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, achieving logarithmic span (parallel time) 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 behavior.

We present an experimental evaluation of our 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 faster index construction while maintaining good clustering quality.

CCS Concepts

• **Theory of computation → Shared memory algorithms**; **Graph algorithms analysis**; • **Information systems → Clustering**.

Keywords

Multicore algorithms, Graph clustering, Locality-sensitive hashing

ACM Reference Format:
Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel Index-Based Structural Graph Clustering and Its Approximation. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457278

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457278

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, 23, 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.).[1] 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 matrix multiplication on dense graphs and using integer sort improve the index construction work bound compared to GS*-Index's bound.

[1] 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.

Description	Work	Span
Exact index, weighted graph	$O((\alpha + \log n)m)$ w.h.p.	$O(\log n)$ w.h.p.
Exact index, unweighted graph	$O((\alpha + \log\log n)m)$ w.h.p. $O(\alpha m)$ w.h.p.	$O(\log n)$ w.h.p. $O(n^{\beta})$ w.h.p.
Approximate index	$O((k + \log\log n)m)$ w.h.p. $O(km)$	$O(\log n)$ w.h.p. $O(n^{\beta})$

Table 1: Summary of asymptotic running time bounds for index construction. The arboricity of the input graph is α, the number of samples used for approximation is k, and $0 < \beta \leq 1$. For the exact indices on dense graphs, the αm work term may be replaced by n^{ω_p}, where $n^{\omega_p} \leq n^{2.373}$ is the asymptotic work to multiply two n-by-n matrices in logarithmic span.

To further improve performance, we show how to use locality-sensitive hashing (LSH) to speed up similarity computation. We provide a non-trivial theoretical analysis of the accuracy of LSH for SCAN. Our experiments show that LSH speeds up index construction while preserving good clustering quality. Table 1 summarizes the asymptotic running time bounds for index construction.

We present 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 LSH-based 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 a single thread. Furthermore, our parallel index query implementation, which extracts a clustering for a specific set of parameters from the index, achieves 5–32× speedup over GS*-Index queries across a range of SCAN parameter values. Our implementation also achieves faster query times on all tested parameter values compared to ppSCAN [18], a state-of-the-art parallel SCAN algorithm.

The contributions of this paper are as follows:

(**1**) We present a new parallel index-based SCAN algorithm that matches the work bounds of the sequential GS*-Index algorithm and has logarithmic span w.h.p. We also show how matrix multiplication and integer sorting improve the work bounds.
(**2**) We introduce the use of locality-sensitive hashing as an approximation technique for SCAN that is provably efficient and has behavior guarantees relative to exact SCAN.
(**3**) We evaluate our algorithm on large real-world graphs. Our experiments demonstrate that our implementation outperforms other existing SCAN algorithms and confirm that locality-sensitive hashing provides running time improvements.
(**4**) We release the implementation of our algorithm.[2]

2 Preliminaries

This section provides background definitions and concepts that subsequent sections use.

2.1 Set similarity

2.1.1 Similarity measures Two common measures for the similarity of two sets A and B with elements from a finite universe U are the Jaccard similarity and the cosine similarity:

$$\text{JaccardSim}(A, B) = \frac{|A \cap B|}{|A \cup B|}, \quad \text{CosineSim}(A, B) = \frac{|A \cap B|}{\sqrt{|A|}\sqrt{|B|}}.$$

If the sets are weighted and have weight functions $w_A, w_B : U \mapsto \mathbb{R}$, then there is a weighted form of cosine similarity:

$$\text{WeightedCosineSim}(A, B) = \frac{\sum_{x \in A \cap B} w_A(x) w_B(x)}{\sqrt{\sum_{x \in A} w_A(x)^2}\sqrt{\sum_{x \in B} w_B(x)^2}}.$$

(There is also a weighted version of Jaccard similarity, which we do not consider in this work.)

The cosine similarity is really a similarity measure between non-zero vectors; given vectors u and v with an angle of θ between the two vectors, the cosine similarity is defined as

$$\text{CosineSim}(u, v) = \cos(\theta) = \frac{u \cdot v}{\|u\|\|v\|}.$$

Defining 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 π 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 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 are more computationally efficient such as k-partition MinHash [41]. There are also variants for weighted Jaccard similarity [70]. Since the weighted variants are more complicated and less practical, 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 θ/π; because v has uniformly random direction, the orthogonal hyperplane to v separates a and b with probability θ/π, 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 π/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] Code: https://github.com/ParAlg/gbbs/tree/master/benchmarks/SCAN/IndexBased

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 by $G = (V, E, w)$, where the weight function $w : E \to \mathbb{R}$ 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 is the size of its neighborhood, $N(v)$.

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 $(v, u) \in E$.

The *arboricity* α 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 $O(\alpha m)$ time [20].

We represent graphs as adjacency lists, in which each vertex has a list of its neighbors. We only consider simple graphs, i.e., graphs with at most one edge between any pair of vertices and no 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 [24]. 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 the complexity of algorithms 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, such as the one in 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 when run on large data sets.

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 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 addition operation but more generally may be any associative binary 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 w.h.p.

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

Integer sort: Suppose that we have n non-negative integers in the range $[0, \text{poly}(n)]$.[3] For any positive integer q, we can sort these integers in $O(qn)$ work and $O(qn^{1/q})$ 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 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 $|ad - bc|/|bd| \geq 1/|bd| \geq 1/r^2$. 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 $O(m + n)$ expected work and $O(\log n)$ span w.h.p. [31].

Matrix multiplication: Two n-by-n matrices can be multiplied in $O(n^{\omega_p})$ work and $O(\log n)$ span with parallel matrix multiplication constant $\omega_p \leq 2.373$ [26].

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 there are few edges between clusters. How exactly to quantify the quality of a clustering depends on the application domain. Section 7.2 lists two clustering quality measures.

SCAN [71] is a graph clustering algorithm on undirected graphs. The output of SCAN diverges slightly from this description of clustering in that SCAN may leave some vertices unclustered. Unclustered vertices are further separated into *hubs* and *outliers*. Hubs are unclustered vertices that neighbor multiple clusters, and outliers are unclustered vertices that neighbor at most one cluster.

