.

The merging algorithm consists of three sections. Each section can be executed in parallel:

- *Identify supernodes:* First, all vertices are scanned in parallel to identify any vertex that has an in-degree higher than α and lower than β (Lines 1-3).
- *Identify subnodes:* The second parallel section (Lines 4-10) consists of processing the list of edges in parallel. For every edge e, such

Algorithm 2: Merging algorithm in V-Combiner.

```
inputs : N (number of vertices), edges (list of edges), inDegrees,
         outDegrees, α, β, π
outputs: newInDegrees, newOutDegrees, superNodes
1   for i ← 0 to N − 1 do
2       if inDegrees [i] > α and inDegrees [i] < β then
3           superNodes [i] = i
4   for e in edges do
5       if superNodes [e.dst] = e.dst
6         and inDegrees [e.src] < π
7         and outDegrees [e.src] < π then
8           superNodes [e.src] = e.dst;
9           newInDegrees [e.src] ← 0;
10          newOutDegrees [e.src] ← 0
11  for e in edges do
12      if e.dst is a subnode and e.src is NOT a subnode then
13          newInDegrees [superNodes[e.dst]] += 1
14      if e.src is a subnode and e.dst is NOT a subnode then
15          newInDegrees [e.dst] -= 1
```

that $e.dst$ is a supernode, we check if $e.src$ qualifies to become a subnode. If so, the *superNodes* entry of $e.src$ is set to $e.dst$. Also, we prune the in- and out-edges of $e.src$. Therefore, the new in- and out-degrees of $e.src$ are set to 0.

- *Merge Vertices and Update Degrees:* Finally in Lines 11-15, the algorithm computes the new in- and out-degrees of all affected vertices. Recall that merging affects both the in-neighbors and the out-neighbors of the subnode. Each in-neighbor has to be connected to the supernode. Hence, we increase the in-degree of the supernode in Lines 12-13. Moreover, the out-edges of the subnode have to be eliminated, and the out-neighbors have to update their in-degrees (Lines 14-15).

Delta Graph Construction. After merging, when the graph is built, V-Combiner also constructs a small graph named the *Delta* graph. The Delta graph contains only the subnodes and their immediate in-neighbors. The Delta graph will be used to generate good final values for the subnodes in the recovery step.

5.3 Recovery Step

Recall that the subnodes do not have any values after the graph algorithm completes. The goal of V-Combiner's recovery step is to assign the final values to these subnodes.

To compute the values of the subnodes, the recovery algorithm takes the Delta graph and proceeds as follows. The in-neighbors of subnodes in the Delta graph are given the values computed by the computation on the approximate graph. Such values are close to their exact values. Then, we run the recovery algorithm, which is specified by the developer. It simply uses the basic operator of the corresponding graph algorithm to generate the values of the subnodes using the values of their in-neighbors in the Delta graph.

5.4 Overall V-Combiner Algorithm

Algorithm 3 shows the overall V-Combiner algorithm. First, the subnode vertices are merged using Algorithm 2. Then, the output of the Merging algorithm, including the *superNodes* array, is passed to a Build step. This step constructs both the Delta and the approximate graphs (Line 2). After that, the graph algorithm, outlined in Algorithm 1 executes on the approximate graph using the

Algorithm 3: Overall V-Combiner algorithm.

```
input : merging_arguments, algo_params
output: final_results
1 merging_output = Merging(merging_arguments);
2 approx_graph, delta_graph = Build(merging_output);
3 results = GraphAlgo(approx_graph, algo_params);
4 final_results = Recovery(results, delta_graph, algo_params);
```

user-specified algorithm parameters (Line 3), and returns its results. Finally, in Line 4, the recovery algorithm receives these results and recovers the values for the subnode vertices using the Delta graph.

5.5 Other Scenarios of the Merging Algorithm

Section 5.2 only described how to perform merging for the most common scenario — i.e. directed graphs and one-way algorithms. In the example algorithm, V-Combiner picks subnodes from the inputs of the supernode, and merges subnodes into supernodes in the forward direction of the edges. In the case where information flows only in the reverse direction of the edges in a directed graph, V-Combiner would pick subnodes from the outputs of the supernode. However, we are unaware of an example for this scenario.

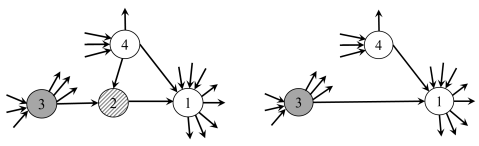
Table 1 shows a taxonomy with two other possible scenarios that V-Combiner can support: (i) directed edges where the information flows in both directions; (ii) undirected edges. V-Combiner supports both scenarios using the same merging techniques.

Table 1: Graph processing scenarios V-Combiner supports.

Example application	Edges	Information flow
Page Rank, Community Detection	Directed	One-way
Hyperlink-Induced Topic Search	Directed	Two-way
Belief Propagation	Undirected	Two-way

Directed graphs and two-way algorithms. Since the goal of V-Combiner is to preserve connectivity in the direction of the update propagation, in this case where updates propagate in both directions, V-Combiner merges subnodes into supernodes in both directions. Recall from Section 5.1 that supernodes are vertices where the sum of in-degree and out-degree is higher than α and lower than β, and that subnodes are vertices where the sum of in-degree and out-degree is less than π, and have at least one edge that connects them to a supernode. We further classified supernodes into in-supernodes and out-supernodes, and subnodes into in-subnodes and out-subnodes. Intuitively, in-supernodes are better connected through their inputs (compared to their outputs), and out-supernodes are better connected through their outputs (compared to their inputs). In these graphs and algorithms, V-Combiner merges in-subnodes into in-supernodes, and out-subnodes into out-supernodes. We call the first kind of merging *forward merging* and the second kind *reverse merging*.

Figure 6 shows an example of reverse merging, where out-subnode ② is merged into out-supernode ③. In this case, edge ② → ① is transformed into ③ → ①, and edge ④ → ② is dropped.



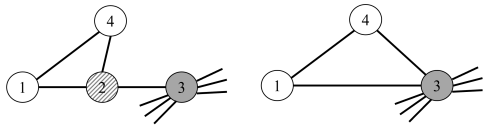
(a) Before merging **(b) After merging**

Figure 6: Merging out-subnode ② into out-supernode ③.

