Parallel Index-Based Structural Graph Clustering and Its Approximation*

Tom Tseng
MIT CSAIL
tomtseng@csail.mit.edu

Laxman Dhulipala
MIT CSAIL
laxman@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract
SCAN (Structural Clustering Algorithm for Networks) is a well-studied, widely used graph clustering algorithm. For large graphs, however, sequential SCAN variants are prohibitively slow, and parallel SCAN variants do not effectively share work among queries with different SCAN parameter settings. Since users of SCAN often explore many parameter settings to find good clusterings, it is worthwhile to precompute an index that speeds up queries.

This paper presents a practical and provably efficient parallel index-based SCAN algorithm based on GS*-Index, a recent sequential algorithm. Our parallel algorithm improves upon the asymptotic work of the sequential algorithm by using integer sorting. It is also highly parallel; it achieves logarithmic span for both index construction and clustering queries. Furthermore, we apply locality-sensitive hashing (LSH) to design a novel approximate SCAN algorithm and prove guarantees for its clustering quality.

We present an experimental evaluation of our parallel algorithms on large real-world graphs. On a 48-core machine with two-way hyper-threading, our parallel index construction achieves 50–151× speedup over the construction of GS*-Index. In fact, even on a single thread, our index construction algorithm is faster than GS*-Index. Our parallel index query implementation achieves 5–32× speedup over GS*-Index queries across a range of SCAN parameter values, and our implementation is always faster than ppSCAN, a state-of-the-art parallel SCAN algorithm. Moreover, our experiments show that applying LSH results in much faster index construction on denser graphs without large sacrifices in clustering quality.

1 Introduction
In data mining and unsupervised learning, clustering is a fundamental technique that organizes data into meaningful groups. Because much real-world data can be represented as graphs, there is significant practical and theoretical interest in graph clustering, in which the goal is to partition the vertices of a graph into clusters such that “similar” vertices fall into the same cluster [3–5, 24, 50, 56, 62, 72]. In particular, a good clustering usually has many edges that fall within clusters and few edges that connect different clusters. Graph clustering is a popular problem with a wide range of applications, including social and biological network analysis [33], load balancing in distributed systems [2], image segmentation [66], natural language processing [7], and recommendation systems [6].

One well-known approach to graph clustering is structural clustering, which Xu et al. first introduced via the Structural Clustering Algorithm for Networks (SCAN) [71]. Structural clustering exploits the idea that vertices whose neighbor sets resemble each other are “similar,” a type of homophily that is often satisfied in practice. The approach is unique in that it also finds hub vertices that connect different clusters as well as outlier vertices that lack strong ties to any cluster. Researchers have used SCAN to find meaningful clusters in biological data [28, 44, 46, 47] and web data [43, 51–53, 57, 58].

SCAN as Xu et al. originally described it suffers from two issues: (1) the costliness of sequentially computing similarities among all adjacent vertices, and (2) the costliness of tuning the parameters of the algorithm to achieve good clustering quality. Many researchers have developed variants of SCAN to address these issues. To alleviate issue (1), some variants exploit parallelism [18, 19, 45, 64, 65, 76, 77] or introduce algorithmic optimizations like pruning unnecessary similarity computations [16, 18, 59]. To alleviate issue (2), some variants precompute an index from which computing the clusterings for different parameter values is fast [13, 37, 68]. To be efficient on large graphs, SCAN-based algorithms should address both issues, which existing algorithms fail to do.

This paper addresses the aforementioned issues by presenting a new parallel index-based SCAN algorithm based on the sequential GS*-Index SCAN algorithm [68]. Our algorithm achieves the same work bounds as GS*-Index and is highly parallel, achieving logarithmic span (parallel time) with high probability (w.h.p.). The key ingredients to achieve our strong time bounds are the careful use of doubling search as well as parallel algorithms for graph connectivity and hash tables. We also show how using integer sort improves the index construction work bound compared to GS*-Index’s bound.

To further improve performance, we present new approximate algorithms that use locality-sensitive hashing (LSH) to speed up similarity computation. We provide a non-trivial theoretical analysis of the accuracy of our algorithms. Our experiments show that using LSH significantly speeds up our index construction on dense graphs without sacrificing clustering quality. We summarize the asymptotic running time bounds for index construction in table 1.

We present highly optimized implementations of our algorithms. The most important optimizations are a merge-based parallel triangle counting algorithm described by Shun and Tangwongsan [63] to compute similarities; concurrent union-find to compute connectivity for queries; and, for our approximate algorithms, a heuristic to avoid using LSH on low-degree vertices that would not benefit from approximation. In our experiments, our index construction algorithm achieves 50–151× speedup over the construction of GS*-Index for several large real-world graphs on a machine with 48 cores and two-way hyper-threading. In fact, our index construction algorithm is faster than GS*-Index even when we run our algorithm on

*The work of an algorithm is the number of operations it performs. The span (parallel time) of an algorithm is the length of its longest sequential dependence. We use with high probability (w.h.p.) to describe events that occur with probability at least $1 − 1/n^c$ where $n$ is the input size and $c$ is some positive real number.
This section provides background definitions and concepts that sub-

Table 1: Approximate index

| Description                              | Work                                    | Span |
|------------------------------------------|-----------------------------------------|------|
| Exact index, weighted graph              | $O((a + \log n) m)$ w.h.p.              | $O(\log n)$ w.h.p. |
| Exact index, unweighted graph            | $O((a + \log \log n) m)$ w.h.p.        | $O(\log n)$ w.h.p. |
| Approximate index                         | $O(k + \log \log n) m)$ w.h.p.        | $O(\log n)$ w.h.p. |

The definition of cosine similarity for sets with elements from $U$ follows by representing sets as vectors in $\mathbb{R}^U$ (namely, as a bit vector for unweighted sets and as a vector of weights for weighted sets).

2.1.2 Locality-sensitive hashing Suppose that there is a collection of large sets with elements from a finite universe $U$. Locality-sensitive hashing (LSH) is a technique to quickly approximate the similarity between pairs of these sets. The idea is to devise a hash function family that maps similar sets to similar, smaller sketches. We estimate similarities by precomputing all sketches and operating on the sketches rather than on the large original sets.

A well-known LSH scheme for estimating Jaccard similarity, for instance, is MinHash [15]. MinHash works by drawing a uniformly random permutation $\pi$ on $U$ and considering the sketch of a non-empty set $S$ to be $\min_{x \in S} \pi(x)$. For any two non-empty sets $A$ and $B$, the probability that the sketches of $A$ and $B$ are equal is exactly $JaccardSim(A, B)$. To increase the precision at the cost of extra work, we fix a number of samples $k \in \mathbb{N}$ and perform this process $k$ times independently to get $k$-length sketches. The proportion of matching coordinates between two sketches is an estimate of the Jaccard similarity between the two corresponding sets. There are variants of MinHash that strive to be more computationally efficient such as $k$-partition MinHash [41]. There are also variants for weighted Jaccard similarity [70]. These weighted variants are more complicated and less practical, which is why we do not use weighted Jaccard similarity in this work.

SimHash [17] is a well-known LSH scheme for estimating the angle between two vectors. The idea behind SimHash is to consider drawing a vector $v$ in $\mathbb{R}^U$ with uniformly random direction by drawing each coordinate independently from the standard normal distribution. We take the sketch of a vector $u$ to be $\text{sign}(u \cdot v)$. For a pair of non-zero vectors $a$ and $b$ with angle $\theta \in [0, \pi]$ in radians between them, the probability that the sketches of $a$ and $b$ differ is exactly $\theta / \pi$; because $v$ has uniformly random direction, the orthogonal hyperplane to $v$ separates $a$ and $b$ with probability $\theta / \pi$, which exactly corresponds to the event that $\text{sign}(a \cdot v) \neq \text{sign}(b \cdot v)$. Like with MinHash, to tune the precision, we repeat this process $k \in \mathbb{N}$ times to get $k$-length sketches. The number of differing entries between the sketches of $a$ and $b$ multiplied by $\pi / k$ is an estimate $\hat{\theta} \sim \text{Binomial}(k, \theta / \pi) \cdot \pi / k$, which in turn provides an estimate $\cos(\hat{\theta})$ for $\cos(\theta) = \text{CosineSim}(a, b)$.

2.2 Graphs and graph notation

We denote an unweighted, undirected graph $G$ by $G = (V, E)$ where $V$ is the set of vertices and $E \subseteq \{(u, v) : u, v \in V\}$ is the set of edges. We denote a weighted graph $G = (V, \mathbb{E}, \omega)$, where the weight function $\omega : E \rightarrow \mathbb{R}$ maps edges to weights. Following common convention, we use $n$ to denote the number of vertices $|V|$ and $m$ to denote the number of edges $|E|$. The neighborhood $N(v)$ of a vertex $v$ is the set of vertices connected to $v$ by an edge. The closed neighborhood of $v$ is $\overline{N}(v) = N(v) \cup \{v\}$. The degree of a vertex $v$ is the size of its neighborhood, though when the exact value is important, we write either $|N(v)|$ or $|\overline{N}(v)|$ to avoid ambiguity.