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

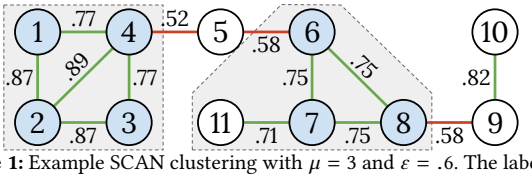

Figure 1: Example SCAN clustering with $\mu = 3$ and $\varepsilon = .6$. The labels on the edges are cosine similarities. Core vertices are blue, whereas non-core vertices are white. Edges with similarity greater than ε are green, whereas other edges are red. There are two clusters (vertices $\{1, 2, 3, 4\}$ and vertices $\{6, 7, 8, 11\}$) as well as three unclustered vertices (hub vertex 5 and outlier vertices 9 and 10).

For each pair of adjacent vertices $\{u, v\} \in E$, SCAN computes a similarity score $\sigma(u, v)$. The original paper [71] assumes that edges are unweighted and defines the similarity score to be the cosine similarity of the closed neighborhoods of the two vertices:

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

For instance, in the graph in Figure 1, the cosine similarity between vertices 5 and 6 is

$$\frac{|\{4, 5, 6\} \cap \{5, 6, 7, 8\}|}{\sqrt{|\{4, 5, 6\}|}\sqrt{|\{5, 6, 7, 8\}|}} = \frac{2}{\sqrt{12}} \approx .58.$$

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 *ε-similar* if their similarity $\sigma(u, v)$ is at least ε. The *ε-neighborhood* $N_\varepsilon(v)$ of a vertex v is the set of its ε-similar neighbors, $\{u \in \overline{N}(v) \mid \sigma(u, v) \geq \varepsilon\}$. The *core* vertices are the vertices whose ε-neighborhood contains at least μ neighbors, i.e., vertices v such that $|N_\varepsilon(v)| \geq \mu$. A vertex u is *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 ε-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 *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 $O(\alpha m)$ 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 (BFS), which takes $O(n + m)$ time.

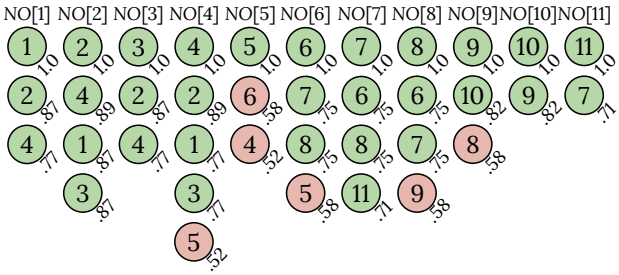

Figure 2: Neighbor order for the graph from Figure 1. In this figure, for each $v \in V$, we display $\mathcal{NO}[v]$ as a column. The numbers beside each vertex are similarity scores. For example, in $\mathcal{NO}[3]$, the .87 label beside vertex 2 represents the cosine similarity of .87 between vertices 3 and 2. Like in Figure 1, we consider the specific case where $\varepsilon = 0.6$ and color all ε-similar neighbors green and all other neighbors red.

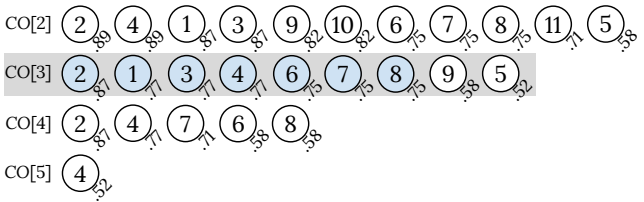

Figure 3: Core order for the graph from Figure 1. Each entry of $\mathcal{CO}$ is displayed horizontally. We omit $\mathcal{CO}[1]$ since we assume $\mu \geq 2$. The number beside each vertex in a row $\mathcal{CO}[\mu]$ is the core threshold for that vertex for that μ. For example, in $\mathcal{CO}[2]$, the number .75 beside vertex 6 means that when $\mu = 2$ and $\varepsilon \leq .75$, vertex 6 is a core vertex. Like in Figure 1, we consider the specific case where $(\mu, \varepsilon) = (3, .6)$; we highlight $\mathcal{CO}[\mu]$ in gray and color the core vertices (i.e., vertices with core threshold at least ε) blue.

3.2 Index-based SCAN: GS*-Index

GS*-Index [68] improves on SCAN by precomputing an index from which finding cores and ε-similar neighbors is fast for any setting of μ and ε. 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 (μ, ε) depends on the size of the resulting clusters rather than on the size of the full graph. Specifically, for a subset of vertices $U \subseteq V$, define $E_{U,\varepsilon}$ to be the set of ε-similar edges in the subgraph induced by U. Then the time to compute the clustering C for parameters μ and ε is $O\left(\left|\bigcup_{U \in C} E_{U,\varepsilon}\right|\right)$. Determining whether unclustered vertices are hubs or outliers is not considered in this time bound.

The index consists of two data structures, the *neighbor order* $\mathcal{NO}$ and the *core order* $\mathcal{CO}$. 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 non-increasing similarity. Figure 2 shows the neighbor order for the graph in Figure 1. The core order is an array where the μ-th entry, $\mathcal{CO}[\mu]$, for any μ is a list of vertices with $|\overline{N}(\cdot)|$ at least μ, i.e., all possible core vertices for this μ value. The vertices in $\mathcal{CO}[\mu]$ are sorted by non-increasing similarity with vertex $\mathcal{NO}[\cdot][\mu]$. This similarity of a vertex v with vertex $\mathcal{NO}[v][\mu]$ is v's *core threshold* value. For any ε no greater than the threshold, the vertex is a core vertex under parameters μ and ε. Figure 3 shows the core order for the graph in Figure 1. For example, to compute $\mathcal{CO}[3]$ in Figure 3, we consider the nine vertices $\{1, 2, 3, \ldots, 9\}$ with $|\overline{N}(\cdot)| \geq 3$, determine

their core thresholds by looking at the similarities in the third row of Figure 2, and sort the vertices by non-increasing core threshold.