Intuitively, merging in both directions provides better connectivity than merging only in one direction. We perform forward merging of subnodes into supernodes that have better connectivity through their inputs, and reverse merging of subnodes into supernodes that have better connectivity through their outputs.

Undirected graphs. Recall from Section 5.1 that supernodes are vertices whose degree is higher than α and lower than β, and that subnodes are vertices whose degree is less than π, and have at least one edge that connects them to a supernode. In this algorithm, when a subnode is merged into a supernode, all the edges of the subnode are added to the supernode. V-Combiner does not drop any edges because paths do not have a specific direction. Duplicate edges are removed. This operation is also known as vertex-contraction [20].

Figure 7 shows how subnode ② is merged into supernode ③ and its edges are also added to ③. All the edges of the former are added to the latter.



(a) Before merging **(b) After merging**

Figure 7: Merging subnode ② into supernode ③.

5.6 Choosing the Merging Parameters

To effectively reduce a graph, V-Combiner needs to choose a good set of supernodes. The number of supernodes is directly controlled by thresholds α and β. To find good values of α and β, we perform the following experiment. We first rank the vertices based on their in-degree, from lower to higher. We then accumulate the number of edges of the vertices, in order.

Figure 8 shows the resulting Cumulative Density Function (CDF) of edges as a function of the in-degrees of vertices in the Friendster graph [24]. We divide the plot into three regions. In the leftmost region, the curve has a sharp slope. This part accounts for the majority of the edges. These edges connect many vertices with small in-degrees.

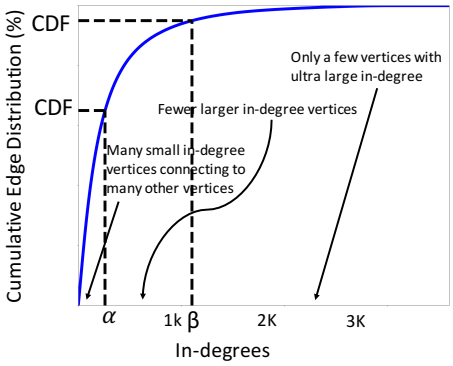


Figure 8: Finding α and β based on the cumulative edge distribution.

In the second region, the curve goes through the knee. Finally, in the third region, the curve flattens up. We argue that supernodes should be chosen from the knee region. This region has many vertices with substantial, but not excessive, in-degrees. Supernodes should not be chosen from the third region. In this region, vertices have very large in-degrees. Increasing their in-degrees further is likely to cause load imbalance.

In Section 8.5, we study different ranges for the knee region, to pick the most profitable one. The boundaries of such a region are the values of α and β.

Once we have picked the values of α and β, we need to pick a value of π. Higher values of π mean that more vertices qualify as subnodes. More subnodes typically translates into lower accuracy, because the elimination of subnodes with large in- and out-degree affects many neighboring vertices. However, more subnodes also translates into more time savings in processing the graph. Therefore, given α and β values, π can be considered as a knob to directly control the trade-off between accuracy and speedup.

6 COMPARISON OF TECHNIQUES

Table 2 qualitatively compares sparsification, k-core, and V-Combiner in terms of their impact on connectivity, length of paths, load balancing, pre- and post-processing overhead, and overall speedup.

Table 2: Qualitative comparison of different techniques. In the table, a check mark means that the technique is doing well; a cross means the opposite.

Technique	Connectivity	Length of Paths	Load Balancing	Pre/Postproc. Overhead	Overall Speedup
Sparsification	✔	✘	✘	✔✔	✔
K-Core	✘	✘	✔	✘	✘
V-Combiner	✔	✔	✔	✔	✔

Connectivity. To attain high accuracy, the reduced graph should strive to maintain connectivity between the remaining vertices. As discussed in Section 4, aggressive k-core easily ends-up causing graph disconnectivity. Sparsification and V-Combiner do not. V-Combiner minimizes disconnectivity by merging a subnode into a hub, and making the in-neighbors of the subnode to become in-neighbors of the hub.

Average Length of Paths. Retaining the length of the paths between vertices in the reduced graph is sometimes important for performance and accuracy. Specifically, if paths are longer, graph algorithms typically take more iterations to converge. Furthermore, in some algorithms, changing the length of the paths causes distortions that result in low accuracy. Generally, all these three algorithms change the average length of the paths. As discussed in Section 4, since sparsification removes edges, it typically increases the length of the paths between vertices. Since k-core and V-Combiner remove vertices, they tend to reduce the average length of the paths. We observe, however, that V-Combiner is the one that changes the average length of the paths the least. Intuitively, this is because it includes two opposite effects: it reduces the length of paths by removing some vertices, but increases the length of paths by dropping some edges.

Load Balancing. To attain high speedups, graph algorithms should keep a good balance of the load between threads. As shown in Algorithm 1, a graph algorithm can be parallelized by assigning a subset of the vertices to each thread to process. The amount of work performed per vertex is proportional to its degree. It is well known that many graphs present a very skewed distribution of the vertices: there are many small-degree vertices and few high-degree ones. We observe that both k-core and V-Combiner improve load balancing by making the distribution less skewed. Specifically, they remove many of the small-degree vertices, either by pruning (in

k-core) or by merging (in V-Combiner). Sparsification largely keeps the same distribution.

Pre- and Post-Processing Overhead. To attain good speedups, it is important that the graph reduction techniques have minimal impact on the pre- and post-processing steps, including the building of the graph. Sparsification has the least pre-processing overhead. k-core has the largest pre-processing overhead: even though we use the state-of-the-art k-core implementation [6], pruning until the remaining vertices have a degree of at least k has substantial overhead. The overhead of the pre- and post-processing in V-Combiner is higher than in sparsification. We show the numbers in Section 8.

Overall Speedup. Based on all the factors considered, the last column of Table 2 outlines the expected relative performance of the techniques.

7 EXPERIMENTAL SETUP

We implement V-Combiner in C++ using OpenMP. We use the Page Rank code from GAP [4], and implement Community Detection [26, 45], Belief Propagation [18], and Hyperlink-Induced Topic Search [22] ourselves.

7.1 Methodology

Machine Specification and Graph Datasets. To evaluate the effectiveness of V-Combiner and the other techniques, we perform experiments on a 2-socket shared-memory system with 44 Intel Xeon Gold 6152 cores and 192GB of memory. We disable the Dynamic Voltage and Frequency Scaling (DVFS) mechanism to minimize run-time variation, and use the *numactl* library with the *interleave all* option to average out the numa effects on the programs. Table 3 shows our graph datasets, which are chosen from several different domains. The Belief Propagation (BP) benchmark uses undirected graphs. Unfortunately, we could not run BP on our largest two graphs, namely FS and TW, as their memory requirements exceed the machine's memory capacity.