For directed graphs, each edge in $E$ becomes an ordered pair rather than an unordered pair. The out-neighborhood of a vertex $v$ is the set of all vertices $u$ such that $(v, u) \in E$. 

\[^{2}\text{Code: https://github.com/PatAlg/gbbto/tree/master/benchmarks/SCAN/IndexBased}\]
The arboricity $a$ of an undirected graph $G$ is the minimum number of spanning forests that covers all edges of the graph. The arboricity is bounded below by $\lceil m/(n-1) \rceil$ since each spanning forest covers at most $n-1$ edges and is bounded above by $O(\sqrt{m+n})$. A triangle is a triplet of edges $\{u, v\}, \{v, x\}, \{x, u\}$ between distinct vertices $u, v, x$ in $V$. There are triangle counting algorithms that find all triangles in a graph in $O(nm)$ time [20].

We represent graphs as adjacency lists, in which each vertex has a list of its neighbors. We only consider simple graphs, graphs without multiple edges between any particular pair of vertices and without self-loop edges. We index vertices with the integers in the range $[1, n]$.

### 2.3 Parallelism

#### 2.3.1 Parallel programming model

We design our algorithms for multicore shared-memory machines. Readily available shared-memory machines are able to operate on the largest publicly-available real-world graphs, which have hundreds of billions of edges [25]. Shared-memory systems are fast due to low communication costs and are easier to program for than distributed systems are.

We use a fork-join programming model with arbitrary forking; a process can “fork” into an arbitrary number of parallel processes in unit time and can “join” to synchronize among forked processes. Most notably, a fork and a join suffice to implement a parallel for-loop. We further assume that processes can concurrently read, write, atomically add, and compare-and-swap at memory locations. Compare-and-swap (CAS) takes three arguments: a memory location $x$, an old value old$_V$, and a new value new$_V$. If the value stored at $x$ is equal to old$_V$, the CAS atomically updates the value at $x$ to be new$_V$ and returns true. Otherwise, the CAS returns false. Almost all modern processors support CAS.

We analyze time complexity with the work-span model, a standard model for analyzing shared-memory parallel algorithms [22, 39]. The work of a program execution is the total number of instructions executed, and the span is the length of the longest sequential critical path of instructions. For a program with $W$ work and $S$ span, a work-stealing scheduler like Cilk [10] can execute the program in $W/P + O(S)$ expected time with $P$ processors. A parallel algorithm whose work asymptotically matches the work of the most efficient known sequential algorithm is work-efficient, which is an important characteristic since $W/P$ is often much higher than $S$ in well-designed parallel algorithms.

#### 2.3.2 Parallel primitives

This paper makes use of many existing parallel algorithms, which we describe below.

**Hash tables:** Gil et al. present a hash table which supports inserting $k$ elements in $O(k)$ work and $O(\log^* k)$ span with both bounds being w.h.p. Looking up an element takes $O(1)$ work [32].

**Primitives on arrays:** The reduce operation computes the sum of all elements in an array. (The sum operation is often the numerical operation. For instance, reduce can compute the maximum element in an array.) The filter operation returns a subsequence of the original sequence consisting of all elements matching a user-specified predicate. For an array of $n$ elements, both operations run in $O(n)$ work and $O(\log n)$ span [9, 39]. The remove duplicates operation returns an array that has the same set of elements as the original input array has but without any duplicate elements. Removing duplicates using a parallel hash table takes $O(n)$ work and $O(\log^* n)$ span with both bounds being w.h.p.

**Comparison sort:** Cole presents a parallel merge sort that sorts $n$ elements in $O(n \log n)$ work and $O(\log \log n)$ span [21].

**Integer sort:** Suppose that we have $n$ non-negative integers in the range $[0, \text{poly}(n)]$. For any positive integer $q$, we can sort these integers in $O(qn)$ work and $O(qn^{1/9})$ span [67]. Raman provides another integer sorting algorithm that runs in $O(n \log \log n)$ work and $O(\log n / \log \log n)$ span with both bounds being w.h.p. [54].

Also, we can sort $n$ non-negative rational numbers whose numerators and denominators are bounded by $r \in \text{poly}(n)$ with the same asymptotic running times. Consider two distinct rational numbers $a/b$ and $c/d$ that meet this criterion. Their absolute difference is $|ad - bc|/|bd| \geq \lceil 1/|bd| \rceil \geq 1/r^2$. Therefore, if we multiply each rational number by $r^2$ and round them down to the nearest integer, we get $n$ integers bounded by $r^3 \in \text{poly}(n)$ whose sorted order matches the sorted order of the original rational numbers.

**Graph connectivity:** Gazit gives an algorithm for graph connectivity with $O(n + m)$ expected work and $O(\log n)$ span w.h.p. [31].

### 3 Review of SCAN algorithms

In this section, we provide an overview of the SCAN [71] and GS*-Index [68] clustering algorithms.

#### 3.1 SCAN definitions

The typical problem formulation for graph clustering is to output a partition (or clustering) of the vertices of the input graph such that each cluster in the partition has many edges within the cluster and such that there are few edges between clusters. How exactly to quantify the quality of a clustering depends on the application domain. Section 7.2 lists a few clustering quality measures.

SCAN [71] is a graph clustering algorithm on undirected graphs. The output of SCAN diverges slightly from this description of clustering; SCAN may leave some vertices unclustered. Unclustered vertices are further separated into hubs and outliers. Hubs are vertices that neighbor several clusters but do not belong to any, and outliers are unclustered vertices that neighbor at most one cluster.

For each pair of adjacent vertices $\{u, v\} \in E$, SCAN computes a similarity score $\sigma(u, v)$. The original paper assumes that edges are unweighted and defines the similarity score to be the cosine similarity of the closed neighborhoods of the two vertices:

$$
\sigma(u, v) = \text{CosineSim}(\overline{N}(u), \overline{N}(v)) = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)| \sqrt{|N(v)|}}.
$$

$\text{poly}(n)$ means $O(n^c)$ for some constant $c$. 

![Figure 1: Example SCAN clustering with $\mu = 3$ and $\epsilon = .6$. The labels on the edges are cosine similarities. Core vertices are blue whereas non-core vertices are white. Edges with similarity greater than $\epsilon$ are green whereas other edges are red. There are two clusters (vertices $\{1, 2, 3, 4\}$ and vertices $\{6, 7, 8, 11\}$) as well as three unclustered vertices (hub vertex 5 and outlier vertices 9 and 10).](image-url)
For instance, in the graph in figure 1, the cosine similarity between vertices 5 and 6 is
\[
\frac{|\{4, 5, 6\} \cap \{5, 6, 7, 8\}|}{\sqrt{|\{4, 5, 6\}| |\{5, 6, 7, 8\}|}} = \frac{2}{\sqrt{12}} \approx .58.
\]
This is just one possible choice for the similarity score, however. Other papers consider using Jaccard similarity, Dice similarity, or weighted cosine similarity for the similarity function \([16, 36, 37, 45]\).

SCAN takes two parameters as input, an integer \(\mu \geq 2\) and a similarity threshold \(\varepsilon \in [0, 1]\). Call vertices \(u\) and \(v\) \(\varepsilon\)-similar if their similarity \(\sigma(u, v)\) is at least \(\varepsilon\). The \(\varepsilon\)-neighborhood \(N_\varepsilon(v)\) of a vertex \(v\) is the set of its \(\varepsilon\)-similar neighbors, \(\{u \in N(v) \mid \sigma(u, v) \geq \varepsilon\}\).

The \textit{core} vertices are the vertices whose \(\varepsilon\)-neighborhood contains at least \(\mu\) neighbors, i.e., vertices \(v\) such that \(|N_\varepsilon(v)| \geq \mu\). A vertex \(u\) is \textit{structurally reachable} from core vertex \(v\) if there is a path of vertices \(v_1, v_2, \ldots, v_k\) for some \(k \geq 2\) where \(v_1 = v\), where \(v_k = u\), and where \(v_i\) is a core and is \(\varepsilon\)-similar to \(v_{i+1}\) for each integer \(i\) from 1 to \(k - 1\).

The two following properties define each cluster in the clustering that SCAN finds:
- The cluster is "connected": for any two vertices \(u\) and \(x\) in the cluster \(C\), there is a vertex \(v\) such that both \(u\) and \(x\) are structurally reachable from \(v\).
- The cluster is "maximal": for every core vertex \(v\) in the cluster, all vertices structurally reachable from \(v\) are also in the cluster.

Figure 1 shows the clusters that result from running SCAN on a small graph.

The \textit{border} vertices, which are the clustered non-core vertices (e.g., vertex 11 in figure 1), may belong to several distinct clusters according to the definition of SCAN clusters. The original SCAN algorithm assigns each of these ambiguous border vertices to any of its possible clusters arbitrarily.

Computing similarity scores takes \(O(am)\) time with an appropriate triangle counting algorithm; to calculate a similarity score \(\sigma(u, v)\), it suffices to count the number of shared neighbors in \(N(u) \cap N(v)\), which is precisely the number of triangles in which edge \([u, v]\) appears. After computing similarities, SCAN finds clusters by performing a modified breadth-first search, which takes \(O(n + m)\) time.

### 3.2 Index-based SCAN: GS*-Index

GS*-Index \([68]\) improves on SCAN by precomputing an index from which finding cores and \(\varepsilon\)-similar neighbors is fast for any setting of \(\mu\) and \(\varepsilon\). It takes \(O((\alpha + \log n)m)\) time to compute the index, and the index takes \(O(m)\) space. After computing the index, the time it takes to compute the clustering for arbitrary query parameters \((\mu, \varepsilon)\) depends on the size of the resulting clusters rather than on the size of the full graph. Specifically, for a subset of vertices \(C \subseteq V\), define \(E_{C, \varepsilon}\) to be the set of \(\varepsilon\)-similar edges in the subgraph induced by \(C\). Then the time to compute the clustering for parameters \((\mu, \varepsilon)\) is \(O(|\bigcup_{C \subseteq C} E_{C, \varepsilon}|)\).