To find the clustering resulting from SCAN parameters μ and ε, we perform a BFS on the core vertices, considering only ε-similar edges in the graph and not searching further from any non-core vertices. The core vertices and ε-similar edges are easy to find from the index since the core vertices are a prefix of $CO[\mu]$ and the ε-similar edges are prefixes of each list in $\mathcal{NO}$ due to the sorting. The BFS 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 AllocateArray(s) function allocates an array that holds s elements. The MakeHashMap($\cdot$) function makes a hash table with the input argument specifying the key-value elements in the table. The MakeHashSet($\cdot$) function makes a hash table containing only keys rather than key-value pairs. The Sum($\cdot$) function returns the sum of the elements in an array via the reduce operation. The RemoveDuplicates($\cdot$) 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 $\sigma(\cdot,\cdot)$: 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

$$\begin{aligned}\sigma(u,v) &= \text{WeightedCosineSim}(\overline{N}(u), \overline{N}(v)) \\ &= \frac{\sum_{x\in\overline{N}(u)\cap\overline{N}(v)} w(u,x)w(v,x)}{\sqrt{\sum_{x\in\overline{N}(u)} w(u,x)^2}\sqrt{\sum_{x\in\overline{N}(v)} w(v,x)^2}}\end{aligned}$$

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 gives pseudocode for computing similarities. It follows a known parallel algorithm for triangle counting [63]. The algorithm creates a hash set for each vertex 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 the shared neighbors between u and v, allowing the algorithm to compute 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 $O\left(\sum_{\{u,v\}\in E} \min\{|\overline{N}(u)|, |\overline{N}(v)|\}\right)$ in expectation, which is bounded by $O(\alpha m)$ [20]. The span is $O(\log n)$ w.h.p.

For dense graphs, we can use matrix multiplication to obtain a work bound of $O(n^{\omega_p})$. Let W be an n-by-n matrix with $W_{u,v} = w(u, v)$ for arbitrary vertices u and v. Then $(W^2)_{u,v}$ is the numerator of WeightedCosineSim($\overline{N}(u), \overline{N}(v)$), so we can skip Lines 4 to 12 and substitute $(W^2)_{u,v}$ for Sum(*shared_neighbor_weights*) on Line 13.

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** ComputeSimilarities($G = (V, E, w)$)
2: *norms* $\leftarrow \left\{\sqrt{\sum_{u\in\overline{N}(v)} w(u,v)} : v \in V\right\}$
▷ Compute each entry of the array *norms* with Sum($\cdot$).
3: *similarities* $\leftarrow$ AllocateArray(m)
▷ For clarity, we index into *similarities* with edges from E.
4: *neighbor_tables* $\leftarrow$ AllocateArray(n)
5: **for** $v \in V$ **do in parallel**
6: *neighbor_tables*[v] $\leftarrow$ MakeHashSet($\overline{N}(v)$)
7: **for** $\{u, v\} \in E$ **do in parallel**
8: (Without loss of generality, let $|\overline{N}(u)| \le |\overline{N}(v)|$.)
9: *shared_neighbor_weights* $\leftarrow$ AllocateArray($|\overline{N}(u)|$)
10: **for** $i \in \{1, 2, 3, \ldots, |\overline{N}(u)|\}$ **do in parallel**
11: $x \leftarrow i$-th element in $\overline{N}(u)$
12: *shared_neighbor_weights*[i] $\leftarrow$ $w(u, x) \cdot w(v, x)$ **if** $x \in$ *neighbor_tables*[v] **else** 0
13: *similarities*[$\{u, v\}$] $\leftarrow$ Sum(*shared_neighbor_weights*)/(*norms*[u] $\cdot$ *norms*[v])
14: **return** *similarities*

Algorithm 2 Algorithms for computing the neighbor order and core order.

1: **procedure** MakeNeighborOrder($G = (V, E, w)$, *similarities*)
2: $\mathcal{NO} \leftarrow$ AllocateArray(n)
3: **for** $v \in V$ **do in parallel**
4: $\mathcal{NO}[v] \leftarrow \overline{N}(v)$
5: Sort u in $\mathcal{NO}[v]$ by non-increasing *similarities*[$\{u, v\}$] value.
6: **return** $\mathcal{NO}$
7: **procedure** MakeCoreOrder($G = (V, E, w)$, $\mathcal{NO}$)
8: *sorted_V* $\leftarrow V$ sorted by non-increasing degree.
9: *max_degree* $\leftarrow \max_{v\in V} |\overline{N}(v)|$
10: $CO \leftarrow$ AllocateArray(*max_degree*)
11: **for** $\mu = \{2, 3, 4, \ldots, \textit{max\_degree}\}$ **do in parallel**
12: $CO[\mu] \leftarrow \{v \in V \mid |\overline{N}(v)| \ge \mu\}$
▷ Find $\{v \in V \mid |\overline{N}(v)| \ge \mu\}$ by doubling search on *sorted_V*.
13: Sort v in $CO[\mu]$ by non-increasing *similarities*[$\{v, \mathcal{NO}[v][\mu]\}$] value.
14: **return** $CO[\mu]$

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 non-increasing similarity (Lines 4 to 5). Then, we form the core order by, for each μ value, finding all possible core vertices under parameter μ (Line 12) and sorting them by non-increasing core threshold (Line 13). On Line 12, to find all possible core vertices (i.e., all vertices v such that $|\overline{N}(v)| \ge \mu$), we perform a doubling search on *sorted_V*, the set of vertices sorted by non-increasing degree (Line 8). This doubling search consists of sequentially searching for the minimum i, such that the 2^i-th entry of *sorted_V* fails to satisfy the predicate $|\overline{N}(\cdot)| \ge \mu$, and then performing binary search on the last interval of the doubling search. Doubling search is needed for optimal work bounds. Using only binary search would add $O(n \log n)$ in total to the 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, giving bounds of $O(m \log n)$ 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 $\mathcal{NO}[v]$ separately for each $v \in V$ like Algorithm 2 describes, we instead prepend v to every entry in $\mathcal{NO}[v]$ for each $v \in V$ and sort all elements in $\mathcal{NO}$ with a single integer sort. We perform the same transformation to compute the core order with one integer sort. By doing this, the complexity for computing the neighbor order and core order match the complexity for integer sort on m integers, as described in Section 2.3.2.

