Parallel index-based structural graph clustering and its approximations

by

Tom Tseng

B.S., Carnegie Mellon University (2018)

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2020

Certified by . . . .
Julian Shun
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Parallel index-based structural graph clustering and its approximations

by

Tom Tseng

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2020, in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

SCAN (structural clustering algorithm for networks) is a well-known approach for graph clustering. Sequential versions of SCAN are prohibitively slow on large graphs, however. Existing parallel versions of SCAN, on the other hand, can cluster graphs relatively quickly on a particular setting of SCAN parameters, but do not effectively share work among queries on different parameter settings. Because users of SCAN need to test several parameter settings in order to find a good clustering, it can be worthwhile to precompute an index to speed up later queries. To that end, this thesis presents a parallelization of GS*-Index, an existing index-based SCAN algorithm. The parallelized algorithm is work-efficient and achieves logarithmic span for both constructing the index and running clustering queries.

We describe an implementation of our algorithm and test it on several real-world large graphs, with the largest graph having 1.8 billion edges. On a machine with 48 cores and 2-way hyper-threading, our parallel index construction achieves 50–151× speedup over the construction of GS*-Index. In fact, our index construction algorithm is faster than GS*-Index even when running our algorithm sequentially. Our parallel index query implementation achieves 5–32× speedup over queries on GS*-Index across a range of SCAN parameter values, and our implementation is also faster than ppSCAN, the fastest existing parallel SCAN algorithm, on all tested parameter values.

We also explore how locality-sensitive hashing can speed up index construction by approximating the similarity scores between vertices, the computation of which is the most time-consuming aspect of SCAN. Our experiments show that this technique can achieve meaningful speedups on denser graphs without large sacrifices in clustering quality.

Thesis Supervisor: Julian Shun
Title: Assistant Professor

Acknowledgments

This thesis, like all of my work, is a collective effort. I would be unable to write this thesis if it were not for the help, direct and indirect, of others.

I thank my advisor, Professor Julian Shun, for his generous support and patience starting from before I even moved to campus. He provided a lot of direction in this first research project of my graduate career.

I also thank my past mentors and close collaborators. My undergraduate thesis advisor, Professor Guy Blelloch, introduced me to the area of parallel algorithms and taught me much of what I needed to make my first contributions to the field. Before that, Professor Anil Ada introduced me to theoretical computer science and advised me when I first chose to pursue research. Laxman Dhulipala worked closely with me throughout the research that led to my undergraduate thesis, and I continue to collaborate with him to this day, including on the project covered by this thesis. His encouragement and advice when I was considering applying to graduate school during my undergraduate years benefited me as well.

I thank my many friends from my hometown, from college, from my time in industry, and from graduate school. They are sources of inspiration, emotional relief, intellectual growth, and joyful memories. Without my friends, my happiness and my moral character would surely both be worse off.

Finally, I thank my parents for the tremendous sacrifices that they made for my brothers and me. The greatest gift I have ever received is their endless and compassionate support.

Contents

Chapter 1

Introduction

A crucial technique in understanding the structure of data is to organize it into meaningful groupings. When the data takes the form of a graph, the problem usually becomes a *graph clustering* problem in which the goal is to partition the vertices of the graph into clusters so that "closely related" vertices fall in the same cluster. In particular, a good graph clustering usually has many edges that fall within clusters but relatively few edges that connect different clusters. Graph clustering is a popular problem with a wide range of applications, including social and biological network analysis [26], recommendation systems [3], image segmentation [58], natural language processing [4], and load balancing in distributed systems [2].

One popular approach to graph clustering is *structural clustering*, which Xu et al. first introduced via the Structural Clustering Algorithm for Networks (SCAN) [61]. In structural clustering, the *similarity* of adjacent vertices depends on the number of shared neighbors between the vertices. The approach is simple, and it is unique in that it also finds *hub* vertices that connect different clusters as well as *outlier* vertices that do not have strong ties to any cluster. Researchers have used SCAN for finding meaningful clusters in biological data [41, 40, 38, 21] and web data [44, 45, 46, 37, 50, 51].

SCAN as Xu et al. originally described it suffers, however, from two issues: (1) the large computational cost of sequentially computing the 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), there are variants that exploit parallelism [14, 65, 56, 66, 57, 13, 39], and there are variants that introduce algorithmic optimizations like pruning away unnecessary similarity computations [52, 11, 13]. To alleviate issue (2), there are variants that precompute an index from which computing the clusterings resulting from a range of parameter values is fast [8, 30, 59]. To run structural clustering effectively on large graphs, SCAN-based algorithms should address both issues, which existing algorithms fail to do.

This thesis addresses the aforementioned issues by presenting a parallel index-based SCAN algorithm based on the sequential GS*-Index SCAN algorithm [59]. Our algorithm achieves the same work bounds as GS*-Index in expectation, and it is highly parallel in the sense that it achieves logarithmic span with high probability.[1] We also describe and provide the code of an implementation of our algorithm that runs quickly in practice. In our experiments on a machine with 48 cores and 2-way hyper-threading against several large real-world graphs, our index construction algorithm achieves 50–151× speedup over the construction of GS*-Index. In fact, our index construction algorithm is faster than GS*-Index even when we run our algorithm sequentially with a single thread. Our parallel index query implementation achieves 5–32× speedup over queries on GS*-Index across a range of SCAN parameter values. Our implementation also achieves faster query times on all tested parameter values compared to ppSCAN [13], the fastest existing parallel SCAN algorithm.

To further address issue (1), we also explore using locality-sensitive hashing to speed up similarity computation by calculating approximate similarities. Our theoretical analysis and experiments show that on dense graphs, using locality-sensitive hashing can speed up our index construction by considerable amounts without large sacrifices in clustering quality.

In summary, the contributions of this thesis are as follows:

- We present a parallel index-based SCAN algorithm that is work-efficient in

[1]We use *with high probability* 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.

expectation and has logarithmic span with high probability.

- We introduce and theoretically analyze the idea of combining locality-sensitive hashing with SCAN to achieve faster, approximate results on dense graphs.
- We present experiments that demonstrate that our implementation of our algorithm outperforms other existing SCAN algorithms and that confirm that locality-sensitive hashing can provide running time improvements on denser graphs while still finding clusterings with good quality.
- We release the implementation of our algorithm. [2]

[2]Code: `https://github.com/ParAlg/gbbs/tree/ed48046f6a20c378ae2e54586d15722cea0a3d75/benchmarks/SCAN/IndexBased`

Chapter 2

Preliminaries

This chapter provides background definitions, concepts, and notation that subsequent chapters use.

2.1 Set similarity

2.1.1 Similarity measures

Two common measures for the similarity of two finite sets A and B are Jaccard similarity and cosine similarity:

$$\begin{aligned}\mathrm{JaccardSim}(A,B) &= \frac{|A \cap B|}{|A \cup B|},\\ \mathrm{CosineSim}(A,B) &= \frac{|A \cap B|}{\sqrt{|A|}\sqrt{|B|}}.\end{aligned}$$

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

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

Suppose that sets A and B have elements from the universe $\{1, 2, 3, \ldots, d\}$ for some

$d \in \mathbb{N}$. Then the definition of the cosine similarity between two sets comes from representing each set as a bit vector in $\mathbb{R}^d$ and then computing the cosine similarity between the two bit vectors. Furthermore, using a similar vector representation, cosine similarity extends in a straightforward way to sets that assign a weight to each of their elements. If sets A and B have weight functions w_A and w_B mapping elements to real number weights, then

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

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

2.1.2 Locality-sensitive hashing

Suppose that there is a collection of large sets with elements in the universe $U = \{1, \ldots, d\}$ for some $d \in \mathbb{N}$. Computing pairwise similarities scores among these sets may be expensive because each set is large. *Locality-sensitive hashing* is a technique that can speed up this computation at the expense of accuracy. The idea is to devise a hash function family that maps similar sets to similar, smaller hash values or *sketches*. Then we can sketch each set and estimate similarities by operating on the sketches instead of on the original sets.

A well-known locality-sensitive hash scheme for estimating Jaccard similarity, for instance, is MinHash [10]. 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 non-empty sets A and B, the probability that the sketches of A and B are equal is exactly $\text{JaccardSim}(A, B)$. To increase the precision, we fix some number of samples $k \in \mathbb{N}$ and perform this process k times independently to get k-length sketches. Measuring the proportion of matching coordinates between two sketches gives an estimate of the Jaccard similarity between the two corresponding sets. Increasing the number of samples k reduces variance at the cost of increased computational effort.

There are variants of MinHash that strive to be more computationally efficient, such as k-partition MinHash [35].