Table 3: Graph datasets.

Graph dataset	Vertices	Directed Edges	Undirected Edges
Friendster (FS) [24]	65.6M	1715.7M	-
Twitter (TW)[27]	61.5M	1456.1M	-
Page-Level Domain (PLD) [31]	42.9M	623.1M	582.6M
Arabic-2005 (AR)[5]	22.7M	631.2M	553.9M
Dbpedia (DB) [24]	18.3M	136.5M	126.9M

Evaluation Configurations. We perform end-to-end execution time analysis of our benchmarks. We run each benchmark using both exact and approximate graphs in several different configurations. For each configuration, we measure the time spent in each step of execution by averaging out the results of five runs. The following steps are measured:

- **Prune/Merge.** Identify the vertices/edges to prune or merge.
- **Build.** Construct the approximate (and Delta) graph.
- **Algorithm.** Execute the graph algorithm.
- **Recovery.** Generate the values of the eliminated vertices (V-Combiner and k-core only).

We evaluate the following configurations:

- **Exact.** This is the baseline execution, which is running on the unmodified (original) graph dataset. All the approximate executions are compared to this baseline. The output of this execution is also used as the ground truth for evaluating accuracy.
- **Sparsification.** Sparsification prunes one of many edges from vertices with large degrees. For each edge (u, v), it calculates the probability of keeping it using the formula $\frac{S \times d_{avg}}{min(d_o^u, d_i^v)}$. Here, S is the Sparsification parameter, which is suggested to be in the interval [0.1, 0.9] [15]. Also, d_{avg} is the average degree of the graph, defined as the ratio of the number of edges over the number of vertices. d_o^u and d_i^v are the out-degree and in-degree of the vertices at the two ends of the edge, respectively. The denominator is the minimum of the two degrees. The equation applies to undirected graphs too. Unlike V-Combiner, sparsification does not require a recovery step, since the values for all the vertices are computed on the approximate graph.
  Optimization: Figure 3a shows that the accuracy achieved by sparsification at 0.9 sparsification is 83%. This low accuracy is because a significant number of edges were dropped. To fix this problem, in our evaluation in Section 8, we constrain sparsification using a second parameter (*Percent_E_Remain*), which enforces that a certain fraction of edges remain in the approximate graph.
  Sweeping parameters: We use three sparsification parameter s values (0.9, 0.7, and 0.5), each with 5 different *Percent_E_Remain* parameter values (90%, 80%, 70%, 60% and 50%).
- **K-core.** We follow the state-of-the-art implementation of k-core[6] to identify all the vertices that have to be dropped given a certain k. When a vertex is dropped, all of its in- and out-edges are dropped too.
  Optimization: In our evaluation in Section 8, we add a recovery step to assign values to the removed vertices. This is similar to the algorithm described in Section 5.3 for V-Combiner.
  Sweeping parameters: We try k values equal to 2, 5, 10, 15, 20, 25, 30, 40, and 50. We only consider k-core configurations that have a percentage of remaining edges over 50%.
- **V-Combiner.** V-Combiner uses a combination of pruning and merging, unlike sparsification and k-core, which are pruning-only.
  Sweeping parameters: The (α, β) interval is selected to create a subrange of edge CDF in the curve of Figure 8 within the (65%, 95%) interval. This corresponds to the knee region of the curve. For all benchmarks except HITS, we consider intervals of length 10% each, i.e. (65%, 75%), (75%, 85%) and (85%, 95%). HITS is a two-way algorithm with directed edges and, therefore, there is more merging and edge dropping. For this reason, we experiment with half-sized intervals for HITs, i.e. (65%, 70%),... , (90%, 95%). Given an interval, we sweep the value of π so that the CDF of edges ranges from 5% to CDF(α)%-5%. Finally, we only consider V-Combiner configurations that have a percentage of remaining edges over 50%.

7.2 Accuracy Metrics

We target an accuracy threshold of 90% for all the benchmarks, which is a common threshold used by past work [8, 39].

CD. In CD, each vertex will be assigned a label, indicating the probability of that vertex to belong to a specific community. To measure accuracy, we compute the fraction of initially-unlabeled vertices that end-up being identified to belong to the correct community. We determined the correct community by the exact computation on the original graph.

HITS and PR. We measure the *top-k accuracy* [32], which is the fraction of the vertices in the top k ranks of the exact output that are also in the top k ranks of the approximate output. k is application-dependent, and is given relative to the number of graph's vertices.

Table 4: Algorithm execution time (*time*), and speedup (*sp*), accuracy (*acc*), and percentage of work done (*work*) relative to the baseline for the three techniques. Each row corresponds to one benchmark and graph. In each row, the technique with the highest speedup is shown in bold. An empty row means that the technique could not run the benchmark and graph with an accuracy equal to or higher than the threshold.