Summing similarity computation bounds with the neighbor and core order construction bounds gives the following theorems.

Theorem 4.1. *Fix an undirected, weighted graph and let α be its arboricity. Running the parallel SCAN index construction algorithm on the graph using cosine similarity as the similarity measure runs in $O((\alpha + \log n)m)$ work (matching the work bound of GS*-Index) and $O(\log n)$ span w.h.p.*

Theorem 4.2. *Fix an undirected, unweighted graph and let α 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 w.h.p.,*
- *$O(\alpha m)$ work and $O(n^{\beta})$ span w.h.p. for any $0 < \beta \leq 1$.*

In both theorems, we can replace the αm work term with n^{ω_p} if we use matrix multiplication to compute similarities.

4.2 Querying for clusters

Next, we describe an efficient parallel algorithm for discovering clusters given the parameters μ and ϵ. 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 (μ, ε).
1: procedure GetCores(μ, ε, NO, CO, similarities)
2:     max_degree ← |CO|
3:     if μ > max_degree then return {}                ▷ No vertices are cores.
4:     else return {v ∈ CO[μ] | similarities[{v, NO[v][μ]}] ≥ ε}
                                  ▷ Find cores using a doubling search on CO[μ].
```

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 μ and ε, the algorithm performs a doubling search on $\mathcal{CO}[\mu]$ to find all core vertices (Line 4 of Algorithm 3) and then performs doubling searches on $\mathcal{NO}[v]$ for each core vertex v to find all ε-similar edges incident on core vertices (Line 4 of Algorithm 5). For instance, for the graph in Figure 1 with parameters $(\mu, \varepsilon) = (3, .6)$, the search on $\mathcal{CO}[\mu]$ finds the blue vertices in Figure 3, and the searches on

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

```
1:  procedure AssignNonCores(similar_edges, cores_set, clusters)
2:      subgraph_vertices ← RemoveDuplicates({v | {u, v} ∈ similar_edges})
3:      subgraph_non_cores ← {v ∈ subgraph_vertices | v ∉ cores_set}  ▷ Filter
4:      non_cores_count ← |subgraph_non_cores|
5:      assignments ← AllocateArray(non_cores_count)
6:      non_core_indices ← MakeHashMap({subgraph_non_cores[i] ↦ i})
7:      for i ∈ {1, 2, 3, ..., non_cores_count} do in parallel
8:          assignments[i] = null
9:      for {u, v} ∈ similar_edges∧(u ∉ cores_set∨v ∉ cores_set) do in parallel
10:         (Without loss of generality, let v ∉ cores_set; then, u ∈ cores_set.)
11:         address ← &(assignments[non_core_indices[v]])
12:         CompareAndSwap(address, null, clusters[u])
                ▷ Assign border vertex v to an arbitrary neighboring ε-similar cluster.
                  If the CAS fails, then that means v is already assigned.
13:     For v in subgraph_non_cores in parallel, insert
            [v ↦ assignments[non_core_indices[v]]] into clusters.
14:     return clusters
```

Algorithm 5 Algorithm for finding the SCAN clustering with parameters μ and ε from the index.