Determining whether unclustered vertices are hubs or outliers is not considered in this time bound.

The index consists of two data structures, the \textit{neighbor order} \(NO\) and the \textit{core order} \(CO\). To compute the index, we first compute the similarity scores between every pair of adjacent vertices. The neighbor order is the adjacency list of the graph with each neighbor list sorted by descending similarity. Figure 2 shows the neighbor order for the graph in figure 1. The core order is an array where the \(\mu\)-th entry, \(CO[\mu]\), for any \(\mu\) is a list of vertices with degree \(|N(\cdot)|\) at least \(\mu\), i.e., all possible cores for this \(\mu\) value. The vertices in \(CO[\mu]\) are sorted by descending similarity with vertex \(NO[k][\mu]\) is \(k\)'s \textit{core threshold} value. For any \(\varepsilon\) no greater than the threshold, the vertex is a core vertex under parameters \(\mu\) and \(\varepsilon\). Figure 3 shows the core order for the graph from figure 1.

To find the clustering resulting from SCAN parameters \((\mu, \varepsilon)\), we perform a breadth-first search on the core vertices, considering only \(\varepsilon\)-similar edges in the graph and not searching further from any non-core vertices. The core vertices and \(\varepsilon\)-similar edges are easy to find from the index since the core vertices are a prefix of \(CO[\mu]\) and the \(\varepsilon\)-similar edges are prefixes of each list in \(NO\) due to the sorting. The breadth-first search reveals all the SCAN clusters in the graph.

### 4 Parallel algorithm

This section presents our new work-efficient, logarithmic-span parallel algorithms for constructing the same SCAN index that GS*-Index constructs and for retrieving clusters from the index.

For the algorithm descriptions in this section, we assume the existence of basic utility functions and of functions implementing the primitives listed in section 2.3.2. The \textsc{AllocateArray}(\(s\)) function allocates an array that holds \(s\) elements. The \textsc{MakeHashMap}(\(\cdot\)) function makes a hash table with the input argument specifying the key-value elements in the table. The \textsc{MakeHashSet}(\(\cdot\)) function makes a hash table containing only keys rather than key-value pairs.
The **SUM(·)** function returns the sum of the elements in an array via the reduce operation. The **REMOVE_DUPLICATES(·)** function returns an array that has the same set of elements that the input array has but without any duplicate values.

### 4.1 Index construction

#### 4.1.1 Computing similarities

To shorten exposition, this section will only focus on one similarity function \(\sigma(·, ·)\): cosine similarity for weighted graphs. Given a weighted undirected graph \(G = (V, E, w)\), the similarity score between two adjacent vertices \((u, v)\) in \(E\) is

\[
\sigma(u, v) = \text{WeightedCosineSim}(\overline{N}(u), \overline{N}(v)) = \frac{\sum_{x \in \overline{N}(u) \cap \overline{N}(v)} w(u, x) w(v, x)}{\sqrt{\sum_{x \in \overline{N}(u)} w(u, x)^2} \sqrt{\sum_{x \in \overline{N}(v)} w(v, x)^2}}
\]

where we set \(w(x, x) = 1\) for each vertex \(x\). This weighted cosine similarity measure is the natural generalization to the cosine similarity measure for unweighted graphs that the original SCAN and GS*-Index algorithms consider. Modifying the algorithm described in this section to instead compute the unweighted cosine similarity or Jaccard similarity is straightforward.

**Algorithm 1** Algorithm for computing the cosine similarity of each edge in a weighted graph.

**Output:** An array of length \(m\) containing the similarity score of each edge.

1. **procedure** **COMPUTE_SIMILARITIES**\((G = (V, E, w))\)
2. \(\text{norms} \leftarrow \left\{ \sqrt{\sum_{x \in \overline{N}(u)} w(u, x)^2} \mid v \in V \right\}\)
   - Compute each entry of the array norms with **SUM(·)**.
3. **similarities** ← **ALLOCATEARRAY**\((m)\)
   - \(\sigma\) for clarity, we index into similarities with edges from \(E\).
4. **neighbor_tables** ← **ALLOCATEARRAY**\((m)\)
5. for \(v \in V\) do in parallel
6. **neighbor_tables**\([v]\) ← **MAKE_HASHSET**\((\overline{N}(v))\)
7. for \((u, v) \in E\) do in parallel
8. (Without loss of generality, let \(\overline{N}(u) \leq \overline{N}(v)\))
9. **shared Neighbor Weights** ← **ALLOCATEARRAY**\((\overline{N}(u))\)
10. for \(i \in \{1, 2, 3, \ldots, |\overline{N}(u)|\}\) do in parallel
11. \(x \leftarrow i\)-th entry in \(\overline{N}(u)\)
12. **shared_neighbor_weights**\([i]\) ← \(w(u, x) \cdot w(v, x)\) if \(x \in \text{neighbor_tables}[v]\) else 0
13. **similarities**\([u, v]\) ← \(\text{SUM(shared_neighbor_weights)} / (\text{norms}[u] \cdot \text{norms}[v])\)
14. return **similarities**

Algorithm 1 gives pseudocode for computing similarities. The logic follows that of a known parallel algorithm for triangle counting [63]. The algorithm creates a hash set for each vertex’s neighborhood (lines 5 to 6). Then, for each pair of adjacent vertices \(u\) and \(v\), looking up the neighbors of \(u\) in the hash set for \(v\)’s neighborhood (lines 10 to 12) gives all of the shared neighbors between \(u\) and \(v\), which allows the algorithm to compute \(\text{WeightedCosineSim}((\overline{N}(u), \overline{N}(v))\)

(line 13).

If the algorithm always searches for neighbors of the lower-degree vertex in the hash set of the higher-degree vertex’s neighborhood, the work is \(O(\sum_{|u| \leq E} \min(|\overline{N}(u)|, |\overline{N}(v)|))\) in expectation, which is bounded by \(O(\alpha m)\) [20]. The span is \(O(\log n)\) w.h.p.

#### 4.1.2 Neighbor order and core order

After computing all similarity values, we construct the neighbor order and core order (algorithm 2). We form the neighbor order by sorting each vertex’s neighbor list by descending similarity (lines 4 to 5). Then we form the core order by, for each \(\mu\) value, finding all possible core vertices under parameter \(\mu\) (line 6) and sorting them by descending core threshold (line 7). In line 6, to find all possible core vertices (i.e., all vertices \(v\) such that \(\overline{N}(v) \geq \mu\)), we perform a doubling search on \(\text{sorted}_V\), the set of vertices sorted by descending degree (line 2). This doubling search consists of sequentially searching for the minimum \(i\) such that the \(2^i\)-th entry of \(\text{sorted}_V\) fails to satisfy the predicate \(\overline{N}(v) \geq \mu\) and performing binary search on the first \(2^i\) entries of \(\text{sorted}_V\). Doubling search is needed for optimal running time bounds. Using only binary search would add \(O(\alpha m \log n)\) to total work since each binary search costs \(O(\log n)\) work. Doubling search, on the other hand, costs only \(O(\log j)\) work to find an item located at index \(j\). The \(O(\log j)\) cost is also better than the \(O(j)\) work and span that linear search would incur.

With a work-efficient comparison sort algorithm, the work analysis is the same as the original analysis for GS*-Index. This gives bounds of \(O(\alpha m \log n)\) work and \(O(\log n)\) span for constructing the orders.

If the graph is unweighted, each Jaccard similarity is a rational number, and each unweighted cosine similarity squared is a rational number. Recall from section 2.3.2 that we can sort rational numbers with an integer sorting algorithm. Therefore, if the graph is unweighted, we can achieve better work bounds by using integer sorting rather than comparison sorting. In order to apply the integer sort running time bounds directly when computing the neighbor order, instead of sorting \(\text{NO}[v]\) separately for each \(v \in V\) like algorithm 2 describes, we instead prepend \(v\) to every entry in \(\text{NO}[v]\) for each \(v \in V\) and sort all elements in \(\text{NO}\) with a single integer sort. We perform the same transformation to compute the core order with one integer sort. By doing this, the asymptotic running time bounds for computing the neighbor order and core order match the running time bounds for integer sort on \(m\) integers from section 2.3.2.

Summing the running time bounds for computing similarities with the bounds for constructing the neighbor and core order gives the following theorems.

**Theorem 4.1.** Fix an undirected, weighted graph and let \(\alpha\) be its arboricity. The parallel SCAN index construction algorithm using cosine similarity as the similarity measure runs in \(O((\alpha + \log m)n)\) work (which matches the work bound of GS*-Index) and \(O(\log n)\) span on the graph with both bounds being w.h.p.
Theorem 4.2. Fix an undirected, unweighted graph and let $\alpha$ be the graph’s arboricity. The parallel SCAN index construction algorithm with cosine similarity or Jaccard similarity as the similarity measure can achieve the following running time bounds depending on what integer sorting algorithm is used:

- $O((\alpha + \log \log n)m)$ work and $O(\log n)$ span with both bounds being w.h.p.,
- $O(am)$ work and $O(\log n)$ span for any fixed $\beta \in (0, 1]$ with both bounds being w.h.p.

4.2 Querying for clusters

Next, we describe an efficient parallel algorithm for discovering clusters given the parameters $\mu$ and $\epsilon$. The algorithm uses the index structure from section 4.1.