Benchmark & Graph	V-Combiner				sparsification				k-core			
	time	sp($\overline{sp}$)	acc	work	time	sp($\overline{sp}$)	acc	work	time	sp($\overline{sp}$)	acc	work
BP-PLD	150.51	1.78(1.87)×	90.1%	53.4%	138.88	**1.93**(1.93)×	90.9%	51.7%	-	-	-	-
BP-AR	93.24	**1.83**(2.06)×	90.8%	48.6%	102.20	1.67(1.89)×	90.3%	53.0%	149.90	1.14(1.13)×	97.0%	88.5%
BP-DB	26.05	1.35(1.63)×	94.9%	61.4%	19.39	**1.81**(1.81)×	95.9%	55.2%	-	-	-	-
CD-FS	74.66	1.21(1.24)×	90.0%	80.5%	52.76	**1.72**(2.08)×	94.8%	48.0%	90.59	1.0(1.0)×	90.1%	99.4%
CD-TW	96.29	**2.3**(2.37)×	90.5%	42.1%	117.63	1.88(2.89)×	91.9%	34.6%	216.92	1.02(1.06)×	98.8%	94.2%
CD-PLD	101.49	1.10(1.09)×	91.3%	91.9%	82.27	**1.38**(1.46)×	90.2%	68.5%	110.77	1.0(1.0)×	90.4%	99.8%
CD-AR	64.82	1.0(1.0)×	99.3%	99.9%	47.16	**1.38**(1.82)×	92.6%	54.9%	62.40	1.04(1.07)×	93.3%	93.7%
CD-DB	31.11	**1.31**(1.34)×	90.0%	74.6%	31.26	1.30(1.72)×	92.6%	58.2%	-	-	-	-
HITS-FS	47.42	1.76(1.72)×	95.8%	58.2%	76.29	1.09(1.13)×	94.6%	88.6%	46.79	**1.78**(1.67)×	99.9%	60.0%
HITS-TW	51.67	**1.21**(1.18)×	90.1%	84.8%	65.34	0.95(1.0)×	95.5%	99.6%	56.13	1.11(1.11)×	99.8%	90.0%
HITS-PLD	11.88	1.65(1.54)×	90.6%	64.9%	-	-	-	-	6.93	**2.82**(2.39)×	97.3%	41.8%
HITS-AR	8.42	1.31(1.31)×	97.2%	76.4%	9.72	1.14(1.2)×	94.9%	83.1%	7.48	**1.48**(1.55)×	98.1%	64.4%
HITS-DB	6.78	1.46(1.25)×	91.1%	80.0%	8.17	1.21(1.08)×	91.2%	92.3%	5.01	**1.97**(1.67)×	90.4%	59.8%
PR-FS	15.80	**2.55**(2.62)×	90.1%	38.2%	21.59	1.87(2.03)×	90.2%	49.2%	33.01	1.22(1.22)×	90.5%	81.8%
PR-TW	20.57	**1.75**(1.74)×	90.4%	57.6%	29.53	1.22(1.26)×	90.2%	79.1%	27.14	1.32(1.35)×	91.8%	73.8%
PR-PLD	8.87	**1.51**(1.35)×	90.3%	73.8%	10.36	1.30(1.43)×	90.4%	70.0%	12.35	1.09(1.09)×	98.3%	91.5%
PR-AR	3.96	1.29(1.27)×	90.0%	78.9%	3.21	**1.59**(2.02)×	90.5%	49.4%	5.12	1.0(1.0)×	98.1%	99.6%
PR-DB	0.92	**1.55**(1.89)×	90.3%	52.8%	1.0	1.42(1.45)×	90.5%	69.2%	-	-	-	-
Average	45.25	**1.55**(1.48)×	91.8%	67.6%	48.04	1.46(1.54)×	92.2%	64.9%	59.32	1.36(1.23)×	95.3%	81.3%

BP. Based on past work [11, 16, 17, 19, 44], we divide BP use cases into two main categories: (i) classification of vertices based on the class they belong to, and (ii) ranking of the vertices based on information such as user trustworthiness and influence. For the classification scenario, since we are only able to run the algorithm for belief vectors of size 2 (due to not enough memory), we observe high accuracy in most cases. Therefore, we choose the ranking scenario, where we can capture the accuracy better. We use the top-k accuracy metric.

8 EVALUATION

We first compare V-Combiner's performance-accuracy trade-offs to sparsification's and k-core's, both for algorithm-time (Section 8.1) and for end-to-end time (Section 8.2). We then present statistics on the approximate graphs (Sections 8.3 and 8.4), and an analysis of the best pruning or merging parameters (Section 8.5).

To compare V-Combiner, sparsification, and k-core, we pick their best parameter configurations from Section 7.1 that, for each individual benchmark and input graph, deliver the highest end-to-end speedup over the baseline. The accuracy is required to be equal to or higher than the 90% threshold. We call these configurations the *best end-to-end* configurations.

8.1 Algorithm Performance and Accuracy

Best Design Points. In this section, to understand the trade-offs between the different algorithms, we take the best end-to-end configurations but recompute the speedups without considering the *pre-processing time*. The "pruning" or "merging" part of the pre-processing time is highly variable across different techniques and overshadows the savings obtained by reducing algorithm time. However, we do consider the post-processing time since it contributes to the final accuracy in both V-Combiner and k-core. In Section 8.2, we will analyze the speedups of the best end-to-end configurations with all the times included.

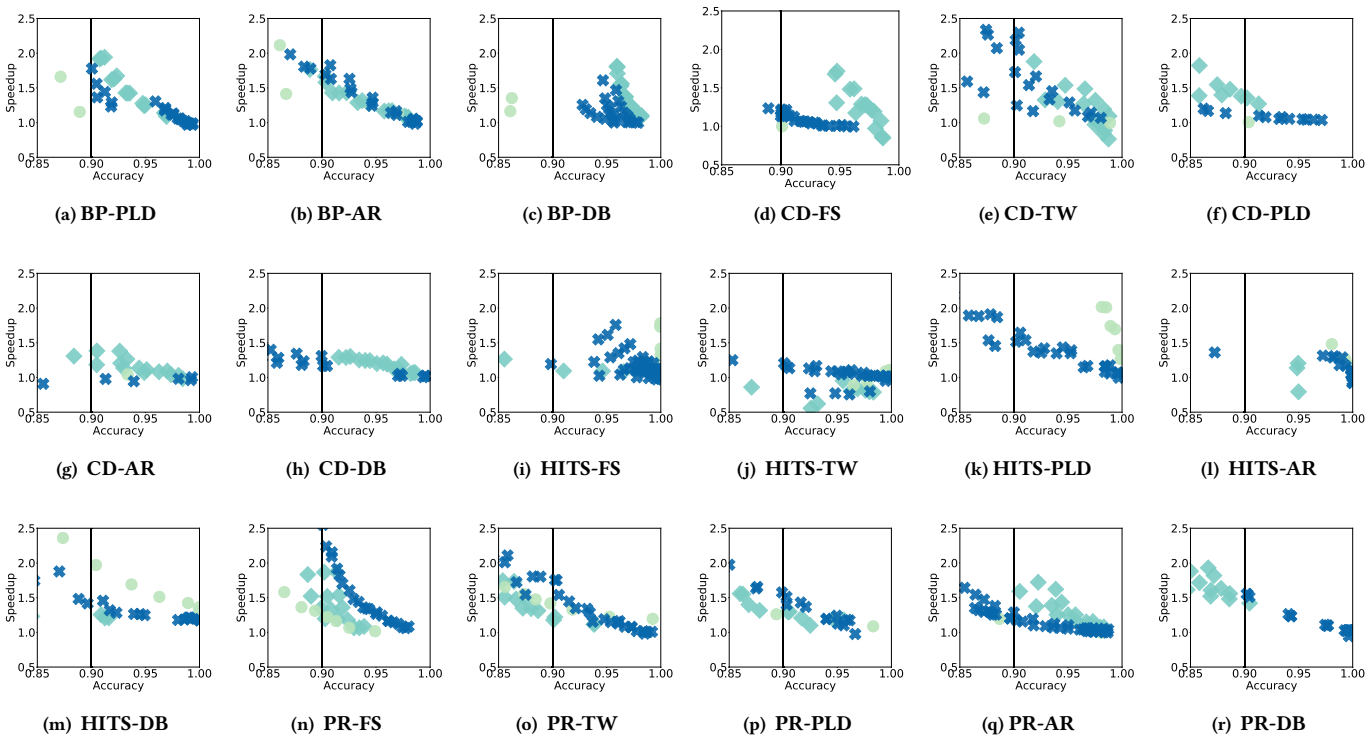
Table 4 shows the resulting algorithm execution time (*time*), and the speedup (*sp*), accuracy (*acc*), and percentage of work done (*work*) relative to the baseline for the three techniques. The amount of work done is proportional to the number of edges multiplied by the number of iterations. We also report the "expected" speedup ($\overline{sp}$) next to each speedup number, which is computed as $\frac{100}{work(\%)}$. It largely indicates the speedup that would be achieved without considering any load imbalance effects in the approximate graph.