SimHash [12] is a well-known locality-sensitive hash scheme for estimating the angle between two vectors. Hence, it may also estimate the cosine similarity between vectors, though this estimate is biased. The idea behind SimHash is to consider drawing a vector v in $\mathbb{R}^d$ by drawing each coordinate independently from the standard normal distribution. This vector v has uniformly random direction. Take the sketch of a vector u to be $\operatorname{sign}(u \cdot v)$. Consider any non-zero vectors a and b, and let $\theta \in [0, \pi]$ denote the angle between them in radians. 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 $\operatorname{sign}(a \cdot v) \neq \operatorname{sign}(b \cdot v)$. Like with MinHash, for more precision, we can fix $k \in \mathbb{N}$ and repeat this process k times to get k-length sketches. From the k-length sketches, we estimate θ by counting the number of differing entries between the sketches of a and b and multiplying that number by π/k. Having an estimate of θ then gives an estimate for $\cos(\theta) = \operatorname{CosineSim}(u, v)$.

2.2 Graph clustering

2.2.1 Graphs

We denote an unweighted, undirected graph G by $G = (V, E)$ where V is the set of graph vertices and $E \subseteq \{\{u, v\} : u, v \in V\}$ is the set of graph edges. We denote a weighted graph G by $G = (V, E, w)$ where $w : E \to \mathbb{R}$ is a function that maps edges to weight values. Following common convention, we often 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 all vertices u 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 size of the vertex's neighborhood, though when the exact value is important, we will write either $|N(v)|$ or $\left|\overline{N}(v)\right|$ to avoid ambiguity.

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

The arboricity α of a 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 the arboricity 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 [15].

The graph representation that this thesis assumes is the adjacency list, which lists of the neighborhoods of each vertex in the represented graph. We only consider simple graphs, which are graphs without multiple edges between any particular pair of vertices and without self-loop edges. We index vertices using the integers in the range $[1, n]$.

2.2.2 SCAN definitions

SCAN [61] is a graph clustering algorithm. In this thesis, we assume that any graphs to be clustered are undirected since SCAN is only intended to run on undirected graphs.

The typical problem formulation for graph clustering is that the goal is to output a partition (or *clustering*) of the vertices of the input graph such that each cluster in the partition has many edges within the cluster and such that there are few edges between clusters. How exactly to measure the quality of a clustering is unclear and depends on the application domain. Section 2.2.4 lists a few clustering quality measures.

The output of SCAN diverges slightly from this description of clustering; SCAN may leave some vertices unclustered. Unclustered vertices are further separated into *hubs* and *outliers*. Hubs are vertices that neighbor several clusters but do not belong to any, and outliers are unclustered vertices that neighbor at most one cluster.

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

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

This definition of similarity score is arbitrary, however; other papers consider using Jaccard similarity, Dice similarity, or weighted cosine similarity for the similarity function [29, 30, 11, 39].

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* of a vertex v is the set of its ε-similar neighbors:

$$N_\varepsilon(v) = \left\{u \in \overline{N}(v) \mid \sigma(u,v) \geq \varepsilon\right\}.$$

The *core* vertices are the vertices whose ε-neighborhood contains at least μ neighbors:

$$\text{vertex } v \text{ is a core } \iff |N_\varepsilon(v)| \geq \mu.$$

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 in the sense that 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 in the sense that for every core vertex v in the cluster, all vertices that are structurally reachable from v are also in the cluster.

Some non-core vertices may be *unclustered* and not belong to any cluster. These vertices are further divided into *hubs*, which are unclustered vertices that neighbor at least two different clusters, and *outliers*, which are all the remaining vertices.

An issue with the definition of SCAN clusters is that the *border* vertices, which are the clustered non-core vertices, may belong to several distinct clusters according to the definition. The original SCAN algorithm assigns these ambiguous border vertices arbitrarily to any of its possible clusters.

Computing similarity scores takes $O(\alpha m)$ time if implemented carefully. 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. There are $O(\alpha m)$-time triangle counting algorithms that can find these per-edge triangle counts. After computing similarities, SCAN finds clusters by performing a modified breadth-first search, which takes $O(n + m)$ time.

2.2.3 Index-based SCAN: GS*-Index

GS*-Index [59] improves on SCAN by pre-computing 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 $C \subseteq V$, define $E_{C,\varepsilon}$ to be the set of ε-similar edges in the subgraph induced by C. Then the time to compute the clustering $\mathcal{C}$ for parameters μ and ε is $O\left(\sum_{C \in \mathcal{C}} |E_{C,\varepsilon}|\right)$. 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 of vertices. The neighbor order is simply the adjacency list of the graph with each neighbor list sorted by descending similarity. The core order is an array where the μ-th entry, $\mathcal{CO}[\mu]$, for any μ is a list of vertices with degree (relative each vertex's closed neighborhood) at least μ. These are the vertices v for which there is some threshold $\varepsilon_{\text{threshold}}(\mu, v)$ such that for all $\varepsilon \leq \varepsilon_{\text{threshold}}(\mu, v)$, the vertex is a core vertex under parameters μ and ε. This value $\varepsilon_{\text{threshold}}(\mu, v)$ for a vertex v is the μ-th entry of $\mathcal{NO}[v]$. Each list $\mathcal{CO}[\mu]$ is sorted by descending threshold values

$\varepsilon_{\text{threshold}}(\mu, \cdot) = \mathcal{NO}[\cdot][\mu]$.

To find the clustering resulting from SCAN parameters μ and ε, we perform a breadth-first search 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 $\mathcal{CO}[\mu]$ and the ε-similar edges are prefixes of $\mathcal{NO}[\cdot]$. This breadth-first search reveals all the SCAN clusters in the graph.

2.2.4 Clustering quality measures

One of the most popular graph clustering quality metrics that relies only the structure of the graph is the *modularity* [43]. The modularity is the proportion of edges that fall within clusters in the clustering minus the expected number of edges that would fall within clusters in a random graph with the same degree distribution. More explicitly, fix some clustering and let $\delta_{u,v}$ for arbitrary vertices u and v be 1 if u and v are assigned the same cluster and be 0 otherwise. The modularity of the clustering is

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

The definition of modularity also easily extends to apply to weighted graphs [42]. 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. One well-known metric for evaluating this similarity is the *adjusted Rand index* (ARI) [31]. 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. To define the formula for ARI, let $\mathcal{C}$ be the proposed clustering on the set of n vertices V and let $\mathcal{G}$ be the ground-truth clustering. For integers i in $\{1, 2, 3, \ldots, |\mathcal{C}|\}$ and j in $\{1, 2, 3, \ldots, |\mathcal{G}|\}$, let $n_{i,j}$ be the number of vertices in both cluster i of $\mathcal{C}$ and cluster j of $\mathcal{G}$. Let $n_{i,*} = \sum_{j=1}^{|\mathcal{G}|} n_{i,j}$ and let $n_{*,j} = \sum_{i=1}^{|\mathcal{C}|} n_{i,j}$.

Then the ARI between $\mathcal{C}$ and $\mathcal{G}$ is

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

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 somehow "worse than random."

2.3 Parallelism

2.3.1 Parallel programming model

We design our algorithm to be run on a multicore shared-memory machine. Although distributed systems may scale more effectively for extremely large inputs, readily available modern shared-memory machines are already able to operate on graphs with hundreds of billions of edges [18]. Shared-memory systems are fast due to low communication costs and tend to be 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. The compare-and-swap is an atomic operation takes the form

$$\textsc{CompareAndSwap}(\textit{memory\_address}, \textit{expected\_value}, \textit{new\_value})$$

which assigns *new_value* to the memory address if the memory address holds the value *expected_value* and otherwise leaves the value at the memory address unchanged. We assume that atomic additions and compare-and-swaps each take $O(1)$ work.

We describe the time complexity of a program execution by its *work*, which is the total number of instructions executed, and its *span* or depth, which is the length of the longest sequential critical path of instructions in the execution [32]. Ideally, a parallel algorithm should have much smaller span than work and should have work close to the work of the best sequential algorithm for the same problem. A parallel algorithm whose work asymptotically matches the work of the most efficient known sequential algorithm is *work-efficient*.

2.3.2 Parallel primitives

This thesis makes use of many existing parallel algorithms, which we list below.

Hash tables Gil et al. present a hash table which supports inserting k elements in $O(k)$ work and $O(\log^* k)$ span with both bounds being with high probability. Looking up an element takes $O(1)$ work [25].

Various array operations 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, reducing an array with the binary operation that takes the maximum of two elements produces the maximum element in the 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 [32, 5]. 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. Using a parallel hash table, this operation runs in $O(n)$ work and $O(\log^* n)$ span with both bounds being with high probability.

Sorting Cole presents a parallel merge sort that sorts n elements in $O(n \log n)$ work and $O(\log n)$ span [16].