```
1: procedure Cluster(μ, ε, NO, CO, similarities)
2:     cores ← GetCores(μ, ε, NO, CO, similarities)
3:     cores_set ← MakeHashSet(cores)
4:     similar_edges ← {{u, v} | u ∈ cores_set ∧ similarities[{u, v}] ≥ ε}
           ▷ Get similar_edges by doubling search on NO[u] for each u ∈ cores.
5:     similar_core_edges ←
           {{u, v} ∈ similar_edges | u ∈ cores_set ∧ v ∈ cores_set}   ▷ Filter
6:     core_clusters ← Connected components of subgraph induced by
           similar_core_edges, represented as a hash table mapping
           [v ↦ component ID] for each v ∈ cores.
7:     return AssignNonCores(similar_edges, cores_set, core_clusters)
```

$\mathcal{NO}[\cdot]$ 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 $\mathcal{NO}[v]$, 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 ε-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 of 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 neighboring ε-similar core (Line 12 of Algorithm 4). The final output is a hash table mapping vertices to cluster IDs. The algorithm achieves the bounds stated in the following theorem.

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

This theorem holds because the doubling searches in Lines 2 to 4 of Algorithm 5 fetch all of the edges $\bigcup_{U \in \mathcal{C}} E_{U,\varepsilon}$ 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 $\bigcup_{U \in \mathcal{C}} E_{U,\varepsilon}$ in a work-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 $O(|N(v)|)$ work and $O(\log|N(v)|)$ span to determine whether v is a hub or outlier. In total, this takes $O(m+n)$ work and $O(\log n)$ span.

5 Approximating similarities

After constructing the index, querying for a clustering is fast. Index construction itself, though, may be expensive since it takes $\Omega(\min\{\alpha m, n^{\omega}\})$ 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 $\overline{N}(v)$ 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 $O(\log n + \log k)$ span. The work bound is better than the work bound for computing exact similarities if k is asymptotically less than the arboricity α. Similarly, we can use MinHash to approximate Jaccard similarities.

We can then compute a neighbor order and core order based 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 π/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 the integer sorting algorithm used:*

- *$O((k+\log\log n)m)$ work and $O(\log n)$ span w.h.p.,*
- *$O(km)$ work and $O(n^{\beta})$ span for $0 < \beta \leq 1$.*

We can also theoretically analyze the clusterings that result from these approximate similarities. In particular, suppose we fix some $\varepsilon \in [0,1]$ and $\delta \in (0,1)$. The SCAN clustering with parameters ε and arbitrary μ is only concerned about whether similarities fall above or below ε, 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 ε 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 ε.*

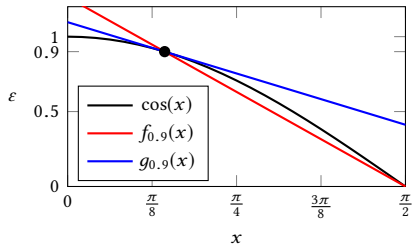

Figure 4: Plot of the SimHash approximation lower and upper bound functions on cosine for $\varepsilon = 0.9$. The bold point is (ϕ, ε).

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 $\overline{N}(u)$ and $\overline{N}(v)$. 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 $\exp(-2\ell t^2)$. Using this inequality on $\hat{\theta}$ with $\ell = k$, $p = \theta/\pi$, 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 $\exp\left(-2k\delta^2/\pi^2\right) \leq 1/(nm)$.

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

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

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

Next, consider the case where $s \in [\varepsilon + \sqrt{1-\varepsilon^2}\delta, 1]$. If $\varepsilon = 1$, then $s = 1$ and SimHash will always return the correct estimate $\cos(\hat{\theta}) = \cos(0) = 1$ as desired. For $\varepsilon < 1$, define $h(\delta) = \left(1 - \delta^2\right)/\left(1 + \delta^2\right)$

and note that

$$\varepsilon + \sqrt{1-\varepsilon^2}\delta \leq 1 \iff \delta \leq \frac{1-\varepsilon}{\sqrt{1-\varepsilon^2}} \iff$$

$$\delta^2 \leq \frac{(1-\varepsilon)^2}{1-\varepsilon^2} = \frac{1-\varepsilon}{1+\varepsilon} \iff \delta^2 + \varepsilon\delta^2 \leq 1-\varepsilon \iff \varepsilon \leq h(\delta)$$

Next, linearize the cosine function at the input point ϕ to get the line

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

which Figure 4 shows in blue for $\varepsilon = .9$. By concavity of the cosine function, we have that $\cos(x) \leq g_\varepsilon(x)$ when $x \in [0, \pi/2]$. Note that we have that $\phi - \delta = \arccos(\varepsilon) - \delta \geq \arccos(h(\delta)) - \delta \geq 0$; the first inequality comes from the arccosine function being decreasing combined with the constraint that $\varepsilon \leq h(\delta)$, 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 $\cos(x) \leq g_\varepsilon(x)$ to get that $\cos(\phi-\delta) \leq \varepsilon + \sqrt{1-\varepsilon^2}\delta \leq s$. Taking the arccosine 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 $\Pr[\hat{\theta} \geq \theta + \delta]$ 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 ε 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 $\varepsilon \in [0,1]$, 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 similarity scores for every edge in G . Then w.h.p., all edges with exact Jaccard similarities outside the interval $(\varepsilon - \delta, \varepsilon + \delta)$ are correctly classified by the approximate similarities as above or below the threshold ε.*

Proof. The result follows from applying Hoeffding's inequality as in the proof of Theorem 5.2. We omit full details for brevity. □

Though the bounds in these theorems require a large number of samples k to achieve high accuracy, experiments in Section 7 show that lower values of k still achieve good clusterings. This approximation strategy helps for denser graphs with large arboricity.

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 [24, 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 Algorithm 1 incurs many cache misses and that a merge-based approach is faster in comparison, even though it increases the asymptotic work bound from $O(m\alpha)$ 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.