In the table, each row corresponds to one benchmark and graph. In each row, the technique with the highest speedup is in bold. An empty row means that the technique could not run the benchmark and graph with an accuracy equal to or higher than the threshold.

Comparing V-Combiner to sparsification. V-Combiner attains an average speedup of 1.55×, while sparsification attains 1.46×. Surprisingly, this happens regardless of the fact that sparsification performs less work on average — i.e. 64.9% compared to 67.6%. The reason is the better load balancing of the work in V-Combiner due to the merging of small vertices. We observe that, on average in V-Combiner, the speedup is 4.7% higher than the expected speedup, while it is 5.2% lower than the expected speedup in sparsification. Additionally, V-Combiner satisfies the accuracy threshold across all the benchmarks, while sparsification fails to meet the accuracy threshold in HITS-PLD.

V-Combiner ensures high speedups by dropping both vertices and edges, as compared to sparsification which only drops edges. We can observe the higher speedups in all of PR and HITS experiments (except PR-AR) and some of BP and CD experiments, i.e. BP-AR, CD-TW, and CD-DB.

Comparing V-Combiner to k-core. k-core obtains a significantly lower average speedup than V-Combiner, i.e. 1.36×. This is because k-core performs more work than V-Combiner (81.3%) to reach an accuracy of above the 90% threshold. Furthermore, k-core fails to satisfy the accuracy threshold in four experiments. Another interesting observation is that both V-Combiner and k-core achieve average higher speedups than their expected speedups. This is because both



(a) BP-PLD (b) BP-AR (c) BP-DB (d) CD-FS (e) CD-TW (f) CD-PLD

(g) CD-AR (h) CD-DB (i) HITS-FS (j) HITS-TW (k) HITS-PLD (l) HITS-AR

(m) HITS-DB (n) PR-FS (o) PR-TW (p) PR-PLD (q) PR-AR (r) PR-DB

Figure 9: Pairs of (speedup, accuracy) for V-Combiner (✖), k-core (●) and sparsification (◆). Speedups only include algorithm execution time. The vertical line shows the 0.9 accuracy threshold.

techniques perform small-degree vertex pruning/merging, which helps load balancing the work better than the original graph.

Compared to k-core, V-Combiner relies on the high connectivity of its approximate graph and recovery algorithm to achieve higher speedup-accuracy operating points. The recovery algorithm is more effective in V-Combiner than in k-core. k-core's recovery is performed using other removed vertices with their initial values, while V-Combiner's recovery is performed using vertices that have been already involved in the computation. Except in HITS-FS, HITS-PLD, HITS-AR, and HITS-DB, V-Combiner achieves comparable or higher speedups than k-core. For these graphs, k-core performs less work with higher accuracy.

Overall, V-Combiner attains the highest average speedup in 8 out of 18 benchmarks. The reasons are better load balancing and the preservation of the average length of paths (relative to sparsification), and performing less work at high accuracy due to better connectivity (relative to k-core).

Speedup-Accuracy Trends. To further compare V-Combiner, sparsification, and k-core, Figure 9 shows pairs of (speedup, accuracy) for the techniques. In the charts, the X-axis shows accuracy from 0.85 (low) and 1.00 (high), and the Y-axis shows speedup over the baseline algorithm. We obtain these points by sweeping the parameters of each technique according to Section 7.1. As before, we include the post-processing time but not the pre-processing time, since we focus on the algorithm speedup only. Note that some of the data points in Figure 9 show higher speedups than in Table 4. This is because Table 4 only shows the best *end-to-end* configurations.

Comparing V-Combiner to sparsification. In the figure, the closer the (speedup, accuracy) pairs are to the top right of each plot, the better the trade-off is. For all HITS and PR experiments except PR-AR, V-Combiner achieves better results than sparsification. Typically in PR, sparsification is more likely to achieve better trade-offs on more "skewed" graphs — i.e. graphs that have many small-degree vertices and few extraordinarily-large degree vertices. In such input graphs, there is not much room to merge subnodes without creating disconnectivity, since those large vertices are not picked as supernodes (according to Algorithm 2). However, in practice, most real-world graphs are less skewed and thus more suitable for V-Combiner. For BP, both schemes have similar results. Finally, for CD, sparsification generally gets better results, especially for highly dense graphs such as CD-FS. In contrast, V-Combiner achieves better results in sparser graphs such as CD-TW.

Comparing V-Combiner to k-core. Figure 9 shows that in most experiments, V-Combiner exhibits better performance-accuracy trade-offs than k-core. The exceptions are HITS-FS, HITS-PLD, HITS-AR, and HITS-DB, for the reason indicated before. However, we will see in Section 8.2 that the end-to-end time of k-core is significantly affected by the pre-processing overhead.