Algorithm 3 Helper function for finding core vertices under a particular setting of SCAN parameters.

Output: An array of core vertices under SCAN parameters $(\mu, \epsilon)$.

\begin{verbatim}
1: procedure GETCORES($\mu, \epsilon$, NO, CO, similarities)
2: max_degree ← $|CO|$  
3: if $\mu > \text{max\_degree}$ then  
4: return ()  
5: else  
6: return $\{v \in CO | \text{similarities}[\{v, NO[v][\mu]\}] \geq \epsilon\}$  
\end{verbatim}

Algorithm 4 Helper function for assigning border non-core vertices to clusters after clustering all the core vertices.

\begin{verbatim}
1: procedure ASSIGNNONCORES(similar_edges, cores_set, clusters)
2: subgraph, vertices ← REMOVEDUPLICATES$(\{u, v\} \in \text{similar\_edges})$
3: subgraph_non_cores ← $\{v \in \text{subgraph, vertices} | v \notin \text{cores\_set}\}$  
4: non_cores_count ← $|\text{subgraph, non\_cores}|$
5: assignments ← ALLOCATEARRAY$(\text{non\_cores\_count})$
6: non_cores_indices ← MAKEHASHMAP$(\text{subgraph, non\_cores}[i] \rightarrow i)$
7: for $i \in \{1, 2, 3, \dots, \text{non\_cores\_count}\}$ do in parallel  
8: assignments[i] = null  
9: for $\{u, v\} \in \text{similar\_edges} \land (u \in \text{cores\_set} \land v \notin \text{cores\_set})$ do in parallel  
10: (Without loss of generality, let $v \notin \text{cores\_set}$. Then $u \in \text{cores\_set}$.)
11: address ← $&$(assignments[non_cores_indices[u]])
12: COMPAREANDSWAP$(\text{address}, \text{null}, \text{clusters}[u])$  
13: Assign border vertex $v$ to an arbitrary neighboring similar cluster.
14: return clusters
\end{verbatim}

Algorithm 5 Algorithm for finding the SCAN clustering with parameters $\mu$ and $\epsilon$ from the index.

\begin{verbatim}
1: procedure SCAN($\mu, \epsilon$, NO, CO, similarities)
2: cores ← GETCORES($\mu$, NO, CO, similarities)
3: cores_set ← MAKEHASHTABLE$(\text{cores})$
4: similar_edges ← $((u, v) | u \in \text{cores\_set} \land \text{similarities}[\{u, v\}] \geq \epsilon)$
5: $\Rightarrow$ Get similar edges by doubling search on $NO[u]$ for each $u$ in cores.
6: similar_core_edges ← $((u, v) \in \text{similar\_edges} | u \in \text{cores\_set} \land v \in \text{cores\_set})$  
7: $\Rightarrow$ Filter core clusters: Connected components of subgraph induced by similar_core_edges, represented as a hash table mapping $[u \rightarrow \text{component\_ID}]$ for each $v$ in cores.
8: return ASSIGNNONCORES$(\text{similar\_edges, cores\_set, core\_clusters})$
\end{verbatim}

Algorithm 5 provides pseudocode for extracting a clustering with arbitrary parameters from the index. Algorithms 3 and 4 are subroutines for algorithm 5. To retrieve the clustering with parameters $\mu$ and $\epsilon$, the algorithm performs a doubling search on $CO[\mu]$ to find all core vertices (line 6 of algorithm 3) and then performs doubling searches on $NO[v]$ for each core vertex $v$ to find all $\epsilon$-similar edges incident on core vertices (line 4 of algorithm 5). For instance, for the graph in figure 1 with parameters $(\mu, \epsilon) = (3, 6)$, the search on $CO[\mu]$ finds the blue vertices in figure 3, and the searches on $NO[v]$ find the green vertices in figure 2. This corresponds exactly to the blue core vertices and green edges in figure 1.

For each of these prefixes of $NO[v]$, the algorithm also creates a copy with all border non-core neighbors (e.g., vertex 11 in figure 1) filtered away (line 5 of algorithm 5). These filtered prefixes constitute an adjacency list for the subgraph induced by the $\epsilon$-similar edges on the core vertices. Running a parallel connectivity algorithm on this subgraph assigns all core vertices to a cluster (line 6 of algorithm 5). Finally, the algorithm takes all the border non-core neighbors (lines 2 to 3 of algorithm 4) and uses compare-and-swap to assign each of them to the same cluster as an arbitrary $\epsilon$-similar core (line 12 of algorithm 4). The final output is a hash table mapping vertices to cluster IDs. The algorithm achieves the running time bounds stated in the following theorem.

Theorem 4.3. Suppose the clustering algorithm, algorithm 5, runs and returns a collection of clusters $C$. For a set of vertices $\mathcal{C} \subseteq C$, define $E_{C, \mathcal{C}}$ to be the set of $\epsilon$-similar edges in the subgraph induced by $\mathcal{C}$. Define $Z = \bigcup_{\mathcal{C} \subseteq C} E_{C, \mathcal{C}} \in O(m)$. Then the run of the clustering algorithm had $O(Z)$ expected work (which matches the work bound for GS*-Index) and $O(\log Z)$ span w.h.p.

This theorem holds because the doubling searches in lines 2 to 4 of algorithm 5 fetch all the edges $\bigcup_{\mathcal{C} \subseteq C} E_{C, \mathcal{C}}$ in the output clustering $\mathcal{C}$ in a work-efficient manner, and the remainder of the clustering algorithm operates only on the subgraph given by $\bigcup_{\mathcal{C} \subseteq C} E_{C, \mathcal{C}}$ in a parallel, efficient manner.

4.3 Determining hubs and outliers

After finding a clustering, we can determine whether unclustered vertices are hubs or outliers. For each unclustered vertex $v$, we map each neighbor in $N(v)$ to its cluster ID and reduce over the neighbors to determine whether the vertex has neighbors belonging to distinct clusters. It takes $O(|N(v)| \log Z)$ work and $O(\log |N(v)|)$ span to determine whether $v$ is a hub or outlier.

5 Approximating similarities

After constructing the index, querying for a clustering is fast. Index construction itself, though, may be expensive since it takes $O(am)$ work. One unexplored technique for speeding up SCAN is to use LSH to approximate similarities.

For example, to use SimHash to approximate cosine similarities, we fix a sample size $k \in \mathbb{N}$. Then, we draw $kn$ random numbers from the standard normal distribution, which is possible via the Box-Muller transform [14] given a source of uniform random numbers. With these normally distributed random numbers, we construct a $k$-sample sketch of $N(v)$ for each vertex $v$. The sketching takes $O(km)$ work and $O(\log n)$ span using the reduce operation to compute inner products. Now we can compute the similarity between any adjacent vertices $u$ and $v$ by comparing their sketches in $O(k)$ work and $O(\log k)$ span. Computing the sketches and the similarities over all edges takes $O(km)$ work and $O(n \log k)$ span. The work bound is better than the work bound for computing exact similarities if $k$ is asymptotically less than the arboricity $\alpha$. Similarly, we can use MinHash to approximate Jaccard similarities.
We can then compute a neighbor order and core order on these similarities. Again we can achieve better work bounds using an integer sort algorithm, and in fact we can use integer sort on both unweighted and weighted graphs. This is because the approximate similarities are non-negative integers scaled by a factor of $\pi/k$ for SimHash or $1/k$ for MinHash, and we can postpone scaling the integers until after sorting. Therefore, we can construct a SCAN index with the following running time bounds.

**Theorem 5.1.** Fix an undirected graph and let $k \leq \text{poly}(n)$. The parallel SCAN index construction algorithm using $k$-sample MinHash (for unweighted graphs) or SimHash (for unweighted or weighted graphs) to compute approximate similarities can achieve the following running time bounds depending on what integer sorting algorithm is used:

- $O((k + \log \log n)m)$ work and $O(\log(n))$ span with both bounds being w.h.p.,
- $O(km)$ work and $O(n^\beta)$ span for any fixed $\beta \in (0, 1]$.

We can also theoretically analyze the clusters resulting from using these approximate similarities. In particular, suppose we fix some $\varepsilon \in [0, 1]$ and $\delta \in (0, 1)$. Notice that the SCAN clustering with parameters $\varepsilon$ and arbitrary $\mu$ is only concerned about whether similarities fall above or below $\varepsilon$ rather than exact similarity values. If the number of samples is sufficiently high, then w.h.p., all edges outside the similarity range $\varepsilon \pm \delta$ will be “correctly classified” as above or below the threshold $\varepsilon$ by the approximate similarities. We present such a result for approximating cosine similarity using SimHash.

**Theorem 5.2.** Let $G = (V, E, w)$ be an undirected graph with non-negative edge weights, let $\varepsilon \in [0, 1]$, and let $\delta \in (0, 1)$. Suppose $k \geq \pi^2 \ln(nm)/(2\delta^2)$ and suppose we use SimHash with $k$ samples to compute approximate cosine similarity scores for every edge in $G$. Then w.h.p., all edges with exact cosine similarities outside the interval $(\varepsilon - \delta, \varepsilon + \sqrt{1 - \varepsilon^2}\delta)$ are correctly classified by the approximate similarities as above or below the threshold $\varepsilon$.

**Proof.** Consider an arbitrary edge $\{u, v\} \in E$ with an exact cosine similarity $s \in [0, 1]$ outside the interval $(\varepsilon - \delta, \varepsilon + \sqrt{1 - \varepsilon^2}\delta)$. It suffices to prove that the edge is correctly classified by the approximate cosine similarity with probability at least $1 - 1/(nm)$. Then, applying a union bound over all $m$ edges gives that all edges outside the similarity interval are classified correctly w.h.p.