To compute similarities using matrix multiplication instead of merge, we use the Intel Math Kernel Library's `cblas_sgemm` function for matrix multiplication.[4] Though its documentation does not provide asymptotic running time bounds, it runs well in practice.

6.2 Querying for clusters

When querying the index for clusters (Algorithm 5), we find the connected components on the core vertices (Line 6) by using the concurrent union-find implementation from the GBBS codebase [25]. Using union-find allows us to avoid materializing the subgraph to pass to a work-efficient connectivity algorithm. We "union" the edges in *similar_core_edges* (Line 5) and apply "find" to each vertex to populate an n-length array of vertices' cluster IDs rather than a hash table as described in Line 6. 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 $O(k + |\overline{N}(v)|)$ work using k-partition MinHash, rather than $O(k|\overline{N}(v)|)$ work using standard MinHash, since k-partition MinHash generates a k-length sketch using only

[4] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

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.

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 merge-based algorithm described in Section 6.1 is cheap and cache-friendly enough that it is better to compute similarities exactly rather than approximately. 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 while maintaining good clustering quality.

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; they 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 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 EC2 `c5.24xlarge` instance, which has 192 GiB of RAM and 48 CPU cores with two-way hyper-threading for a maximum of 96 hyper-threads. We enable hyper-threading in our parallel experiments by default. We implement our parallel algorithm using the merge-based approach for computing similarities described in Section 6.1 (GBBSIndexSCAN in the experimental plots) as well as using matrix multiplication (GBBSIndexSCAN-MM in the plots). We compare our parallel algorithm using all 48 cores with hyper-threading to our algorithm using only 1 thread, to the original sequential GS*-Index implementation,[5] and to ppSCAN with AVX2 instructions[6] using all 48 cores with hyper-threading. ppSCAN's authors show that ppSCAN outperforms 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 written is C++ code, and compiled with GCC 7.5.0 using the `-O3` optimization 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. Each time measurement is the median of five trials, unless specified otherwise.

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.

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].[7] 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[8] provided by Network Repository [55].[9] The graph represents a mapping of human brain connections. "WebBase" is the webbase-2001 graph from the Laboratory for Web Algorithmics [11, 12].[10] The graph represents the links discovered by a web crawler. Although the original WebBase graph is a directed graph, we change the edges to be undirected and remove self-loop edges so that SCAN can operate on the graph. "Blood vessel" and "cochlea" are weighted graphs from HumanBase [34].[11] Vertices represent genes, edges represent pairs of genes with evidence of a functional relationship in blood vessel tissues or cochlea tissues, and edge weights represent the probability of there being a relationship. For convenience, on the brain, Friendster, blood vessel, and cochlea graphs, we compact vertex IDs so that all IDs are contiguous with no zero-degree vertices.

Neither GS*-Index and ppSCAN run on weighted graphs, so we do not run them on the blood vessel and cochlea graphs. We also only test cosine similarity on the weighted graphs since we did not implement weighted Jaccard similarity for GBBSIndexSCAN.

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

[5] We obtained the GS*-Index code via personal correspondence with its authors.

[6] The ppSCAN code is available at https://github.com/RapidsAtHKUST/ppSCAN/tree/master/pSCAN-refactor.

[7] https://snap.stanford.edu/data/

[8] https://neurodata.io/

[9] http://networkrepository.com/bn-human-Jung2015-M87113878.php

[10] http://law.di.unimi.it/webdata/webbase-2001/

[11] https://hb.flatironinstitute.org/download under the "top edges" column

would fall within clusters in a random graph with the same degree distribution [49]. Specifically, fix some clustering, let $A_{u,v}$ for arbitrary vertices u and v be 1 if u and v are neighbors and be 0 otherwise, and let $\delta_{u,v}$ be 1 if u and v are assigned the same cluster and be 0 otherwise. The modularity is computed as

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

The definition of modularity also easily extends 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 $\mathcal{G}$ be the ground-truth clustering. For integers i in $\{1,2,3,\ldots,|C|\}$ and j in $\{1,2,3,\ldots,|\mathcal{G}|\}$, let $n_{i,j}$ be the number of vertices in both cluster i of C and cluster j of $\mathcal{G}$. Let $n_{i,*}=\sum_{j=1}^{|\mathcal{G}|}n_{i,j}$ and let $n_{*,j}=\sum_{i=1}^{|C|}n_{i,j}$ for each i and j. Then, the ARI between C and $\mathcal{G}$ is

$$\frac{\sum_{i=1}^{|C|}\sum_{j=1}^{|\mathcal{G}|}\binom{n_{i,j}}{2}-\sum_{i=1}^{|C|}\binom{n_{i,*}}{2}\sum_{j=1}^{|\mathcal{G}|}\binom{n_{*,j}}{2}/\binom{n}{2}}{\left(\sum_{i=1}^{|C|}\binom{n_{i,*}}{2}+\sum_{j=1}^{|\mathcal{G}|}\binom{n_{*,j}}{2}\right)/2-\sum_{i=1}^{|C|}\binom{n_{i,*}}{2}\sum_{j=1}^{|\mathcal{G}|}\binom{n_{*,j}}{2}/\binom{n}{2}}.$$

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 is about the same (at most 9% difference for GBBSIndexSCAN on 48 cores), so we do not report it separately. Figure 5 shows the time measurements. GBBSIndexSCAN achieves a parallel self-relative 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 on 48 cores with hyper-threading is 50–151× over GS*-Index. On the two dense graphs with fewer vertices, GBBSIndexSCAN-MM outperforms GBBSIndexSCAN, but it takes too much memory to run on the other graphs.