Graph connectivity Gazit describes a algorithm for computing graph connectivity with $O(n + m)$ expected work and $O(\log n)$ span with high probability [24].

Chapter 3

Algorithm

3.1 Basic description

This section describes work-efficient, logarithmic-span parallel algorithms for constructing the same SCAN index that GS*-Index constructs and for retrieving clusters from the index. Our algorithm is a mostly straightforward parallelization of the original sequential GS*-Index algorithms.

For describing the algorithms in this section, we assume the existence of basic utility functions as well as functions implementing the primitives discussed in section 2.3.2. The AllocateArray(*size*) function allocates an array that holds *size* elements. The ArrayLength($\cdot$) function returns the size of the input array. The MakeHashMap($\cdot$) function makes a hash table with the input argument specifying the key-value elements in the table. The MakeHashSet($\cdot$) function also makes a hash table, but the table contains 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.

3.1.1 Index construction

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 Helper function for computing cosine similarities in algorithm 2.

Output: An n-length array of $\sqrt{\sum_{u \in \overline{N}(v)} w(u,v)}$ for each vertex v.

```
1: procedure ComputeNorms($G = (V, E, w)$)
2:     $norms \leftarrow$ AllocateArray($n$)
3:     for $v \in V$ do in parallel
4:         $weights\_squared \leftarrow$ AllocateArray($|\overline{N}(v)|$)
5:         for $i \in \{1, 2, 3, \dots, |\overline{N}(v)|\}$ do in parallel
6:             $u \leftarrow i$-th element in $\overline{N}(v)$
7:             $weights\_squared[i] \leftarrow w(u,v)^2$
8:         $norms[v] \leftarrow \sqrt{\text{Sum}(weights\_squared)}$
9:     return $norms$
```

Algorithm 2 gives pseudocode for computing similarities, and it calls algorithm 1 as a helper function. The logic is the same as that of a known hash-based parallel algorithm for triangle counting [55]. The algorithm creates a hash set for each vertex's neighborhood (line 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

Algorithm 2 Algorithm for computing the cosine similarity of each edge in a weighted graph.

```
Output: An $m$-length array of the similarity score of each edge.
 1: procedure ComputeSimilarities($G = (V, E, w)$)
 2:     $norms \leftarrow$ ComputeNorms($G$)
 3:     $similarities \leftarrow$ AllocateArray($m$)
                ▷ For clarity, we will index into similarities with elements 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 $|\overline{N}(u)| \leq |\overline{N}(v)|$.)
 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 $similarities$
```

all the shared neighbors between u and v, which allows 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 of the algorithm is $O\left(\sum_{\{u,v\}\in E} \min\{|\overline{N}(u)|, |\overline{N}(v)|\}\right)$ in expectation. This value is bounded by $O(\alpha m)$ [15]. The span is $O(\log n)$ with high probability.

Neighbor order and core order

After computing all similarity values, constructing the neighbor order and core order simply becomes an act of sorting several arrays, as algorithm 3 shows. The logic follows directly from the definition of the neighbor order and core order from section 2.2.3.

With a work-efficient sorting algorithm, the work analysis is the same as the original analysis for GS*-Index. This gives a work bound of $O(m \log n)$ for constructing the orders. The span is $O(\log n)$.

Algorithm 3 Algorithms for computing the neighbor order and core order.

```
1: procedure ConstructNeighborOrder($G = (V, E, w)$, 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 descending $similarities[\{u, v\}]$ value.
6:     return $\mathcal{NO}$
1: procedure ConstructCoreOrder($G = (V, E, w), \mathcal{NO}$)
2:     Sort $v$ in $V$ by descending degree.
3:     $max\_degree \leftarrow \max_{v \in V} |\overline{N}(v)|$
4:     $\mathcal{CO} \leftarrow$ AllocateArray($max\_degree$)
5:     for $\mu = \{2, 3, 4, \ldots, max\_degree\}$ do in parallel
6:         $\mathcal{CO}[\mu] \leftarrow \{v \in V \mid |\overline{N}(v)| \geq \mu\}$
                       ▷ Find $\{v \in V \mid |\overline{N}(v)| \geq \mu\}$ by binary search on sorted $V$.
7:         Sort $v$ in $\mathcal{CO}[\mu]$ by descending $\mathcal{NO}[v][\mu]$ value.
8:     return $\mathcal{CO}[\mu]$
```

Summing those bounds with the work and span of computing similarities gives the following theorem.

Theorem 3.1.1. *Fix an undirected graph and let α be the arboricity of the graph. The parallel SCAN index construction algorithm using cosine similarity or Jaccard similarity as the similarity measure runs in $O((\alpha + \log n)m)$ expected work and $O(\log n)$ span with high probability on the graph.*

Therefore, the parallel index construction algorithm is work-efficient in expectation relative to the original sequential algorithm for GS*-Index.

3.1.2 Querying for clusters

Algorithm 6 provides pseudocode for extracting a clustering with arbitrary parameters from the index. Algorithm 4 and algorithm 5 are subroutines for algorithm 6.

To retrieve the clustering with parameters μ and ε, the algorithm performs a binary search on $\mathcal{CO}[\mu]$ to find all core vertices (algorithm 4 line 9) and then performs binary searches on $\mathcal{NO}[v]$ for each core vertex v to find all ε-similar edges incident on core vertices (algorithm 6 line 4). For each of these prefixes of $\mathcal{NO}[v]$, the algorithm also creates a copy with all non-core neighbors filtered away (algorithm 6 line 5).

Algorithm 4 Helper function for finding core vertices under a particular setting of SCAN parameters.

Output: An array of core vertices under SCAN parameters μ and ε.

```
1: procedure GetCores($\mu, \varepsilon, \mathcal{NO}, \mathcal{CO}$)
2:     $max\_degree \leftarrow$ ArrayLength($\mathcal{CO}$)
3:     if $\mu \leq 1$ then                                   ▷ All vertices are cores.
4:         $num\_vertices \leftarrow$ ArrayLength($\mathcal{NO}$)
5:         return $\{1, 2, 3, \ldots, num\_vertices\}$
6:     else if $\mu > max\_degree$ then                       ▷ No vertices are cores.
7:         return $\{\}$
8:     else
9:         return $\{v \in \mathcal{CO}[\mu] \mid \mathcal{NO}[v][\mu] \geq \varepsilon\}$
               ▷ Find cores $\{v \in \mathcal{CO}[\mu] \mid \mathcal{NO}[v][\mu] \geq \varepsilon\}$ by binary search on $\mathcal{CO}[\mu]$.
```

Algorithm 5 Helper function for assigning border non-core vertices to clusters after clustering all the core vertices.

```
1: procedure AssignNonCores($similar\_edges$, $cores\_set$, $core\_clustering$)
2:     $subgraph\_vertices \leftarrow$ RemoveDuplicates($v \mid \{u, v\} \in similar\_edges$)
3:     $subgraph\_non\_cores \leftarrow$                                ▷ Filter
           $\{v \in subgraph\_vertices \mid v \notin cores\_set\}$
4:     $non\_cores\_count \leftarrow$ ArrayLength($subgraph\_non\_cores$)
5:     $non\_core\_assignments \leftarrow$ AllocateArray($non\_cores\_count$)
6:     $non\_core\_indices \leftarrow$ MakeHashMap($subgraph\_non\_cores[i] \mapsto i$)
7:     for $i \in \{1, 2, 3, \ldots, non\_cores\_count\}$ do in parallel
8:         $non\_core\_assignments[i] = null$
9:     for $\{u, v\} \in similar\_edges \wedge (u \notin cores\_set \vee v \notin cores\_set)$ do in parallel
10:        (Without loss of generality, let $v \notin cores\_set$. Then $u \in cores\_set$.)
11:        $address \leftarrow \&(non\_core\_assignments[non\_core\_indices[v]])$
12:        CompareAndSwap($address, null, u$)
13:    Assign each vertex $v$ in $subgraph\_non\_cores$ to the cluster
           that vertex $non\_core\_assignments[non\_core\_indices[v]]$ is
           in and return the clustering. (Full details omitted for brevity.)
```

Algorithm 6 Algorithm for finding the SCAN clustering with parameters μ and ε from the index.

```
1: procedure CLUSTER($\mu, \varepsilon, \mathcal{NO}, \mathcal{CO}$, similarities)
2:     cores ← GETCORES($\mu, \varepsilon, \mathcal{NO}, \mathcal{CO}$)
3:     cores_set ← MAKEHASHSET(cores)
4:     similar_edges ← $\{\{u,v\} \mid u \in$ cores_set $\wedge$ similarities$[\{u,v\}] \geq \varepsilon\}$
               ▷ Get similar_edges by binary search on $\mathcal{NO}[u]$ for each $u$ in cores.
5:     similar_core_edges ←                                                ▷ Filter
           $\{\{u,v\} \in$ similar_edges $\mid u \in$ cores_set $\wedge\ v \in$ cores_set$\}$
6:     core_clusters ←
           Connected components of subgraph induced by similar_core_edges
7:     return ASSIGNNONCORES(similar_edges, cores_set, core_clustering)
```

These filtered prefixes constitute an adjacency list for the subgraph induced by the ε-similar edges on the core vertices. Running a parallel connectivity on this subgraph assigns all core vertices to a cluster (algorithm 6 line 6). Finally, the algorithm uses compare-and-swap to assign border non-core vertices to the same cluster as an arbitrary ε-similar core (algorithm 5).