Let $\theta = \cos(s)$ be the angle between the vectors corresponding to $\overrightarrow{u}(u)$ and $\overrightarrow{v}(v)$. The angle is in the range $[0, \pi/2]$ since all edge weights are non-negative. Recall from section 2.1.2 that the SimHash estimate for the angle between the two vectors is $\hat{\theta} \sim \text{Binomial}(k, \theta/\pi) \cdot \pi/k$. Hoeffding’s inequality [35] implies that given arbitrary $\ell \in \mathbb{N}$, $p \in [0, 1]$, and $t > 0$, for a binomial random variable $X \sim \text{Binomial}(\ell, p)$, the probabilities $\Pr[X/\ell \geq p + t]$ and $\Pr[X/\ell \leq p - t]$ are each bounded above by $\exp(-2rt^2)$. Using this inequality on $\hat{\theta}$ with $t = k$, $p = \theta/\pi$, and $t = \delta/\pi$ gives that both $\Pr[\hat{\theta} \geq \theta + \delta]$ and $\Pr[\hat{\theta} \leq \theta - \delta]$ are each bounded above by $\exp(-2k\delta^2/\pi^2) \leq 1/(nm)$.

Let $\phi = \arccos(s) \in [0, \pi/2]$ be the similarity threshold $\epsilon$ transformed into an angle threshold. First consider the case where $s \in [0, \epsilon - \delta]$, which also implies that $\epsilon \geq \delta$. The straight line from the

![Figure 4: Plot of the SimHash approximation lower and upper bound functions on cosine for $\varepsilon = 0.9$. The bold point is $(\phi, \epsilon)$.](image)

point $(\phi, \epsilon)$ to the point $(\pi/2, 0)$ has the equation

$$f_\epsilon(x) = \epsilon - \frac{\epsilon}{\pi/2 - \phi} (x - \phi),$$

which figure 4 shows in red for $\varepsilon = .9$. By concavity of the cosine function in $[0, \pi/2]$, we have that $\cos(x) \geq f_\epsilon(x)$ when $x \in [\phi, \pi/2]$. We have that $\phi + \delta = \arccos(\epsilon) + \delta \leq \arccos(\delta) + \delta \leq \pi/2$; the first inequality comes from the arc cosine function being decreasing combined with the constraint that $\epsilon \geq \delta$, and the second inequality comes from taking derivatives to maximize $\arccos(\delta) + \delta$ for $\delta \in (0, 1)$. Therefore, we can substitute $\phi + \delta$ for $x \in \arccos(s) \geq f_\epsilon(x)$ to get that $\cos(\phi + \delta) \geq \epsilon - (\pi/2 - \phi)^2$. Plotting the multiplicative factor $\frac{1}{\pi/2 - \phi}$ with varying $\phi$ shows that the factor falls in the range $[2/\pi, 1]$, giving a looser but cleaner bound that $\cos(\phi + \delta) \geq \epsilon - \delta \geq s$. Taking the arc cosine of the leftmost and rightmost sides of the inequality gives $\phi + \delta = \arccos(\phi + \delta) \leq \arccos(s) = \theta$, where the first equality uses the fact that $\phi + \delta \in [0, \pi/2]$. Now the upper bound on $\Pr[\theta \leq \delta - \delta]$ gives that the probability that $\hat{\theta} > \theta - \delta \geq \phi$ is at least $1 - 1/(nm)$. Taking the cosine of both sides gives that the cosine similarity estimate $\cos(\hat{\theta})$ falls below $\epsilon$ with probability at least $1 - 1/(nm)$ as desired.

Consider next the case where $s \in [\epsilon + \sqrt{1 - \epsilon^2}\delta, 1]$. If $\epsilon = 1$, then $s = 1$ and SimHash will always return the correct estimate $\cos(\hat{\theta}) = \cos(0) = 1$ as desired. For $\epsilon < 1$, define $h(\delta) = (1 - \delta^2)/(1 + \delta^2)$ and note that this case implies that $r + \sqrt{1 - \epsilon^2}\delta \leq 1$, which rearranges to the constraint $\epsilon \leq h(\delta)$. Linearize the cosine function at the input point $\epsilon$ to get the line

$$g_\epsilon(x) = \epsilon - \sin(\phi)(x - \phi) = \epsilon - \sin(\arccos(\epsilon))(x - \phi) = \epsilon - \sqrt{1 - x^2}(x - \phi),$$

which figure 4 shows in blue for $\varepsilon = .9$. By concavity of the cosine function, we have that $\cos(x) \leq g_\epsilon(x)$ when $x \in [0, \pi/2]$. Note that we have that $\phi - \delta = \arccos(\epsilon) - \delta \geq \arccos(h(\delta)) - \delta \geq 0$; the first inequality comes from the arc cosine function being decreasing combined with the constraint that $\epsilon \leq h(\delta)$, and the second inequality comes from plotting $\arccos(h(\delta)) - \delta$ to see that it is non-negative for $\delta \in (0, 1)$ . Hence we can substitute $\phi - \delta$ for $x$ in the inequality $\cos(x) \leq g_\epsilon(x)$ to get that $\cos(\phi - \delta) \leq \epsilon + \sqrt{1 - \epsilon^2}\delta \leq s$. Taking the arc cosine of the leftmost and rightmost sides of the inequality gives $\phi - \delta = \arccos(\cos(\phi - \delta)) \geq \arccos(s) = \theta$, where the first equality uses the fact that $\phi - \delta \in [0, \pi/2]$. Now the upper bound on $\Pr[\theta \leq \delta - \delta]$ gives that the probability that $\hat{\theta} < \theta + \delta \leq \phi$ is at least $1 - 1/(nm)$, Taking the cosine of both sides gives that the cosine similarity estimate $\cos(\hat{\theta})$ is above $\epsilon$ with probability at least $1 - 1/(nm)$ as desired. □
We also present a similar result for approximating the Jaccard similarity using MinHash.

**Theorem 5.3.** Let $G = (V, E)$ be an undirected graph, let $\epsilon \in [0, 1]$, and let $\delta \in (0, 1)$. Suppose $k \geq \ln(nm)/(2\delta^2)$ and suppose we use standard MinHash with $k$ samples to compute approximate Jaccard similarity scores for every edge in $G$. Then w.h.p., all edges with exact Jaccard similarities outside the interval $(\epsilon - \delta, \epsilon + \delta)$ are correctly classified by the approximate similarities as above or below the threshold $\epsilon$.

**Proof.** The result follows from applying Hoeffding’s inequality as in the proof of theorem 5.2. We omit full details for brevity. \hfill \Box

Though the bounds in these theorems require a large number of samples $k$ to achieve high accuracy, our experiments in section 7 show that lower values of $k$ still achieve good clusterings. This approximation strategy helps for denser graphs with high arboricity and many high-degree vertices. For low arboricity graphs, it is faster to simply compute exact similarities.

**6 Implementation**

We implement the algorithms described in sections 4 and 5 to determine how they perform in practice. We write our code in C++ within the Graph Based Benchmark Suite (GBBS) framework [25, 27]. GBBS provides libraries useful for implementing parallel graph algorithms. Our implementations use the concurrent hash table implementation [61], parallel sorting algorithms, and various graph processing helper functions that GBBS provides.

Though the algorithms as described in sections 4 and 5 achieve good theoretical bounds, our actual implementations make several changes for better performance. This section explains the more significant changes.

**6.1 Computing similarities**

We implement similarity computation for both cosine similarity and Jaccard similarity. Experiments by Shun and Tangwongsan [63] suggest that the hash-based approach to triangle counting or computing similarities in section 4.1.1 incurs many cache misses and that a merge-based approach is faster in comparison even though it increases the asymptotic work bound from $O(m\alpha r)$ to $O(m^{3/2})$. Our implementation uses the merge-based approach of Shun and Tangwongsan. This approach assumes that each neighbor list in the adjacency list of the input graph is sorted by vertex number, which is true for graphs converted to GBBS’s graph file format. In order to count each triangle only once and hence reduce work, we construct a directed version of the input graph by filtering each neighbor list so as to direct each edge towards its higher-degree vertex. Then, for each pair of adjacent vertices $(u, v)$, we find triangles of the form $\{(u, v), (v, x), (u, x)\}$ for $x \in N(u) \cap N(v)$ by merging the out-neighborhoods of $u$ and $v$ in the directed graph. To get similarity scores for each pair of adjacent vertices, the implementation maintains an atomic counter for each edge and increments the counters for all three edges of any triangle found.

The merge logic between two neighbor lists follows the logic of the parallel merge implementation in GBBS: if both neighbor lists are small, we iterate through the sorted neighbor lists sequentially to find shared neighbors; if one neighbor list is small and the other is large, then we search for each element of the small neighbor list in the larger list via binary search; and finally, if both neighbor lists are large, then we split them into smaller sub-lists and recursively merge the sub-lists in parallel.

**6.2 Querying for clusters**

Most work-efficient parallel connectivity algorithms are complicated and lack readily available implementations. Instead, in our implementation for querying the index for clusters, we find the connected components on the core vertices by using the concurrent union-find implementation in the GBBS codebase [26]. Using union-find allows us to avoid materializing the subgraph to pass to a connectivity algorithm. Instead, we populate an $n$-length array on the fly where each entry is the cluster ID of the corresponding vertex. Having this array also simplifies the logic for ASSIGNNONCORES (algorithm 4) by changing ASSIGNNONCORES to skip the preprocessing in lines 2–8 and instead compare-and-swap directly into the cluster ID array.