8.2 End-to-End Analysis

Figure 10 shows the total execution time of the best end-to-end configurations for the different benchmark-graph pairs and different techniques. For each benchmark-graph pair, we show, from left to right, bars for the Exact, V-Combiner, sparsification, and k-core techniques, all normalized to Exact. The numbers on top of the bars are the end-to-end speedups over Exact. The missing bars (sparsification in HITS-PLD, and k-core in BP-PLD, BP-DB, CD-BD, and PR-BD) are for configurations that failed to reach the threshold accuracy. The accuracy of each experiment was shown in Table 4.

The bars are broken down into the different components of the execution time as shown in the steps of Figure 4: (i) pruning or merging, (ii) building the graph, (iii) executing the graph algorithm,

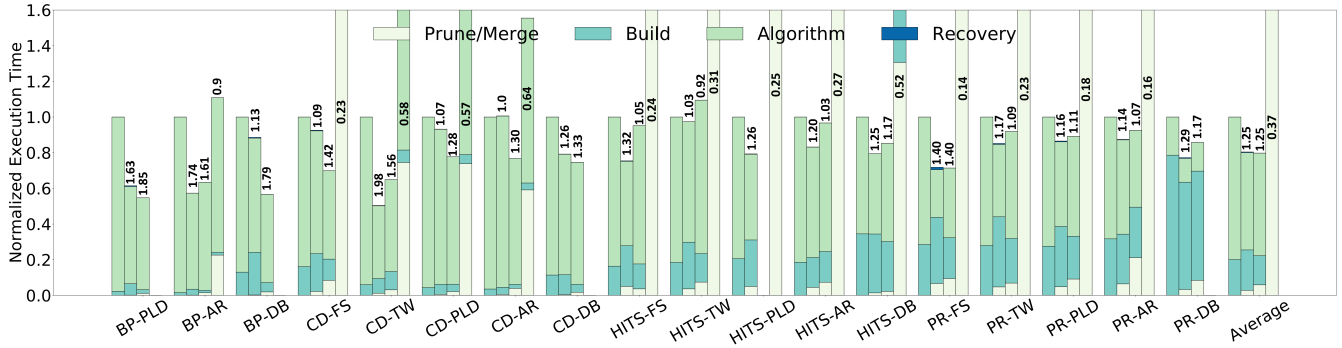


Figure 10: Total execution time of the best end-to-end configurations for the different benchmark-graph pairs and different techniques. For each benchmark-graph pair, we show, from left to right, bars for the Exact, V-Combiner, sparsification, and k-core techniques, all normalized to Exact. The numbers on top of the bars are the end-to-end speedups over Exact. The missing bars (one in sparsification and four in k-core) are for configurations that failed to reach the threshold accuracy.

and (iv) post-processing or recovering. From the figure, we see that the prune/merge step is cheap in V-Combiner and sparsification. However, it is very expensive in most of the k-core experiments: even though we use the state-of-the-art k-core implementation [6], pruning until the remaining vertices have a degree of at least k has substantial overhead.

The Build step takes, on average, 20%, 22.6%, 16.4%, and (not shown) 19.7% of the *total Exact time*, in Exact, V-Combiner, sparsification, and k-core, respectively. The Build time in V-Combiner is higher than in sparsification because, in this step, V-Combiner performs the actual vertex merging and generates the Delta graph. (Recall that, in the merge step, V-Combiner executes Algorithm 2, which computes the new vertex degrees). However, the algorithm time is often shorter in V-Combiner than in sparsification, as we saw in Table 4. The recovery overheads are negligible.

Overall, we see that V-Combiner obtains an average end-to-end speedup of 1.25× over the baseline across the 18 benchmarks. It does so with an average accuracy of 91.8% (Table 4). Sparsification typically has a slower algorithm but a shorter pre-processing time. The resulting average end-to-end speedup of sparsification over baseline is like V-Combiner in Figure 10. However, one of the benchmark-graph pairs (HITS-PLD) does not reach the required accuracy in sparsification. Specifically, it can be shown that HITS-PLD only reaches 71% accuracy in sparsification. Such experiment is not included in the average sparsification bar of Figure 10. As a result, V-Combiner is preferable over sparsification. Finally, K-core exhibits a substantial slowdown over baseline on average due to its large pruning overhead.

8.3 Analysis of the Connectivity

Table 5 shows pruning or merging statistics for different techniques with the configurations of Table 4. For V-Combiner, we show the breakdown of the vertices in the original graph into supernodes, subnodes, and regular vertices. On average, the fraction of supernodes, subnodes, and regular vertices is 0.1%, 18.5%, and 81.4%, respectively. The percentage of vertices that remain in the approximate graph is equal to the percentage of supernodes plus regular vertices. On average, it is 81.5%. The percentage of edges remaining in the V-Combiner approximate graph is 71.4% on average.

For sparsification, the table shows the percentage of remaining edges (since no vertex is removed). On average, such number is 66.0%. Finally, for k-core, the table shows the percentages of remaining vertices and edges. On average, they are 54.4% and 87.6%.

Table 5: Pruning or merging statistics.

Benchmark & Graph	V-Combiner super	V-Combiner sub	V-Combiner regular	V-Combiner edges	sparsif. edges	k-core vertices	k-core edges
BP-PLD	0.0%	21.6%	78.3%	53.4%	51.7%	-	-
BP-AR	0.1%	37.3%	62.6%	50.6%	53.1%	90.2%	99.6%
BP-DB	0.4%	20.9%	78.7%	61.4%	55.2%	-	-
CD-FS	0.4%	27.6%	72.0%	83.9%	50.0%	84.2%	99.4%
CD-TW	0.0%	7.5%	92.5%	59.1%	50.0%	65.5%	98.8%
CD-PLD	0.0%	4.6%	95.4%	79.0%	68.0%	81.1%	99.2%
CD-AR	0.0%	0.6%	99.3%	99.8%	54.4%	92.7%	99.7%
CD-DB	0.0%	3.7%	96.3%	90.4%	54.3%	-	-
HITS-FS	0.7%	30.6%	68.7%	54.4%	80.0%	24.4%	76.5%
HITS-TW	0.1%	10.4%	89.5%	78.1%	84.8%	22.9%	84.1%
HITS-PLD	0.0%	25.4%	74.6%	75.3%	-	8.1%	55.8%
HITS-AR	0.1%	15.0%	84.9%	86.1%	89.9%	31.7%	75.8%
HITS-DB	0.0%	8.9%	91.1%	87.6%	90.1%	11.8%	65.4%
PR-FS	0.1%	36.6%	62.7%	51.4%	50.0%	41.1%	91.7%
PR-TW	0.1%	21.3%	78.6%	63.0%	80.2%	38.1%	94.1%
PR-PLD	0.0%	19.2%	80.8%	72.3%	70.0%	77.4%	98.9%
PR-AR	0.0%	20.4%	79.6%	82.6%	50.4%	90.2%	99.6%
PR-DB	0.2%	15.6%	84.2%	57.4%	90.1%	-	-
Average	0.1%	18.5%	81.4%	71.4%	66.0%	54.4%	87.6%

V-Combiner keeps more remaining vertices in the approximate graph than k-core. This explains why V-Combiner provides better connectivity than k-core. We also see that, on average, the techniques generate approximate graphs with 65%-90% of the edges, which provide the most profitable performance-accuracy trade-off. Finally, sparsification has the lowest percentage of edges remaining. Because it does not remove vertices, it can tolerate dropping so many edges while still preserving good connectivity.