To get the best running time bounds, each binary search should actually start with a doubling search. For example, when using binary search on $\mathcal{CO}[\mu]$ to find all core vertices, we first search for the minimum $i \in \mathbb{N}$ such that entry $\mathcal{CO}[\mu][2^i]$ does not satisfy the predicate that $\mathcal{NO}[\cdot][\mu] \geq \varepsilon$ and then perform a traditional binary search on the first 2^i entries of $\mathcal{CO}[\mu]$.

It is straightforward to derive the running time bounds in the following theorem.

Theorem 3.1.2. *Suppose the clustering algorithm, algorithm 6, runs and returns a collection of clusters $\mathcal{C}$. For a cluster of vertices $C \in \mathcal{C}$, define $E_{C,\varepsilon}$ the same way that section 2.2.3 defines it. Define*

$$Z = \sum_{C \in \mathcal{C}} \left|E_{C,\varepsilon}\right| \in O(m).$$

Then the run of the clustering algorithm had $O(Z)$ expected work and $O(\log Z)$ span with high probability.

3.1.3 Determining hubs and outliers

After finding a clustering, it is easy to determine whether unclustered vertices are hubs or outliers. First, we construct a hash table that maps clustered vertices to an ID for their cluster. Then, 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. After constructing the hash table, it takes $O(|N(v)|)$ work and $O(\log|N(v)|)$ span to determine whether v is a hub or outlier.

3.2 Approximating similarities

After constructing the index, querying for a clustering is fast. Index construction itself, though, may be expensive, particularly in computing all edge similarities (algorithm 2). One unexplored technique for speeding up edge similarity computation for SCAN is to use locality-sensitive hashing to approximate similarities.

For example, to use SimHash to approximate cosine similarities, we fix some 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 [9] given a source of uniform random numbers. We then use these normally distributed random numbers to construct a k-sample sketch of $\overline{N}(v)$ for each vertex v. This sketching takes $O(km)$ work and $O(\log n)$ span by using the reduce operation to compute inner products. Now we can compute the similarity between any pair of 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 an asymptotic improvement over 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. Feeding these similarities into algorithm 3 constructs a SCAN index with the following running time bounds.

Theorem 3.2.1. *Fix an undirected graph. The parallel SCAN index construction* *algorithm using k-sample MinHash or SimHash to compute approximate similarities runs in $O((k + \log n)m)$ work and $O(\log n + \log k)$ span on the graph.*

We can also make some theoretical statements about what kind of clustering results we get from using these approximate similarities. In particular, suppose we fix some $\varepsilon \in [0, 1]$ and $\delta \in (0, 1)$. Notice that the SCAN clustering with parameters ε and arbitrary μ only cares about whether similarities fall above or below ε. If the number of samples is sufficiently high, then with high probability, all edges with exact similarities below $\varepsilon - \delta$ will have approximate similarities below ε, and all edges with exact similarities above $\varepsilon + \delta$ will have approximate similarities above ε. In other words, all edges outside the similarity range $\varepsilon \pm \delta$ will be "correctly classified" as above or below the threshold ε by the approximate similarities.

Theorem 3.2.2. *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*

$$k \geq \frac{\pi^2 \ln(nm)}{2\delta^2},$$

and suppose we use SimHash with k samples to compute approximate cosine similarity scores for every edge in G . Then with high probability, all edges with exact cosine similarities outside the interval $(\varepsilon - \delta, \varepsilon + \sqrt{1 - \varepsilon^2}\delta)$ will be correctly classified by the approximate cosine similarities as above or below the threshold ε.

Proof. Consider an arbitrary edge $\{u, v\} \in E$ such that the exact cosine similarity of the edge is 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 with high probability.

Let θ be the angle between the vectors corresponding to $\overline{N}(u)$ and $\overline{N}(v)$. Since all edge weights are non-negative, this angle is in the range $[0, \pi/2]$. Recall from section 2.1.2 that SimHash estimates the angle between two vectors by counting the number of entries with differing signs in the sketches of the vectors and multiplying

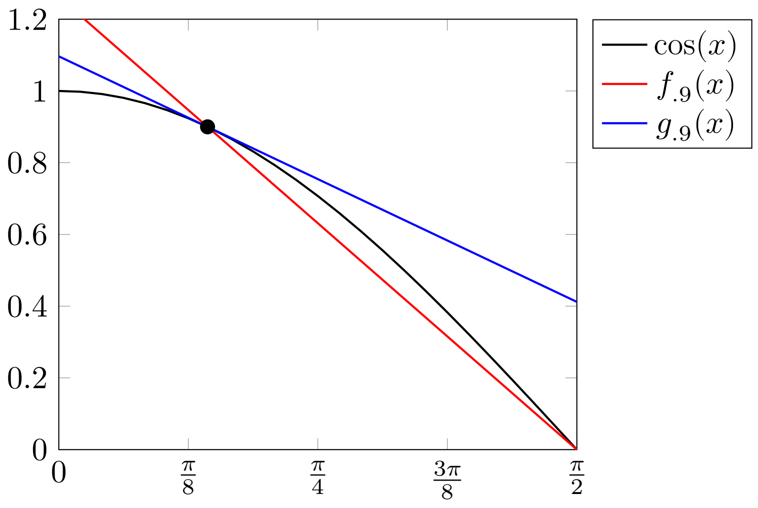

Figure 3-1: Plot of the SimHash approximation lower and upper bound functions on cosine for $\varepsilon = 0.9$.

that number by π/k. Also recall that the cosine similarity estimate is the cosine of this angle. The angle estimate is a random variable X distributed according to $X \sim \text{Binomial}(k, \theta/\pi) \cdot \pi/k$.

Hoeffding's inequality [28] implies that given arbitrary $\ell \in \mathbb{N}$, $p \in [0,1]$, and $t > 0$, for a binomial random variable $Y \sim \text{Binomial}(\ell, p)$, the probability that $\Pr[Y/\ell \geq p + t]$ and the probability that $\Pr[Y/\ell \leq p - t]$ are both bounded above by $\exp(-2\ell t^2)$.

Using this inequality on X gives that that both $\Pr[X \geq \theta + \delta]$ and $\Pr[X \leq \theta - \delta]$ are bounded above by

$$\exp\left(-\frac{2k\delta^2}{\pi^2}\right) \leq \frac{1}{nm}.$$

Let $\phi = \arccos(\varepsilon)$. Since the cosine function is monotonic on the range $[0, \pi/2]$, the probability bound implies that if $\theta \notin (\phi - \delta, \phi + \delta)$, then the edge is classified correctly with probability at least $1 - 1/(nm)$. Therefore, if the exact cosine similarity $\cos(\theta)$ is outside the range $(\cos(\phi+\delta), \cos(\phi-\delta))$, the edge is classified correctly with probability at least $1 - 1/(nm)$.

It remains to find a lower bound on $\cos(\phi + \delta)$ and an upper bound on $\cos(\phi - \delta)$. For the lower bound, draw a straight line from the point (ϕ, ε) to the point $(\pi/2, 0)$, which gives the line

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

By concavity, $\cos(x) \geq f_\varepsilon(x)$ when $x \in [\phi, \pi/2]$. Now this gives $\cos(\phi + \delta) \geq \varepsilon - \frac{\varepsilon}{(\pi/2 - \arccos(\varepsilon))}\delta$. Plotting the multiplicative factor $\frac{\varepsilon}{(\pi/2 - \arccos(\varepsilon))}$ against varying ε shows that the factor falls in the range $[2/\pi, 1]$, which gives a looser but clearer bound that $\cos(\phi + \delta) \geq \varepsilon - \delta$.

For the upper bound, linearize the cosine function at the input point ϕ to get the line

$$g_\varepsilon(x) = \varepsilon - \sin(\phi)(x - \phi).$$

By concavity, $\cos x \leq g_\varepsilon(x)$ when $x \in [0, \pi/2]$. Substituting the input value $\phi - \delta$ gives that

$$\cos(\phi - \delta) \leq \varepsilon + \sin(\arccos(\varepsilon))\delta = \varepsilon + \sqrt{1 - \varepsilon^2}\delta.$$

Figure 3-1 displays the lower and upper bound functions f_ε and g_ε.