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 (μ,ε). The plots only consider exact cosine similarity since times are about the same using Jaccard similarity (at most either 10^{-4} absolute difference or 4% difference for GBBSIndexSCAN on 48 cores). Clustering behavior is the same between GBBSIndexSCAN and GBBSIndexSCAN-MM, so we omit times for GBBSIndexSCAN-MM. Figure 6 measures the running times with $\mu=5$ and $\varepsilon\in\{.1,.2,.3,\ldots,.9\}$, and Figure 7 measures the running times with $\varepsilon=0.6$ and $\mu\in\{2^i\mid 1\le i\le 14, 2^i\le \text{max\_degree}\}$.

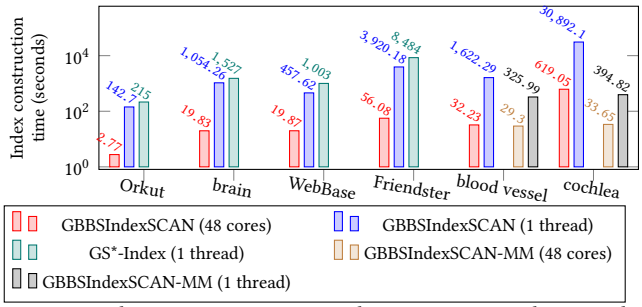

Figure 5: Index construction times with exact cosine similarity as the similarity measure. We only run GBBSIndexSCAN-MM on the blood vessel and cochlea graphs, whose adjacency matrices fit in memory.

GBBSIndexSCAN is faster than ppSCAN and GS*-Index on all tested parameter settings, though this ignores the time that 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 sum of the time measurements for ppSCAN on the nine parameter settings in Figure 6 exceeds the sum of the corresponding 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 48 cores, 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 1.26–12,070×.

7.3.3 Approximate index construction time The third experiment measures the running time of constructing GBBSIndexSCAN with 48 cores 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 trial uses a different pseudorandom seed for the randomness in the approximation schemes. Figure 8 displays the results. 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 ε-similar

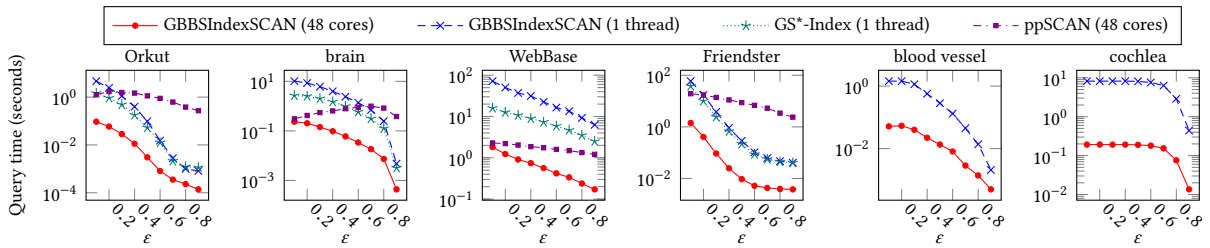

Figure 6: Clustering time with $\mu = 5$ and varying ε using exact cosine similarity as the similarity measure.

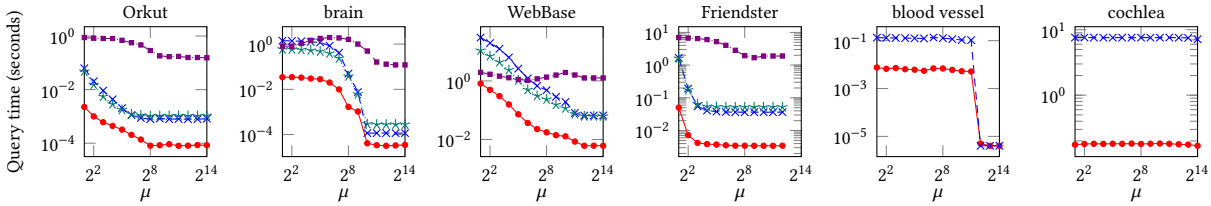

Figure 7: Clustering time with $\varepsilon = 0.6$ and varying μ using exact cosine similarity as the similarity measure.

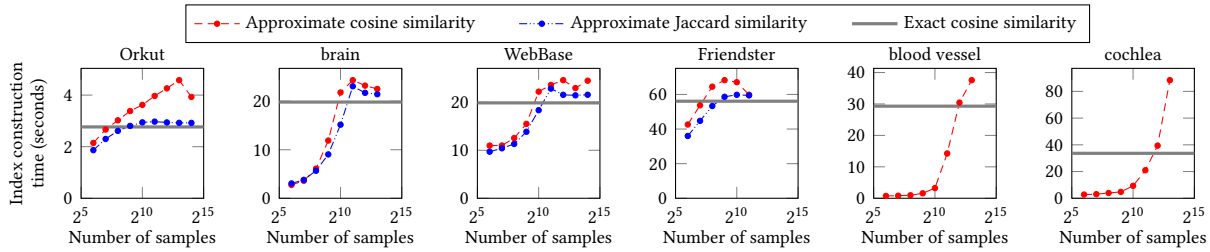

Figure 8: Index construction times for GBBSIndexSCAN (48 cores with hyper-threading) using approximate similarity measures with varying sample sizes.

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 (μ, ε) maximizing modularity from the following set Σ:

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

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 Σ 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 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.

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 modularity and ARI scores indicate that approximating similarities can significantly speed up index construction while still achieving good quality clusterings. The modularity plots in Figure 9 look more favorable than the ARI plots in Figure 10, suggesting that though at low sample sizes the approximate clusters at a particular parameter setting may noticeably differ from the corresponding exact clusters, we are still able to find a good quality clustering by searching over a range of parameter values.

8 Related Work

Xu et al. introduced the original SCAN algorithm [71] and borrowed ideas from the popular spatial clustering algorithm DBSCAN [29]. One major inconvenience of SCAN is the difficulty of choosing its two user-selected parameters, μ and ε. GS*-Index alleviates this issue by creating an index upon which future SCAN queries with

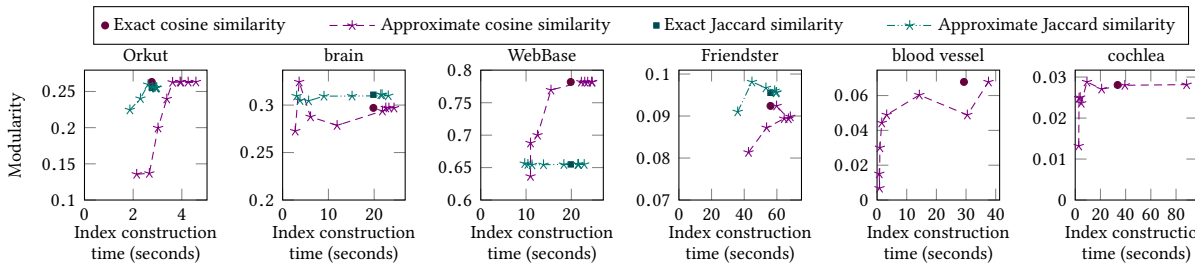

Figure 9: Trade-off curve of approximate similarity index construction time with varying numbers of samples versus the best modularity score found among parameters from Σ (Equation (1)). The index construction times on the horizontal axis come from Figure 8.

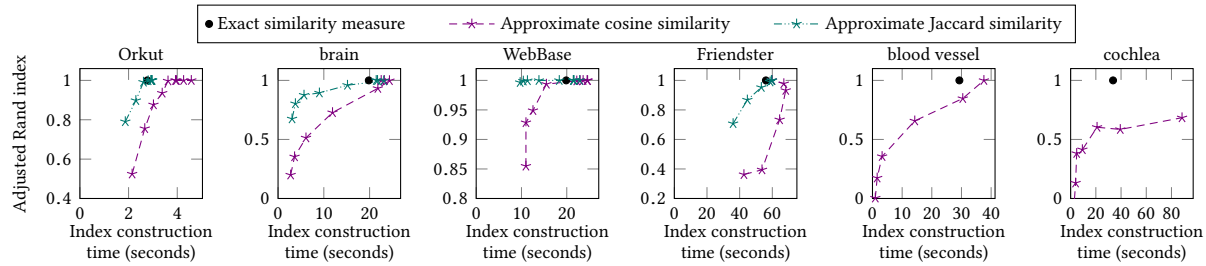

Figure 10: Trade-off curve of approximate similarity index construction time with varying numbers of samples versus the accuracy (measured via adjusted Rand index) of the resulting approximate clustering relative to a "ground-truth" clustering from the corresponding exact similarity index. For each graph, the tested parameters (μ, ε) are the modularity-maximizing parameters from Σ (Equation (1)) relative to the exact similarity measure. The index construction times on the horizontal axis come from Figure 8.

arbitrary parameters are efficient [68]. SCOT [13] and gSkeletonClu [37] also essentially compute indices for SCAN, but only for a fixed μ 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 on 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 parallelizations of SCAN, and SparkSCAN [77] is a Spark parallelization of SCAN. GPUSCAN [64] uses GPUs to speed up SCAN.

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 and have logarithmic span w.h.p. We also present an 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.

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 the speed and clustering quality of SCAN when using other similarity measures. Last, we wish to compare SCAN to other parallel clustering algorithms in quality and speed.

Acknowledgments

We thank Rezaul Chowdhury for suggesting using matrix multiplication on dense graphs. This research was supported by DOE Early Career Award #DE-SC0018947, 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] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. *SIGMOD Record* 28, 2 (1999), 49–60.

[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] Ariful 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).

[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 Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab Mirrokni. 2017. Affinity 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 Biemann. 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 32nd ACM 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 Mervin E. 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, Wenjie 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 Qiong 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. *SIAM J. Comput.* 17, 4 (1988), 770–785.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press.

[23] Erik D. Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. 2006. Correlation clustering in general weighted graphs. *Theoretical Computer Science* 361, 2–3 (2006), 172–187.