8.4 Analysis of the Average Length of Paths

Table 6 compares the average length of the paths in each benchmark-graph pair in the original and approximate graphs. To compute the average path length in each original graph, we select and run 10,000 shortest path queries and average out all the shortest path distances. Next, we use the same queries to measure the average length of the paths in the approximate graphs that the different techniques generated using the configurations of Table 4. Finally, we compute the error as the difference between the values in the original and approximate graphs, as a percentage.

V-Combiner preserves the average length of paths much more than sparsification or k-core. On average, the error in average path length in V-Combiner is only 3.8%, while it is 30.1% and 37.6% in

Table 6: Average length of the paths in different graphs. The percentage numbers show the percentage of error relative to the original graph.

Benchmark & Graph	original	V-Combiner	sparsification	k-core
BP-PLD	3.73	3.25(12.9%)	4.68(25.5%)	-
BP-AR	7.18	6.18(13.9%)	8.43(17.4%)	4.66(35.1%)
BP-DB	3.81	3.39(11.0%)	5.33(39.9%)	-
CD-FS	5.82	6.13(5.3%)	8.29(42.4%)	4.28(26.5%)
CD-TW	4.21	4.22(0.2%)	5.30(25.9%)	2.19(48.0%)
CD-PLD	4.34	4.36(0.5%)	5.54(27.6%)	0.58(86.6%)
CD-AR	17.64	17.62(0.1%)	21.10(19.6%)	4.38(75.2%)
CD-DB	5.20	5.06(2.7%)	8.63(66.0%)	-
HITS-FS	5.82	5.95(2.2%)	7.73(32.8%)	4.69(19.4%)
HITS-TW	4.21	4.29(1.9%)	4.77(13.3%)	3.15(25.2%)
HITS-PLD	4.34	4.35(0.2%)	-	1.76(59.4%)
HITS-AR	17.64	17.86(1.2%)	18.47(4.7%)	16.86(4.4%)
HITS-DB	5.20	5.10(1.9%)	6.65(27.9%)	4.22(18.9%)
PR-FS	5.82	6.06(4.1%)	8.27(42.1%)	5.09(12.5%)
PR-TW	4.21	4.39(4.3%)	5.30(25.9%)	3.55(15.7%)
PR-PLD	4.34	4.36(0.5%)	5.55(27.9%)	4.23(2.5%)
PR-AR	17.64	18.02(2.2%)	20.80(17.9%)	0.5(97.2%)
PR-DB	5.20	5.05(2.9%)	8.60(55.0%)	-
Mean error	-	3.8%	30.1%	37.6%

sparsification and k-core, respectively. Generally, k-core reduces the average length of the paths, while sparsification increases it. The reason is that k-core prunes vertices and therefore reduces the number of hops to go from one vertex to another. Moreover, k-core often disconnects parts of the graph, causing longer paths to disappear, and the average path length to decrease. In contrast, sparsification increases the number of hops between vertices because it reduces the connections between the vertices by removing edges.

8.5 Analysis of Pruning/Merging Parameters

Table 7 shows the pruning or merging parameters that we use to generate the best end-to-end configurations for each benchmark-graph pair and technique. In V-Combiner, the parameters are the supernode thresholds α and β, and the subnode threshold π. The table shows these parameters as the corresponding values in the CDF of number of edges (Figure 8). We observe that, for CD and BP, the best (α, β) values are mostly (85%, 95%). For HITS, the best values are mostly (65%, 70%), and for PR, they are mostly (75%, 85%). The π parameter shows more variation, as it generally ranges between 60% and 20%. In contrast, for sparsification and k-core, we do not see a clear trend on how to set the s and k parameters; hence a full sweep of parameters is required.

Table 7: Best parameters of the different techniques.

Benchmark & Graph	V-Combiner α	V-Combiner β	V-Combiner π	sparsification s param	k-core k param
BP-PLD	85%	95%	80%	0.7	-
BP-AR	85%	95%	60%	0.7	2
BP-DB	75%	85%	70%	0.9	-
CD-FS	85%	95%	70%	0.7	2
CD-TW	85%	95%	75%	0.5	5
CD-PLD	85%	95%	20%	0.9	2
CD-AR	85%	95%	5%	0.9	2
CD-DB	85%	95%	20%	0.9	-
HITS-FS	65%	70%	60%	0.7	40
HITS-TW	70%	75%	50%	0.9	25
HITS-PLD	65%	70%	30%	-	40
HITS-AR	65%	70%	20%	0.7	30
HITS-DB	65%	70%	30%	0.7	25
PR-FS	75%	85%	60%	0.7	15
PR-TW	75%	85%	60%	0.5	10
PR-PLD	75%	85%	40%	0.9	2
PR-AR	85%	95%	15%	0.5	2
PR-DB	75%	85%	40%	0.9	-

9 RELATED WORK

Previous research focused on algorithm-specific approximations such as FrogWild [32] for Page Rank and approximate K-level asynchronous Breadth-First Search [9]. While these approximations are effective for one algorithm, they often lack the generality to be applied to a broad range of algorithms. More general approaches include graph summarization techniques [1, 3, 7, 21, 28, 30, 37], and other graph processing approximate frameworks [25, 33, 38], or general-purpose frameworks that can be applied to graph processing domains too [12, 40]. Compared to the general-purpose frameworks, V-Combiner is deterministic and provides application transparency.