This argument shows that if the exact cosine similarity $\cos(\theta)$ is outside the range $(\varepsilon - \delta, \varepsilon + \sqrt{1 - \varepsilon^2}\delta)$, the edge is classified correctly with probability at least $1 - 1/(nm)$. $\square$

Theorem 3.2.3. *Let $G = (V, E)$ be an undirected graph, let $\varepsilon \in [0, 1]$, and let $\delta \in (0, 1)$. Suppose*

$$k \geq \frac{\ln(nm)}{2\delta^2},$$

and suppose we use standard MinHash with k samples to compute approximate Jaccard similarity scores for every edge in G . Then with high probability, all edges with exact Jaccard similarities outside the interval $(\varepsilon - \delta, \varepsilon + \delta)$ will be correctly classified by the approximate Jaccard similarities as above or below the threshold ε.

Proof. The result follows from applying Hoeffding's inequality like in the proof for theorem 3.2.2. $\square$

These bounds are somewhat underwhelming — the number of samples k needs to be quite high to get reasonable accuracy bounds out of these theorems. However, these are worst-case bounds, and it may still be possible to achieve reasonable clusterings with lower values of k. Chapter 4 explores this further.

Regardless, this approximation strategy only helps for denser graphs with high arboricity and many high-degree vertices. Since k needs to be high to get good accuracy, it will be faster to simply compute exact similarities for low arboricity graphs.

3.3 Implementation

We implement the algorithms described in this chapter to determine whether they perform well in practice. We write our code in C++ within the Graph Based Benchmark Suite (GBBS) framework [18, 20] and add the implementation to the GBBS codebase at `https://github.com/ParAlg/gbbs`. GBBS provides libraries that make it easier implement many classes of parallel graph algorithms. Our implementations use the concurrent hash table implementation [54], parallel sorting algorithms, and various graph processing helper functions that GBBS provides.

Though the algorithms as described in section 3.1 get good theoretical bounds, our actual implementations make several changes for better performance. This section details some of the more significant changes.

3.3.1 Computing similarities

We implement similarity computation for both cosine similarity and Jaccard similarity.

Experiments by Shun and Tangwongsan [55] suggest that the hash-based approach to triangle counting or computing similarities in section 3.1.1 incurs a lot of cache misses and that a “merge-based” approach may be faster in comparison even though it increases the asymptotic work bound from $O(m\alpha)$ to $O(m^{3/2})$. Our implementation imitates 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.

The merge logic between two neighbor lists follows the logic of the implementation already in GBBS: if both neighbor lists are small, we iterate across 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 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.

3.3.2 Querying for clusters

Most work-efficient parallel connectivity algorithms are complicated and do not have readily available implementations. Instead, in our implementation for querying the index for clusters, we find the connected components on the core vertices by using a concurrent union-find implementation recently added to the GBBS codebase [19]. Instead of getting a list of connected components, we populate an n-length array where each entry is the cluster ID of the corresponding vertex. This format makes more sense for union-find and simplifies the logic for AssignNonCores (algorithm 5) by changing AssignNonCores to skip the preprocessing in lines 2–8 and instead compare-and-swap directly into the cluster ID array.

3.3.3 Approximate similarities

We implement similarity approximation logic using both SimHash and MinHash. For MinHash, we implement a variant called k-partition MinHash or one permutation hashing [35]. It is noticeably more computationally efficient than the original version of MinHash since 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. The k-partition variant still provides reasonable clustering results, though the accuracy bound in theorem 3.2.3 no longer applies for this variant; k-partition MinHash is unbiased and has lower variance than standard MinHash, but it is no longer clear whether there is a convenient tail bound for k-partition MinHash like the Hoeffding bound that the proof of theorem 3.2.3 uses.

When the number of samples k for the locality-sensitive hashing approximation scheme is high, it becomes more expensive to compute sketches and to process the sketches to get the approximate similarities. For low-degree vertices, the merging algorithm described in section 3.3.1 is cheap enough that it is better to compute similarities exactly rather approximate them. As a simple example, consider the case where two adjacent vertices have degree significantly less than k. It is faster to process the original neighbor lists of the vertices than it is to process 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 an arbitrarily set 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. (There is likely room for improvement for this heuristic. For one, we did not choose the threshold values particularly carefully. Secondly, computing the sketch for a high-degree vertex v is really only worth the $O(k|\overline{N}(v)|)$-work cost if the vertex has a large number of high-degree neighbors.)

Chapter 4

Experiments

4.1 Benchmarking environment

We run experiments on an Amazon Elastic Cloud Compute (EC2) c5.24xlarge instance, which has 192 GiB of RAM and 48 CPU cores with two-way hyper-threading for a maximum of 96 threads. We compare our parallel algorithm (which we refer to as GBBS-IndexSCAN on the plots in this chapter) using all 96 threads to our algorithm using only 1 thread, to the original sequential GS*-Index implementation,[1] and to ppSCAN [13][2] using all 96 threads. Comparing against ppSCAN, a parallel and well optimized SCAN variant, is valuable because experiments by the authors of ppSCAN suggest that it outperforms several other SCAN variants. For fixed parameters μ and ε, all of these algorithms return the same output except that any ambiguous border vertices might have different assignments. All code is C++ code compiled with GCC using the `-O3` optimization flag. We run the parallel codes with `numactl --interleave=all`, which interleaves memory allocations across CPUs and gives better performance for this particular problem on the EC2 instance.

Table 4.1 summarizes the graphs we use in the experiments. "Orkut" and "Friendster" are the com-Orkut and com-Friendster graphs respectively from the Stanford Large Network Dataset Collection [34].[3] Both are social network graphs in which

[1]GS*-Index code received via personal correspondence with the authors of the algorithm.
[2]ppSCAN code available at `https://github.com/RapidsAtHKUST/ppSCAN`.
[3]`https://snap.stanford.edu/data/`

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 4.1: Summary of the graphs for the experiments. Each undirected edge is counted only once rather than being counted as two directed edges.

the nodes are users and the edges represent friend relationships. "Brain" is the bn-human-Jung2015-M87113878 dataset from NeuroData[4] provided by Network Repository [48][5] that represents a mapping of human brain connections. "WebBase" is the webbase-2001 graph from the Laboratory for Web Algorithmics [7, 6][6] representing the links discovered by a web crawler. Although the WebBase original graph is a directed graph, we remove self-loop edges and change the edges to be undirected so that SCAN can operate on the graph. "Blood vessel" and "cochlea" are weighted graphs from HumanBase [27].[7] Nodes 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 functional relationship. For computational convenience, on the brain, WebBase, 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 only run GBBS-IndexSCAN on the blood vessel and cochlea graphs. Moreover, we only test cosine similarity on the weighted graphs since our implementation of Jaccard similarity does not handle weighted graphs. The main reason for including the two weighted graphs in the experiments is that they serve as denser graphs on which similarity approximation has potential to be useful.

[4]`https://neurodata.io/`

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

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

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

4.2 Results

4.2.1 Index construction time comparison

The first experiment measures the running time to construct the SCAN index with exact cosine similarities. The running time to compute the index using Jaccard similarity as the similarity measure is about the same, so we do not measure it separately. Each time measurement is the median of five trials. Figure 4-1 shows the time measurements, and figure 4-2 translates the time measurements into the speedup factor that GBBS-IndexSCAN achieves. GBBS-IndexSCAN achieves a parallel self-speedup factor of 23–70$\times$ on index construction. Moreover, GBBS-IndexSCAN running sequentially is 1.4-2.2$\times$ faster than the original GS*-Index implementation, likely due to the directed triangle counting optimization that section 3.3.1 describes, so the speedup of GBBS-IndexSCAN running on 96 threads is 50–151$\times$ over GS*-Index.

4.2.2 Clustering time comparison

The second experiment is to measure the running time for querying for the clustering for various settings of parameters (μ, ε). This experiment again only considers exact cosine similarity for the similarity measure since the patterns of query running times do not differ in a particularly insightful way when the similarity measure differs. Each time measurement is the median of five trials. Figure 4-3 measures the running times with $\mu = 5$ and $\varepsilon \in \{.1, .2, .3, \dots, .9\}$, figure 4-4 measures the running times $\varepsilon = 0.6$ and $\mu \in \left\{2, 4, 8, 16, \dots, \min\left\{16384, 2^{\lfloor \log(\text{max degree}) \rfloor}\right\}\right\}$, and figure 4-5 plots the speedup factor of GBBS-IndexSCAN on one particular parameter setting: $(\mu, \varepsilon) = (5, .6)$.