**6.3 Approximate similarities**

We implement similarity approximation logic using both SimHash and MinHash. For MinHash, we use a variant called $k$-partition MinHash or one permutation hashing [41]. It is more computationally efficient than the original version of MinHash; computing a sketch of a vertex $v$ takes only $O(k + |N(v)|)$ work using $k$-partition MinHash rather than $O(k|N(v)|)$ work using standard MinHash since $k$-partition MinHash generates a $k$-length sketch using only one permutation rather than $k$ permutations. The $k$-partition variant still provides reasonable clustering results, but the accuracy bound in theorem 5.3 no longer applies for this variant; though $k$-partition MinHash has lower variance than standard MinHash, it is no longer clear whether there is a convenient tail bound for $k$-partition MinHash like the Hoeffding bound that the proof of theorem 5.3 uses.

When the number of samples $k$ for the LSH approximation scheme is high, it becomes more expensive to compute and process sketches to get approximate similarities. For low-degree vertices, the merging algorithm described in section 6.1 is cheap and cache-friendly enough that it is better to compute similarities exactly rather than approximate them. As a simple example, if two adjacent vertices have degree significantly less than $k$, it is faster and more accurate to process the original neighbor lists of the vertices instead of their $k$-length sketches. To avoid sketching low-degree vertices, we add a heuristic to choose which vertices to sketch and which similarities to approximate. The heuristic is to only approximate similarities between pairs of vertices that both have sufficiently high degree and to compute similarities exactly with triangle counting for all other pairs of vertices. We determine whether a vertex is high degree by checking whether its degree exceeds a threshold value of $k$ for approximate cosine similarity and $3k/2$ for approximate Jaccard similarity. No sketches are needed for vertices that either do not have high degree or do not have any neighbors with high degree.

**7 Experiments**

Our timing experiments show that our implementation achieves good speedup over sequential baselines and performs competitively against ppSCAN [18], a state-of-the-art parallel shared-memory SCAN implementation. Our results for our approximate SCAN implementation suggest that LSH can speed up index construction significantly on dense graphs without sacrificing clustering quality.
“Orkut” and “Friendster” are the com-putational focus of our parallel algorithm (which we refer to as GBB-SIndexSCAN) on graphs with fewer than two million edges and do not report times. Stovall et al. [64] only test their GPU-based SCAN algorithm on graphs with fewer than six million edges, whereas we focus on much larger graphs in our experiments.

7.1 Benchmarking environment

We run experiments on an Amazon Elastic Cloud Compute (EC2) c5.24xlarge instance, which has 192 GiB of RAM and 48 CPU cores with two-way hyper-threading for a maximum of 96 threads. We compare our parallel algorithm (which we refer to as GBB-SIndexSCAN in the experimental plots) using all 96 threads to ppSCAN’s authors show that ppSCAN out-performs other existing parallel SCAN algorithms (pSCAN [16], anySCAN [45], and SCAN-XP [65]). For fixed parameters μ and ε, all of these algorithms return the same output except that ambiguous border vertices may have different assignments. All code is C++ code compiled with GCC 7.5.0 using the -march=native flag. The GBBSSIndexSCAN code uses GBBS’s scheduler library [8] written using standard C++ threads. We run the parallel codes with numactl --interleave=all, which interleaves memory allocations across CPUs and gives better performance for this particular problem on the EC2 instance.

Table 2 summarizes the graphs that we use in the experiments. “Orkut” and “Friendster” are the com-Orkut and com-Friendster graphs from the Stanford Large Network Dataset Collection [40]. Both are social network graphs in which the vertices are users and the edges represent friend relationships. “Brain” is the bn-human-Jung2015-M87113878 dataset from NeuroData.[11, 12] The brain connections. “WebBase” is the webbase-2001 graph from the Laboratory for Web Algorithmics [11, 12].

| Name         | Number of vertices | Number of edges | Type   |
|--------------|--------------------|-----------------|--------|
| Orkut        | 3,072,441          | 117,185,083     | unweighted |
| brain        | 784,262            | 267,844,669     | unweighted |
| WebBase      | 118,142,155        | 854,809,761     | unweighted |
| Friendster   | 65,608,366         | 1,806,067,135   | unweighted |
| blood vessel | 25,825             | 70,240,269      | weighted |
| cochlea      | 25,825             | 282,977,319     | weighted |

Table 2: Summary of the graphs for the experiments.

Non-shared-memory parallel algorithms fail to outperform our implementation as well. Zhao et al.’s reported timings for their distributed SCAN algorithm [76] are much slower than our times; Zhao et al. report taking 36 minutes with fifteen eight-core machines to cluster their largest graph, which has four million vertices and sixty million edges, whereas our algorithm takes under three seconds to process the larger, denser Orkut graph using fewer cores. Chen et al. [19] and Zhou and Wang [77] only test their distributed SCAN algorithms on graphs with fewer than six million edges and do not report times. Stovall et al. [64] only test their GPU-based SCAN algorithm on graphs with fewer than six million edges, whereas we focus on much larger graphs in our experiments.

7.2 Clustering quality measures

We evaluate our clustering results using the modularity and adjusted Rand index measures. These quality measures are popular and consistent with existing graph clustering literature. The modularity of a clustering is the proportion of edges that fall within clusters in the clustering minus the expected number of edges that would fall within clusters in a random graph with the same degree distribution [49]. Specifically, fix some clustering and let \( \delta_{u,v} \) for arbitrary vertices \( u \) and \( v \) be 1 if \( u \) and \( v \) are assigned the same cluster and be 0 otherwise. The modularity is

\[
\frac{1}{2m} \sum_{u,v \in V} \left( A_{u,v} - \frac{|N(u)||N(v)|}{|G|} \right) \delta_{u,v}.
\]

The definition of modularity also easily extends to apply to weighted graphs [48]. Higher modularity scores suggest better clusterings.

Another way to measure the quality of a proposed clustering is to check how similar it is against a known ground-truth clustering. The adjusted Rand index (ARI) [38] is one well-known metric for evaluating this similarity. ARI counts the number of pairs of vertices such that the two vertices are assigned to the same clusters or to different clusters in both the proposed clustering and the ground-truth clustering. This count is then adjusted for chance. Let \( C \) be the proposed clustering on the set of \( n \) vertices \( V \) and let \( G \) be the ground-truth clustering. For integers \( i \) in \( \{1, 2, 3, \ldots, |C|\} \) and \( j \) in \( \{1, 2, 3, \ldots, |G|\} \), let \( n_{i,j} \) be the number of vertices in both cluster \( i \) of \( C \) and cluster \( j \) of \( G \). Let \( n_{+} = \sum_{i=1}^{|C|} n_{i,j} \) and let \( n_{+,j} = \sum_{i=1}^{|C|} n_{i,j} \) for each \( i \) and \( j \). Then the ARI between \( C \) and \( G \) is

\[
\frac{\sum_{i=1}^{|C|} \sum_{j=1}^{|G|} \left( n_{i,j} \right)^2 - \sum_{i=1}^{|C|} \left( n_{i,+} \right)^2 \sum_{j=1}^{|G|} \left( n_{+,j} \right)^2}{\left( \sum_{i=1}^{|C|} \left( n_{i,+} \right)^2 \right) \left( \sum_{j=1}^{|G|} \left( n_{+,j} \right)^2 \right) / 2 - \sum_{i=1}^{|C|} \left( n_{i,+} \right)^2 \sum_{j=1}^{|G|} \left( n_{+,j} \right)^2 / 2}.
\]

Higher ARI scores suggest a better match with the ground-truth clustering. Neither the modularity nor the ARI can exceed 1, and they may be negative if the clustering is “worse than random.”

7.3 Results

7.3.1 Index construction time comparison

The first experiment measures the running time to construct the SCAN index with exact cosine similarity. The time to compute the index using Jaccard similarity flag. The GBBSIndexSCAN code uses GBBS’s scheduler library [8] written using standard C++ threads. We run the parallel codes with numactl --interleave=all, which interleaves memory allocations across CPUs and gives better performance for this particular problem on the EC2 instance.

| Name         | Number of vertices | Number of edges | Type   |
|--------------|--------------------|-----------------|--------|
| Orkut        | 3,072,441          | 117,185,083     | unweighted |
| brain        | 784,262            | 267,844,669     | unweighted |
| WebBase      | 118,142,155        | 854,809,761     | unweighted |
| Friendster   | 65,608,366         | 1,806,067,135   | unweighted |
| blood vessel | 25,825             | 70,240,269      | weighted |
| cochlea      | 25,825             | 282,977,319     | weighted |

Table 2: Summary of the graphs for the experiments.
similarity is about the same (at most 9% difference for GBBSIndexSCAN on 96 threads), so we do not report it separately. Each time measurement is the median of five trials. Figure 5 shows the time measurements. GBBSIndexSCAN achieves a parallel self-speedup of 23–70× on index construction. Moreover, GBBSIndexSCAN running sequentially is 1.4–2.2× faster than the original GS*-Index implementation, likely due to the directed triangle counting optimization that section 6.1 describes, so the speedup of GBBSIndexSCAN running on 96 threads is 50–151× over GS*-Index.