Graph summarization techniques vary widely from algorithm-specific such as sketching algorithms [1, 7, 30, 37] to more general-purpose such as sparsification [3, 15] and k-core [21]. The original proposal of sparsification [3] provides bounds for the Page Rank algorithm. However, it is not applicable in practice. First, it requires the graph input to be undirected, while most of the real-world graphs are directed. Second, it requires the knowledge of the graph eigenvalues, which will take much higher time to compute than the actual Page Rank values. A practical implementation of sparsification [15] replaced the eigenvalues with a tunable sparsification parameter and average degree of the graph that provides end-to-end performance gains, but with no accuracy guarantees.

The idea behind sketching algorithms is to summarize the graph information tailored to a specific application into a data structure that can be queried later to return a fast approximate answer after simple computations. Unfortunately, while sketching techniques provide high accuracy, their end-to-end performance is much worse than the exact baseline due to their high pre-processing overheads. V-Combiner has lower overhead and hence provides higher end-to-end performance at high accuracy. In contrast to some popular sketching algorithms that are tailored to specific graph algorithms, V-Combiner can be applied to a wide variety of graph algorithms, without modifying their code.

10 CONCLUSION

Fast graph processing has an important role in many applications where a small level of inaccuracy is acceptable. To speed-up iterative graph processing algorithms, we propose to merge certain graph vertices into hub vertices next to them — i.e. vertices with many connections. Our novel scheme, *V-Combiner*, systematically and deterministically constructs an approximate graph using pruning and merging. It also includes an inexpensive correction step after the graph algorithm executes, to recover the contribution of the pruned vertices. The result is faster execution at acceptable accuracy levels.

We evaluated V-Combiner on a 44-core shared-memory platform. On average across 4 applications and 5 graph datasets, V-Combiner attained an end-to-end speed-up of 1.25× over the exact baseline system, with an accuracy of 91.8%. We also showed that V-Combiner provides an overall better performance-accuracy trade-off than the sparsification and k-core techniques.

ACKNOWLEDGMENTS

This research was funded in part by the National Science Foundation under grants CCF-1629431 and CCF-1703637.

REFERENCES

[1] Takuya Akiba and Yosuke Yano. 2016. Compact and scalable graph neighborhood sketching. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 685–694.

[2] Hongji Bao and Edward Y Chang. 2010. AdHeat: An influence-based diffusion model for propagating hints to match ads. In *Proceedings of the 19th international conference on World wide web*. ACM, 71–80.

[3] Joshua Batson, Daniel A Spielman, Nikhil Srivastava, and Shang-Hua Teng. 2013. Spectral sparsification of graphs: theory and algorithms. *Commun. ACM* 56, 8 (2013), 87–94.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[5] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[6] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 293–304.

[7] Dorit Dor, Shay Halperin, and Uri Zwick. 2000. All-pairs almost shortest paths. *SIAM J. Comput.* 29, 5 (2000), 1740–1759.

[8] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 449–460.

[9] Adam Fidel, Francisco Coral Sabido, Colton Riedel, Nancy M Amato, and Lawrence Rauchwerger. 2016. Fast approximate distance queries in unweighted graphs using bounded asynchrony. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 40–54.

[10] Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, Takeshi Mishima, and Makoto Onizuka. 2013. Fast and exact top-k algorithm for pagerank. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*.

[11] Wolfgang Gatterbauer. 2017. The linearization of belief propagation on pairwise markov random fields. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[12] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 383–397.

[13] Priya Govindan, Chenghong Wang, Chumeng Xu, Hongyu Duan, and Sucheta Soundarajan. 2017. The k-peak decomposition: Mapping the global structure of graphs. In *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1441–1450.

[14] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 745–761.

[15] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. 2018. Bridging the GAP: towards approximate graph analytics. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, 10.

[16] Min-Hee Jang, Christos Faloutsos, Sang-Wook Kim, U Kang, and Jiwoon Ha. 2016. Pin-trust: Fast trust propagation exploiting positive, implicit, and negative information. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 629–638.

[17] Kyomin Jung, Wooram Heo, and Wei Chen. 2012. Irie: Scalable and robust influence maximization in social networks. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, 918–923.

[18] U Kang, Duen Horng Chau, and Christos Faloutsos. 2011. Mining large graphs: Algorithms, inference, and discoveries. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 243–254.

[19] U Kang, Duen Horng, et al. 2010. Inference of beliefs on billion-scale graphs. *Workshop on Large-scale Data Mining: Theory and Applications* (2010).

[20] David R Karger and Clifford Stein. 1996. A new approach to the minimum cut problem. *Journal of the ACM (JACM)* 43, 4 (1996), 601–640.

[21] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[22] Jon M Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)* 46, 5 (1999), 604–632.

[23] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2017. GPU-Accelerated Graph Clustering via Parallel Label Propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 567–576.

[24] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.

[25] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 245–257.

[26] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

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

[28] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 62.

[29] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*. 631–643.

[30] Andrew McGregor. 2014. Graph stream algorithms: a survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[31] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2019. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/.

[32] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. 2015. FrogWild!: Fast PageRank approximations on graph engines. *Proceedings of the VLDB Endowment* 8, 8 (2015), 874–885.

[33] Hamza Omar, Masab Ahmad, and Omer Khan. 2017. GraphTuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 201–208.

[34] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[35] Ali Pinar, Tamara G Kolda, and Changbin Peng. 2014. *Accelerating Community Detection by using k-core subgraphs.* Technical Report. Sandia National Lab.(SNL-CA), Livermore, CA (United States).

[36] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 324–334.

[37] Tamás Sarlós, Adrás A Benczúr, Károly Csalogány, Dániel Fogaras, and Balázs Rácz. 2006. To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *Proceedings of the 15th international conference on World Wide Web*. ACM, 297–306.

[38] Zechao Shang and Jeffrey Xu Yu. 2014. Auto-approximation of graph computing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1833–1844.

[39] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 124–134.

[40] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. 2016. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 607–621.

[41] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 1785–1794.

[42] Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 507–516.

[43] Biao Xiang, Qi Liu, Enhong Chen, Hui Xiong, Yi Zheng, and Yu Yang. 2013. Pagerank with priors: An influence propagation perspective. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[44] Jaemin Yoo, Saehan Jo, and U Kang. 2017. Supervised Belief Propagation: Scalable Supervised Inference on Attributed Networks. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 595–604.

[45] Xiaojin Zhu and Zoubin Ghahramani. 2002. *Learning from labeled and unlabeled data with label propagation.* Technical Report. Carnegie Mellon University.