GBBS-IndexSCAN is faster than ppSCAN and GS*-Index on all tested parameter settings, though of course this is not quite a fair comparison against ppSCAN since GBBS-IndexSCAN takes a considerable amount of time precomputing an index. This extra cost that GBBS-IndexSCAN incurs is preferable over ppSCAN only when the user wants to query many different parameter settings. Notably, though, it might

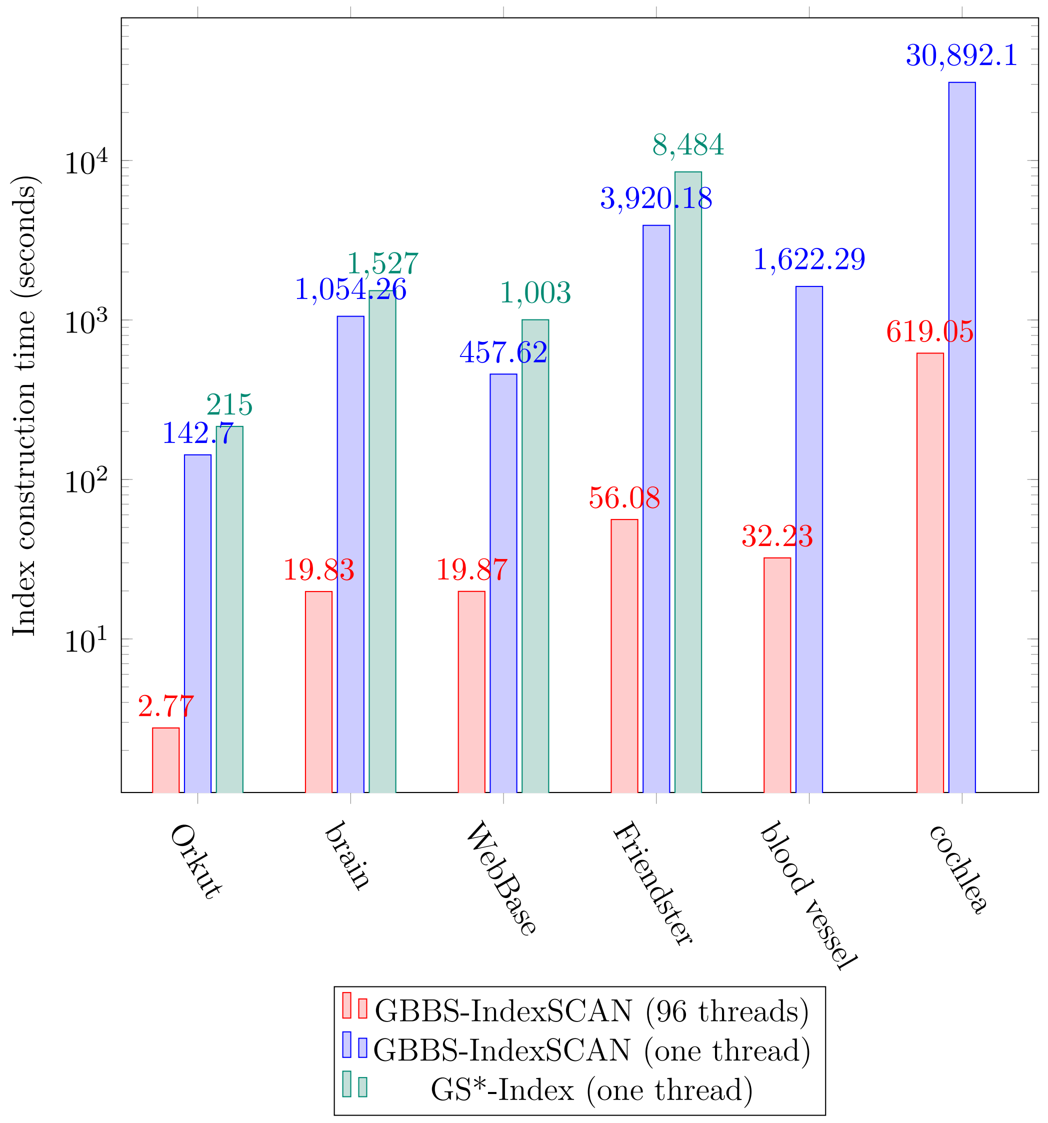

Figure 4-1: Index construction times with exact cosine similarity as the similarity measure.

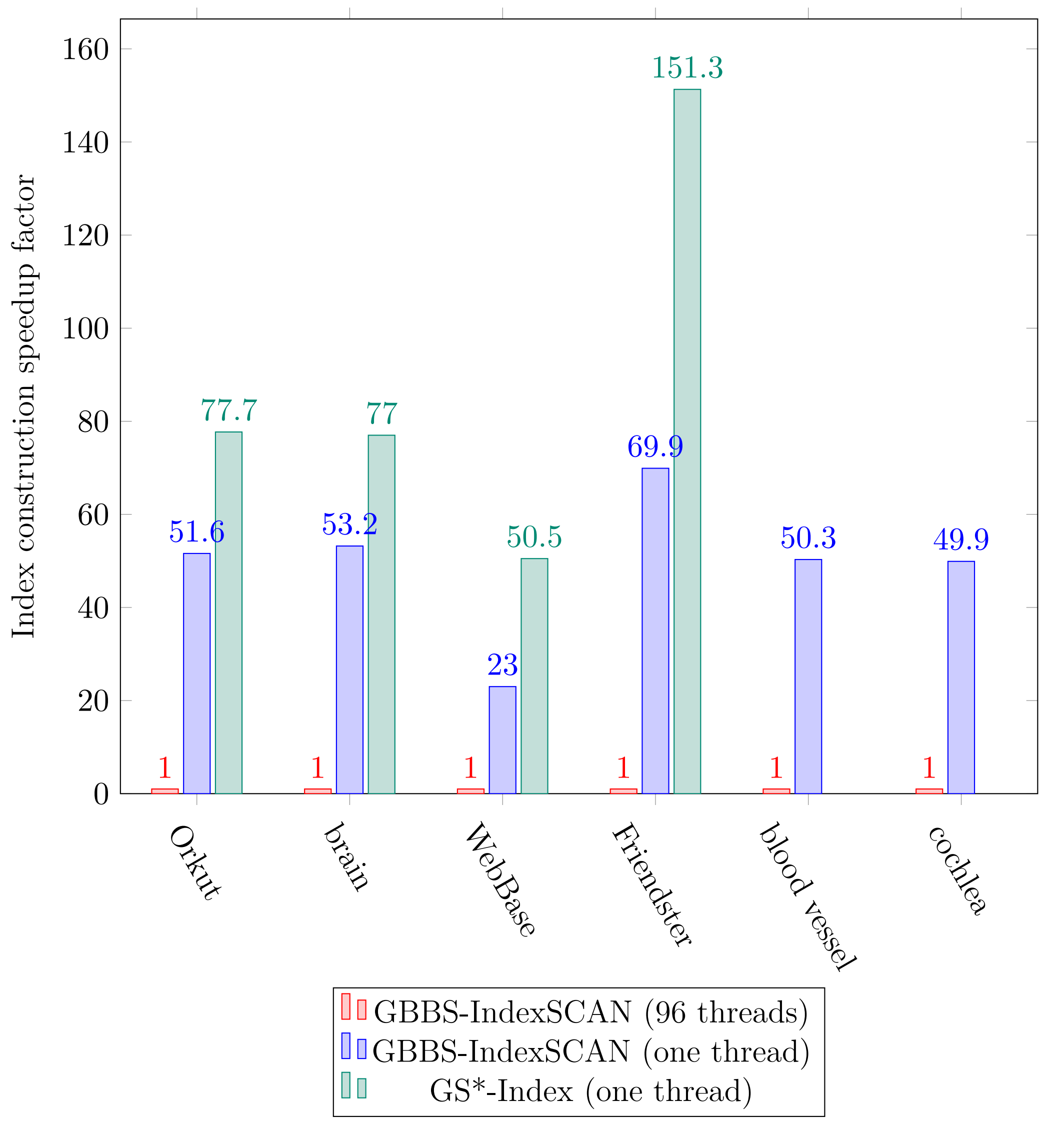

Figure 4-2: Speedup factor on index construction that GBBS-IndexSCAN (96 threads) achieves relative to GBBS-IndexSCAN running sequentially and relative to the original GS*-Index implementation.