7.3.2 Clustering time comparison The second experiment is to measure the running time for querying for the clustering over various settings of parameters ($\mu$, $\epsilon$). The plots only consider exact cosine similarity since there are about the same using Jaccard similarity (at most either $10^{-4}$ absolute difference or 4% difference for GBBSIndexSCAN on 96 threads). Each time measurement is the median of five trials. Figure 6 measures the running times with $\mu = 5$ and $\epsilon \in \{.1, .2, .3, \ldots, .9\}$, and figure 7 measures the running times with $\epsilon = 0.6$ and $\mu \in \{2, 4, 8, 16, \ldots, \min\{16384, 2^{\lceil \log(\text{max degree}) \rceil}\}\}$. GBBSIndexSCAN is faster than ppSCAN and GS*-Index on all tested parameter settings, though this ignores the time GBBSIndexSCAN takes to precompute its index. This precomputation cost that GBBSIndexSCAN incurs is preferable over ppSCAN only when the user makes many queries. Notably, though, it might not take many queries for GBBSIndexSCAN to become preferable over ppSCAN. For example, on the Orkut and Friendster graphs, the Orkut and Friendster graphs, the time measurements for ppSCAN on the nine parameter settings in figure 6 exceeds the time measurements for GBBSIndexSCAN plus the index construction time for GBBSIndexSCAN.

Sequentially, GBBSIndexSCAN can be slower at querying for clusters than GS*-Index due to adjustments made in GBBSIndexSCAN to make it more friendly to parallelism, namely using union-find instead of sequential breadth-first search as well as iterating over all edges an additional time to assign non-core vertices (algorithm 4). It is up to 4.5× slower than GS*-Index on the tested parameter settings and graphs. GBBSIndexSCAN running on 96 threads, however, is faster than the other implementations on all tested parameter settings; it is faster than GS*-Index by 5–32× and faster than ppSCAN by 1.26–12.07×.

7.3.3 Approximate index construction time The third experiment measures the running time of constructing GBBSIndexSCAN with 96 threads using the approximate cosine and approximate Jaccard similarity measures with varying numbers of samples. For the weighted graphs, we only test the approximate cosine similarity measure since the $k$-partition MinHash variant that we implement for Jaccard similarity does not handle weighted graphs. Each time measurement is again a median of five trials. Each trial uses a different pseudorandom seed for the randomness in the approximation schemes. Figure 8 displays the results. As anticipated, the best speedups are on the denser graphs like the cochlea graph, whereas approximation is unhelpful on the Orkut graph even with modest sample sizes. The approximate Jaccard similarity implementation is consistently faster than the approximate cosine similarity implementation because of the better efficiency of constructing sketches for $k$-partition MinHash compared to SimHash. The times plateau or even decrease at large sample sizes for some graphs due to the heuristic discussed in section 6.3 for avoiding sketching for low-degree vertices.

7.3.4 Quality of approximate clusterings The fourth experiment measures the quality of the clusterings achieved with the approximate similarity measures compared to the clusterings achieved with the exact similarity measures. Although the ASSIGNNONCORES (algorithm 4) portion of the clustering algorithm assigns each border non-core vertex to the same cluster as an arbitrary $r$-similar core vertex, to get consistent measurements for this experiment, we remove this source of non-determinism by assigning each border vertex to the same cluster as the most similar neighboring core vertex, breaking ties in favor of lower vertex IDs.

We use the modularity as a heuristic measurement for clustering quality, treating unclustered vertices as each being in its own cluster. We select the parameters ($\mu$, $\epsilon$) maximizing modularity in the following set $\Sigma$:

$$\Sigma = \{2, 4, 8, 16, \ldots, 2^{18}\} \times \{.01, .02, .03, \ldots, .99\}.$$  (1)

Figure 9 plots the best modularity scores found when using the approximate similarity measures with varying numbers of samples. To better illustrate the trade-off between computation time and clustering quality, we use the index construction times from figure 8 on the horizontal axis instead of the number of samples. Each plotted modularity score for a fixed number of samples is the mean of five trials with different pseudorandom seeds on each trial.

Similarly, figure 10 plots the ARI of the clustering found by the approximate similarity measures with varying numbers of samples versus the “ground-truth” clustering under the corresponding exact similarity measures. Again, each plotted ARI is the mean of five trials with different pseudorandom seeds. The SCAN parameters used in this plot are the best parameters in $\Sigma$ relative to the exact similarity measures. Hence this plot shows how well the clusterings using approximate similarity measures match the clusterings using exact similarity measures at a particular parameter setting.

Points to the top and the left represent sample sizes that give good quality as well as low index construction times. The figures also include the times to construct indices with exact similarity measures from figure 5 with the assumption that the times for exact Jaccard similarity are the same as those measured for exact cosine similarity.

The improved approximation accuracy in these plots as the sample size increases is not only attributable to better LSH accuracy with more samples but also to the heuristic described in section 6.3 that reverts to computing exact similarity for vertices that have low degree relative to the number of samples.
To represent the running time for exact index construction. Since construction times using exact cosine similarity versus using exact Jaccard similarity are similar, we only plot one point.

Figure 6: Clustering time with $\mu = 5$ and varying $\epsilon$ using exact cosine similarity as the similarity measure.

Figure 7: Clustering time with $\epsilon = 0.6$ and varying $\mu$ using exact cosine similarity as the similarity measure.

Figure 8: Index construction times for GBBSIndexSCAN (96 threads) using approximate similarity measures with varying sample sizes.

Figure 9: Trade-off curve of approximate similarity index construction time with varying numbers of samples versus the best modularity score found among parameters from $\Sigma$ (equation (1)). The index construction times on the horizontal axis come from figure 8.

Figure 10: Trade-off curve of approximate similarity index construction time with varying numbers of samples versus the accuracy (measured via adjusted Rand index) of the resulting approximate clustering relative to a “ground-truth” clustering from the corresponding exact-similarity index. For each graph, the tested parameters ($\mu, \epsilon$) are the modularity-maximizing parameters from $\Sigma$ (equation (1)) relative to the exact similarity measure. The index construction times on the horizontal axis come from figure 8. Since construction times using exact cosine similarity versus using exact Jaccard similarity are similar, we only plot one point to represent the running time for exact index construction.
The approximate Jaccard clusterings approach the quality of the corresponding exact similarity clusterings at lower sample sizes than approximate cosine clusterings do, which is perhaps expected due to the better sampling efficiency that MinHash variants tend to have over SimHash as suggested by Shrivastava and Li [60] and by our bounds in theorems 5.2 and 5.3.

The ARI scores indicate that clusterings found from using approximate similarity measures at a particular parameter setting sometimes do not match well with the clusters found from the corresponding exact similarity measures unless the number of samples is high. The modularity scores, on the other hand, suggest that by searching over a range of parameter values, it is possible to significantly speed up index construction on denser graphs by approximating similarities while still achieving a good quality clustering.

8 Related Work

Xu et al. introduced the original SCAN algorithm [71] and borrowed ideas from the popular point clustering algorithm DBSCAN [29]. One major inconvenience of SCAN is the difficulty of choosing its two user-selected parameters, $\gamma$ and $\epsilon$. GS*-Index alleviates this issue by creating an index upon which future SCAN queries with arbitrary parameters are efficient [68]. SCOT [13] and gSkeletonClu [37] also essentially compute indices for SCAN, but only for a fixed $\gamma$ value. SCOT outputs an ordering of vertices, similar to what the OPTICS algorithm [1] outputs for DBSCAN, such that vertices that tend to be in the same cluster are nearby in the ordering. gSkeletonClu computes a spanning tree upon potential core vertices.

SHRINK [36], DHSCAN [73], and AHSCAN [74] are all based on SCAN but avoid the parameter selection issue by being parameter-free algorithms that use a quality function like the modularity to guide the clustering process. DPSCAN [69] is another parameter-free SCAN-based algorithm that uses a density metric to select clusters. These algorithms are easier to use due to their lack of parameters, although having tunable parameters can be helpful in allowing the user to explore alternative clusterings.

Other work building on SCAN focuses on making SCAN scale to large graphs. LinkSCAN* [42] reduces computation time at the cost of accuracy by operating on a sampled subgraph of the original graph. It may be worthwhile in the future to compare the efficiency and clustering quality of the LinkSCAN* sampling approach versus the LSH approach of our paper. Zhao et al. [75] and Mai et al. [45] describe anytime algorithms for SCAN, with Mai et al.’s algorithm being parallel. Users may pause queries and examine intermediate clustering results, making it useful for large graphs on which finishing a query may take a long time. Our work, on the other hand, strives to make finishing a query as fast as possible so that this anytime functionality is unnecessary.

SCAN++ [59], pSCAN [16], and ppSCAN [18], for a fixed setting of SCAN parameters, speed up SCAN by pruning many unnecessary similarity score computations between pairs of vertices. Che et al.’s ppSCAN is parallel and uses vectorized instructions as well for additional performance. SCAN-XP [65] is another parallel SCAN algorithm but does not perform pruning.

For distributed systems, Chen et al. [19] and Zhao et al. [76] present MapReduce [23] parallelizations of SCAN, and SparkSCAN [77] is a Spark parallelization of SCAN. GPUSCAN [64], on the other hand, explores using GPUs to speed up SCAN, whereas we focus on CPU-based algorithms in this paper.

There are many other graph clustering algorithms besides SCAN and its variants. Interested readers may refer to surveys written by other researchers such as Schaeffer [56] and Fortunato [30].