[24] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*. 393–404.

[25] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. *Proceedings of the VLDB Endowment* 14, 4 (2020), 653–667.

[26] Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. 2021. Parallel Batch-Dynamic k-Clique Counting. In *Symposium on Algorithmic Principles of Computer Systems*. SIAM, 129–143.

[27] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics*. Article 11, 8 pages.

[28] Yijun Ding, Minjun Chen, Zhichao Liu, Don Ding, Yanbin Ye, Min Zhang, Reagan Kelly, Li Guo, Zhenqiang Su, Stephen C. Harris, Feng Qian, Weigong Ge, Hong Fang, Xiaowei Xu, and Weida Tong. 2012. atBioNet–an integrated network analysis tool for genomics and biomarker discovery. *BMC Genomics* 13, Article 325 (2012).

[29] 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 Second International Conference on Knowledge Discovery and Data Mining*. 226–231.

[30] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3–5 (2010), 75–174.

[31] Hillel Gazit. 1991. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.* 20, 6 (1991), 1046–1067.

[32] Joseph Gil, Yossi Matias, and Uzi Vishkin. 1991. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*. 698–710.

[33] Michelle Girvan and Mark E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826. arXiv:https://www.pnas.org/content/99/12/7821.full.pdf

[34] Casey S. Greene, Arjun Krishnan, Aaron K. Wong, Emanuela Ricciotti, Rene A. Zelaya, Daniel S. Himmelstein, Ran Zhang, Boris M. Hartmann, Elena Zaslavsky, Stuart C. Sealfon, Daniel I. Chasman, Garret A. FitzGerald, Kara Dolinski, Tilo Grosser, and Troyanskaya Olga G. 2015. Understanding multicellular function and disease with human tissue-specific networks. *Nature Genetics* 47 (2015), 569–576.

[35] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[36] Jianbin Huang, Heli Sun, Jiawei Han, Hongbo Deng, Yizhou Sun, and Yaguang Liu. 2010. SHRINK: A Structural Clustering Algorithm for Detecting Hierarchical Communities in Networks. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. 219–228.

[37] Jianbin Huang, Heli Sun, Qinbao Song, Hongbo Deng, and Jiawei Han. 2013. Revealing Density-Based Clustering Structure from the Core-Connected Tree of a Network. *IEEE Transactions on Knowledge and Data Engineering* 25, 8 (2013), 1876–1889.

[38] Lawrence Hubert and Phipps Arabie. 1985. Comparing Partitions. *Journal of Classification* 2 (1985), 193–218.

[39] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

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

[41] Ping Li, Art Owen, and Cun-Hui Zhang. 2012. One Permutation Hashing. In *Advances in Neural Information Processing Systems 25*. 3113–3121.

[42] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping Community Detection Using the Link-Space Transformation. In *IEEE 30th International Conference on Data Engineering*. 292–303.

[43] Cindy Xide Lin, Yintao Yu, Jiawei Han, and Bing Liu. 2010. Hierarchical Web-page Clustering via In-page and Cross-page Link Structures. In *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. 222–229.

[44] Zhichao Liu, Qiang Shi, Don Ding, Reagan Kelly, Hong Fang, and Weida Tong. 2011. Translating Clinical Findings into Knowledge in Drug Safety Evaluation - Drug Induced Liver Injury Prediction System (DILIps). *PLOS Computational Biology* 7, 12 (2011).

[45] Son T. Mai, Sihem Amer-Yahia, Ira Assent, Mathias Skovgaard Birk, Martin Storgaard Dieu, Jon Jacobsen, and Jesper M. Kristensen. 2019. Scalable Interactive Dynamic Graph Clustering on Multicore CPUs. *IEEE Transactions on Knowledge and Data Engineering* 31, 7 (2019), 1239–1252.

[46] Venkata-Swamy Martha, Zhichao Liu, Li Guo, Zhenqiang Su, Yanbin Ye, Hong Fang, Don Ding, Weida Tong, and Xiaowei Xu. 2011. Constructing a robust protein-protein interaction network by integrating multiple public databases. In *BMC Bioinformatics*, Vol. 12. Article S7.

[47] Mutlu Mete, Fusheng Tang, Xiaowei Xu, and Nurcan Yuruk. 2008. A structural approach for finding functional modules from large biological networks. In *BMC Bioinformatics*, Vol. 9. Article S19.

[48] Mark E. J. Newman. 2004. Analysis of weighted networks. *Physical Review E* 70 (2004), 056131. Issue 5.

[49] Mark E. J. Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69 (2004), 026113. Issue 2.

[50] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2015. Parallel Correlation Clustering on Big Graphs. In *Advances in Neural Information Processing Systems*. 82–90.

[51] Symeon Papadopoulos, Yiannis Kompatsiaris, and Athena Vakali. 2009. Leveraging Collective Intelligence through Community Detection in Tag Networks. In *Proceedings of Workshop on Collective Knowledge Capturing and Representation*.

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

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

[54] Rajeev Raman. 1990. The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting. In *Foundations of Software Technology and Theoretical Computer Science*. 161–175.

[55] 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.

[56] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer Science Review* 1, 1 (2007), 27–64.

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

[58] Manos Schinas, Symeon Papadopoulos, Georgios Petkos, Yiannis Kompatsiaris, 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.

[59] Hiroaki Shiokawa, Yasuhiro Fujiwara, 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.

[60] Anshumali Shrivastava and Ping Li. 2014. In Defense of MinHash Over SimHash. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. 886–894.

[61] Julian Shun and Guy Edward Blelloch. 2014. Phase-Concurrent 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 Algorithm for Networks. *IEEE Transactions on Parallel and Distributed Systems* 26, 12 (2015), 3381–3393.

[65] Tomokatsu Takahashi, Hiroaki Shiokawa, 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*. 626–641.

[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* (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 13th 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. AHSCAN: 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.