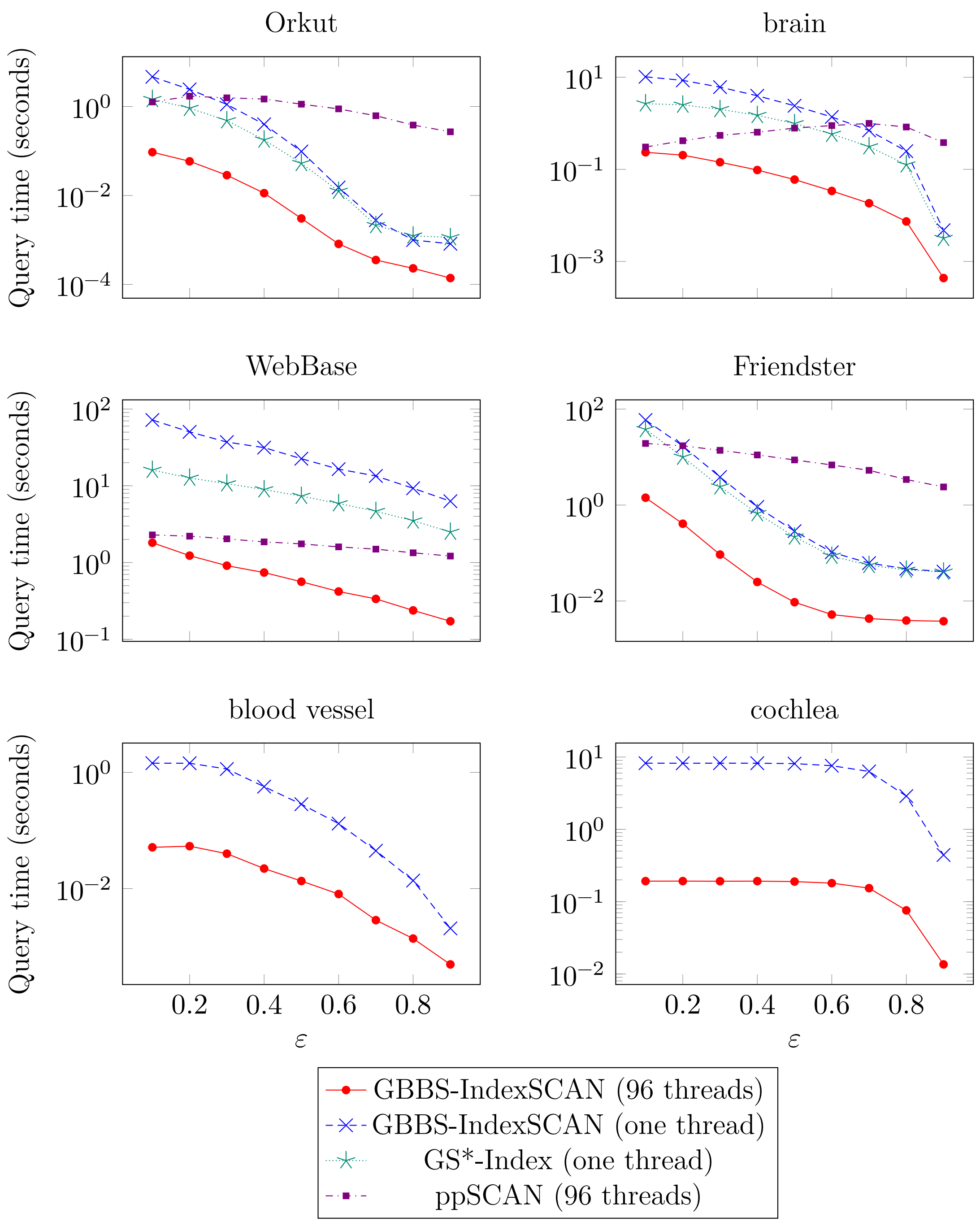

Figure 4-3: Clustering time with $\mu = 5$ and varying ε using exact cosine similarity as the similarity measure.

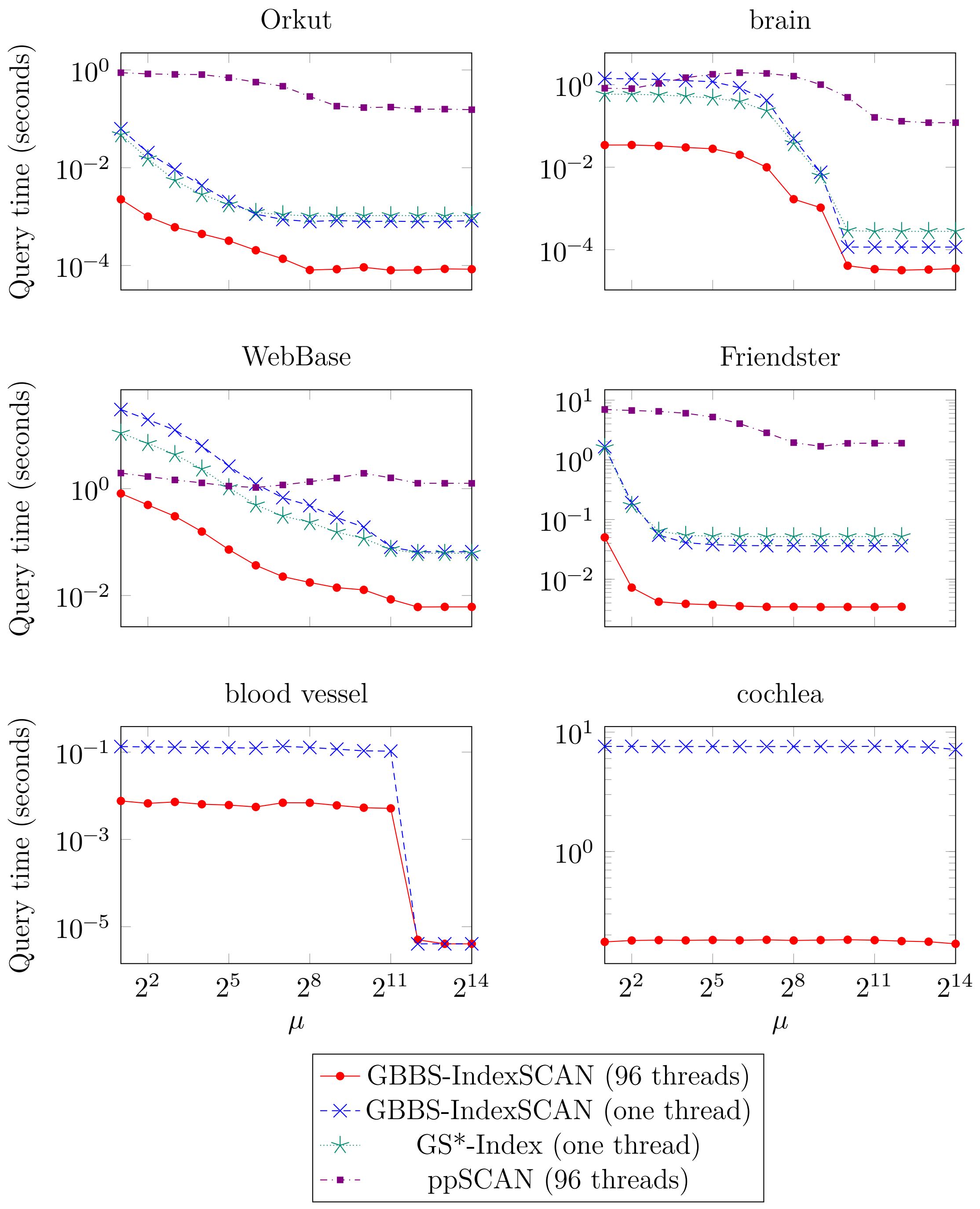

Figure 4-4: Clustering time with $\varepsilon = 0.6$ and varying μ using exact cosine similarity as the similarity measure.