9 Conclusion

This paper presents index-based SCAN algorithms that achieve significant parallel speedup over the state of the art. They allow users to query efficiently for SCAN clusterings for arbitrary parameter settings. The algorithms achieve improved work bounds over GS*-Index due to the use of integer sorting and have logarithmic span w.h.p. We also present a highly optimized multicore implementation of the algorithm that runs well in practice. Moreover, we demonstrate that LSH is a viable approximation scheme to speed up the computationally expensive component of index construction on dense graphs without large sacrifices in cluster quality.

For future work, first, we are interested in extending our work to dynamic graphs by devising parallel algorithms for processing batches of edge updates. Second, we are interested in quickly extracting hierarchical clusterings from the SCAN index. Third, we would like to investigate whether using other similarity measures like weighted Jaccard similarity or a random-walk-based similarity produces better or faster clusters. Last, we wish to compare parallel SCAN to other parallel clustering algorithms in quality and speed.

Acknowledgments

This research was supported by DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Michael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points To Identify the Clustering Structure. SIGMOD Record 28, 2 (1999), 49-50.

[2] Kevin Aydin, MohammadHossein Bateni, and Vahab Mirrokni. 2016. Distributed Balanced Partitioning via Linear Embedding. In Proceedings of the Ninth ACM International Conference on Web Search and Data Mining. 387–396.

[3] Artifal Azad, Georgios A. Pavlopoulos, Christos A. Ouzounis, Nikos C. Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. Nucleic Acids Research 46, 6 (2018), e33.

[4] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. Machine Learning 56, 1-3 (2004), 89–113.

[5] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshian, Mohammad-Taghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Allioryr Clustering: Hierarchical Clustering at Scale. In Advances in Neural Information Processing Systems. 6864–6874.

[6] Alejandro Bellogín and Javier Parapar. 2012. Using Graph Partitioning Techniques for Neighbour Selection in User-Based Collaborative Filtering. In Proceedings of the Sixth ACM Conference on Recommender Systems. 213–216.

[7] Chris Biersmann. 2006. Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. In Proceedings of the First Workshop on Graph-based Methods for Natural Language Processing. 73–80.

[8] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In Proceedings of the 52nd Symposium on Parallelism in Algorithms and Architectures. 507–509.

[9] Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel Algorithms. In Algorithms and Theory of Computation Handbook: Special Topics and Techniques (2nd ed.), Mikhail J. Atallah and Marina Blanton (Eds.). Vol. 2. Chapter 25.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. J. ACM 46, 5 (1999), 720–748.
[11] 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. 587–596.

[12] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In Proceedings of the 13th International Conference on World Wide Web. 595–602.

[13] Dustin Bortner and Jiawei Han. 2010. Progressive Clustering of Networks Using Structure-Connected Order of Traversal. In IEEE 26th International Conference on Data Engineering. 653–656.

[14] George E. P. Box and D. R. Muller. 1958. A note on the generation of random normal deviates. The Annals of Mathematical Statistics 29, 2 (1958), 610–611.

[15] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity of SEQUENCES. 21–29.

[16] Lijun Chang, Wei Li, Lu Qin, Wenyue Zhang, and Shiyu Yang. 2017. pSCAN: Fast and Exact Structural Graph Clustering. IEEE Transactions on Knowledge and Data Engineering 29, 2 (2017), 387–401.

[17] Moses S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing. 380–388.

[18] Yulin Che, Shixuan Sun, and Qing Luo. 2018. Parallelizing Pruning-based Graph Structural Clustering. In Proceedings of the 47th International Conference on Parallel Processing. Article 77.

[19] Jia-Jun Chen, Ji-Meng Chen, Jie Liu, and Va-Lou Huang. 2013. PSCAN: A Parallel Structural Clustering Algorithm for Networks. In International Conference on Machine Learning and Cybernetics, Vol. 2. 839–844.

[20] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. SIAM Journal on computing 14, 1 (1985), 210–223.

[21] Richard Cole. 1988. Parallel merge sort. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing. 226–231.

[22] Santo Fortunato. 2010. Community detection in graphs.

[23] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining. 222–229.

[24] Yang Ding, Minjun Chen, Zhichao Liu, Don Ding, Yanbin Ye, Min Zhang, Reagan Zelaya, Daniel S. Himmelstein, Ran Zhang, Boris M. Hartmann, Elena Zaslavsky, and Yue Zhou. 2020. The Graph Based Benchmark Suite (GBBS). In Proceedings of Workshop on Collective Knowledge Capturing and Representation. 82–90.

[25] Symeon Papadopoulos, Yannis Kompatzis, and Athena Vakali. 2009. Leveraging Collective Intelligence through Community Detection in Tag Networks. In Proceedings of Workshop on Collective Knowledge Capturing and Representation.

[26] Symeon Papadopoulos, Yannis Kompatzis, and Athena Vakali. 2010. A Graph-Based Clustering Scheme for Identifying Related Tags in Folksonomies. In Data Warehousing and Knowledge Discovery. 65–76.

[27] Symeon Papadopoulos, Christos Zikogiannis, Giorgos Tolias, Yannis Kalantidis, Phivos Mylonas, Yannis Kompatzis, and Athena Vakali. 2010. Image clustering through community detection on hybrid image similarity graphs. In IEEE International Conference on Image Processing. 2353–2356.

[28] Rajeev Raman. 1990. The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting. In Foundations of Software Technology and Theoretical Computer Science. 266–273.

[29] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. 4292–4293.

[30] Satu Elisa Schaeffer. 2007. Graph clustering. Computer Science Review 1, 1 (2007), 26–74.

[31] Manos Schinas, Symeon Papadopoulos, Yannis Kompatzis, and Pericles A. Mitkas. 2015. Visual Event Summarization on Social Media Using Topic Modeling and Graph-based Ranking Algorithms. In Proceedings of the 5th ACM International Conference on Multimedia Retrieval. 203–210.

[32] Manos Schinas, Symeon Papadopoulos, Georgios Petkos, Yannis Kompatzis, and Pericles A. Mitkas. 2015. Multimodal Graph-Based Event Detection and Summarization in Social Media Streams. In Proceedings of the 23rd ACM International Conference on Multimedia. 189–192.

[33] Hiroshi Shiozawa, Yasufuru Fujawa, and Makoto Onizuka. 2015. SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-scale Graphs. Proceedings of the VLDB Endowment 8, 11 (2015), 1178–1189.

[34] Shunsuke Shiritava and Ping Li. 2014. In Defense of MinHash Over SimHash. In Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics. 866–874.

[35] Julian Shun and Guy Edward Blelloch. 2014. Phase-Contiguous Hash Tables for Determinism. In Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures. 96–107.
[62] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel Local Graph Clustering. Proceedings of the VLDB Endowment 9, 12 (2016), 1041–1052.
[63] Julian Shun and Kanat Tangwongsan. 2015. Multicore Triangle Computations Without Tuning. In IEEE 31st International Conference on Data Engineering, 149–160.
[64] Thomas Ryan Stovall, Sinan Kockara, and Recep Avci. 2015. GpuSCAN: GPU-based Parallel Structural Clustering Algorithms for Networks. IEEE Transactions on Parallel and Distributed Systems 26, 12 (2015), 3381–3393.
[65] Tomokatsu Takahashi, Hiroaki Shukawa, and Hiroyuki Kitagawa. 2017. SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors. In Proceedings of the 2nd International Workshop on Network Data Analytics. Article 6.
[66] David A. Tolliver and Gary L. Miller. 2006. Graph Partitioning by Spectral Rounding: Applications in Image Segmentation and Clustering. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Vol. 1. 1053–1060.
[67] Uzi Vishkin. 2010. Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques.
[68] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and XueMin Lin. 2017. Efficient structural graph clustering: an index-based approach. Proceedings of the VLDB Endowment 11, 3 (2017), 243–255.
[69] Changfa Wu, Yu Gu, and Ge Yu. 2019. DPSCAN: Structural Graph Clustering Based on Density Peaks. In Database Systems for Advanced Applications. 628–641.
[70] Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. 2020. A Review for Weighted MinHash Algorithms. IEEE Transactions on Knowledge and Data Engineering (Early Access) (2020).
[71] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: A Structural Clustering Algorithm for Networks. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 824–833.
[72] Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. 2017. Local Higher-Order Graph Clustering. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 555–564.
[73] Nurcan Yuruk, Mutlu Mete, Xiaowei Xu, and Thomas A. J. Schweiger. 2007. A Divisive Hierarchical Structural Clustering Algorithm for Networks. In Proceedings of the Seventh IEEE International Conference on Data Mining Workshops. 441–448.
[74] Nurcan Yuruk, Mutlu Mete, Xiaowei Xu, and Thomas A. J. Schweiger. 2009. AH-SCAN: Agglomerative Hierarchical Structural Clustering Algorithm for Networks. In Proceedings of the International Conference on Advances in Social Network Analysis and Mining. 72–77.
[75] Weizhong Zhao, Gang Chen, and Xiaowei Xu. 2017. AnySCAN: An Efficient Anytime Framework with Active Learning for Large-scale Network Clustering. In IEEE International Conference on Data Mining. 665–674.
[76] Weizhong Zhao, Venkataswamy Martha, and Xiaowei Xu. 2013. PSCAN: A parallel structural clustering algorithm for big networks in MapReduce. In Proceedings of the IEEE 27th International Conference on Advanced Information Networking and Applications. 862–869.
[77] Qijun Zhou and Jingbin Wang. 2016. SparkSCAN: A Structure Similarity Clustering Algorithm on Spark. In Big Data Technology and Applications. 163–177.