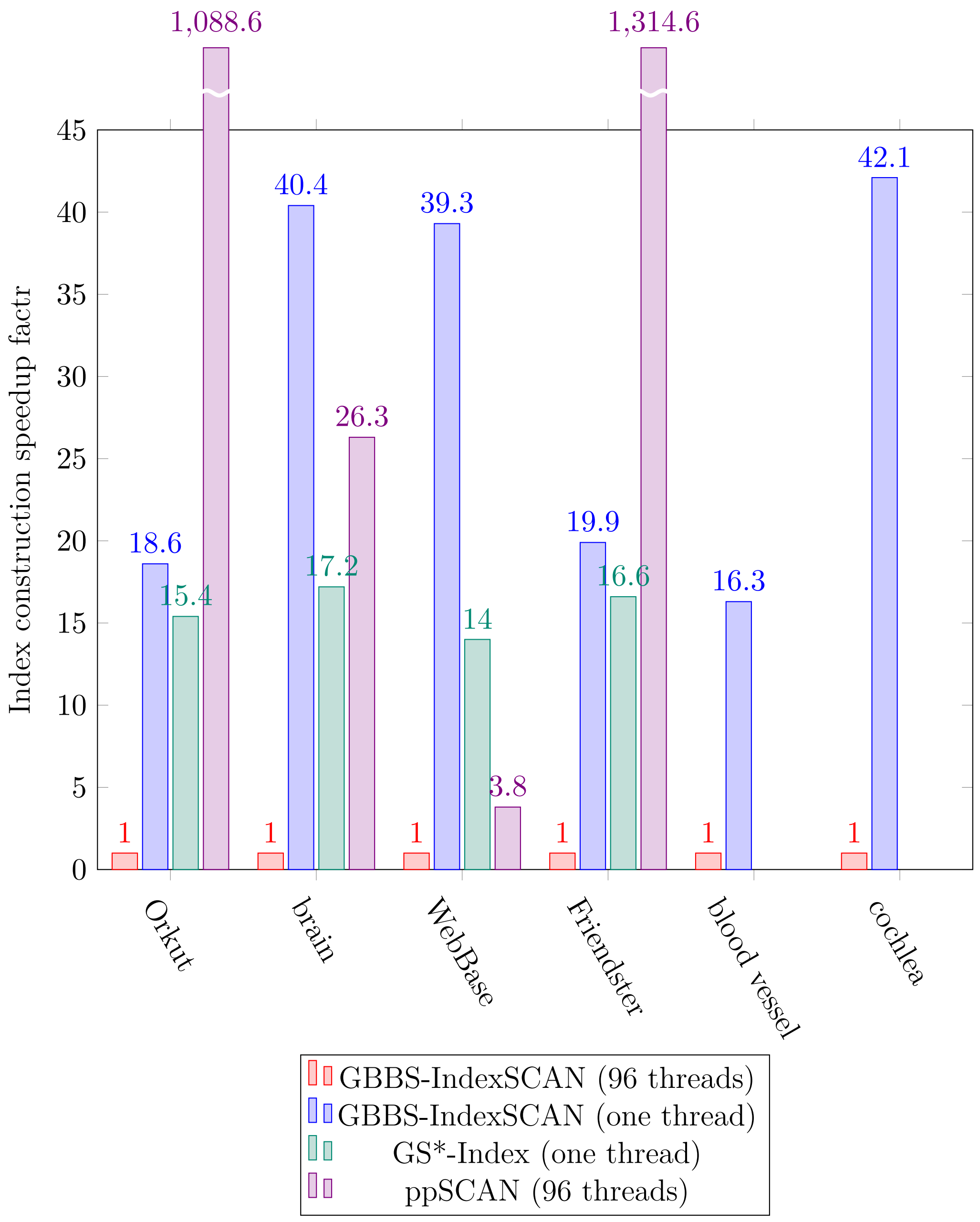

Figure 4-5: Speedup factor that GBBS-IndexSCAN (96 threads) achieves relative to other SCAN implementations on querying for clusters with the parameter setting $(\mu, \varepsilon) = (5, .6)$.

not take many queries for GBBS-IndexSCAN 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 4-3 exceeds the sum of the corresponding time measurements for GBBS-IndexSCAN plus the time for GBBS-IndexSCAN to construct its index.

GBBS-IndexSCAN running sequentially tends to be slower at querying for clusters than GS*-Index. The worse sequential performance is due to the adjustments made in GBBS-IndexSCAN to make it more friendly to parallelism, namely its use of union-find instead of sequential breadth-first search as well as how it iterates over all edges an additional time to assign non-core vertices (algorithm 5). It is up to 4.5× slower than GS*-Index on the tested parameter settings and graphs. GBBS-IndexSCAN running on 96 threads, however, is faster than the other implementations on all tested parameter settings; it is faster than GS*-Index by 5–32× and faster than ppSCAN by 1.26–12,070×.

4.2.3 Approximate index construction time

The third experiment measures the running time of constructing GBBS-IndexSCAN with 96 threads using the approximate cosine and approximate Jaccard similarity measures with varying numbers of samples. For the weighted graphs, we only test the approximate cosine similarity measure since the k-partition MinHash variant that we implement does not handle weighted graphs. Each time measurement is again a median of five trials. Each trial uses a different pseudorandom seed for the randomness in the approximation schemes. Figure 4-6 displays the results. As anticipated, the best speedups are on the denser graphs like the cochlea graph, whereas approximation is unhelpful on the Friendster graph even with modest sample sizes. The approximate Jaccard similarity implementation is consistently faster than the approximate Cosine similarity implementation because of the better efficiency of constructing sketches for k-partition MinHash compared to for SimHash. The times plateau or even decrease at large sample sizes for some of the graphs due to the heuristic discussed in section 3.3.3 for avoiding computing sketches for low-degree vertices.

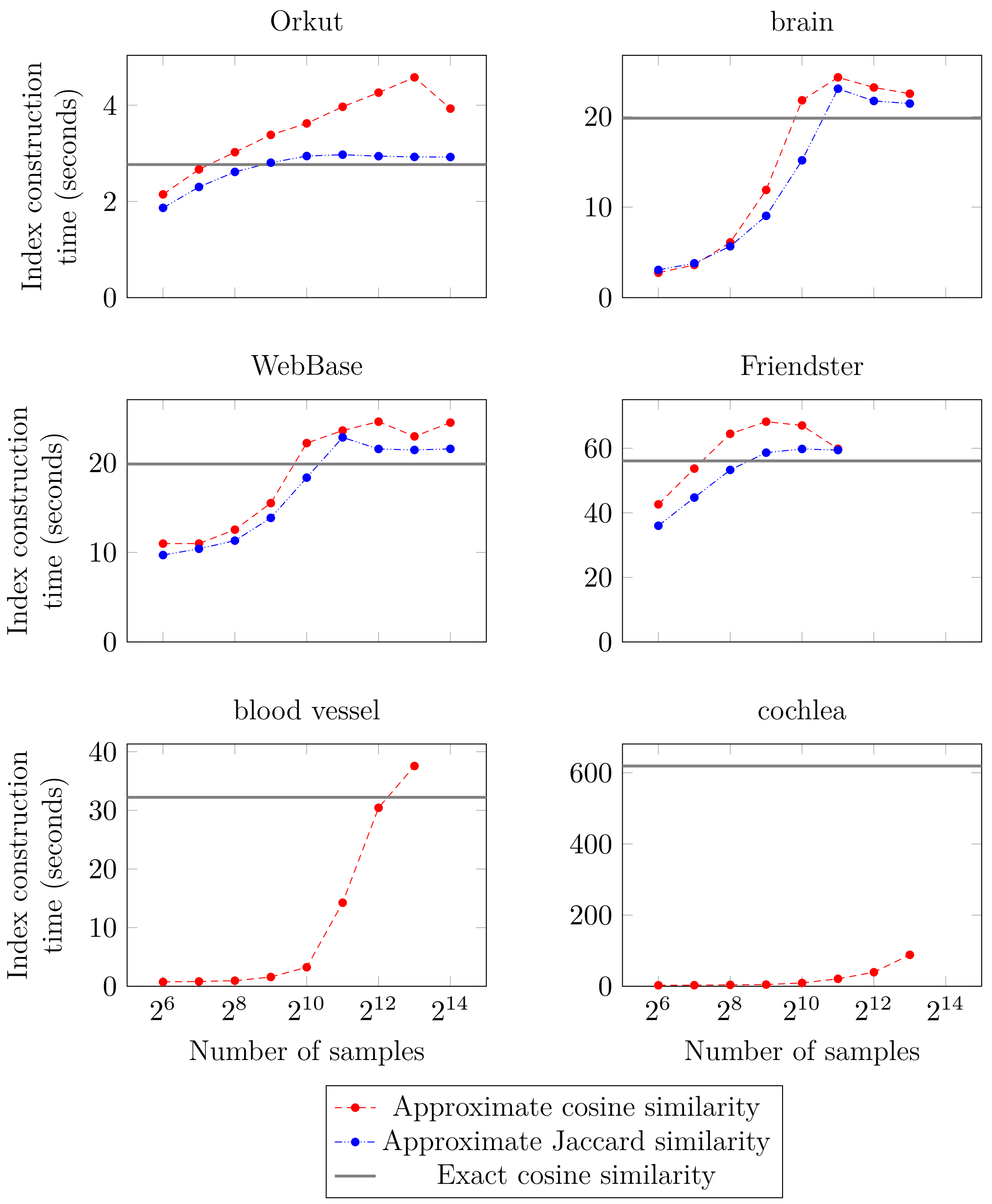

Figure 4-6: Index construction times for GBBS-IndexSCAN (96 threads) using approximate similarity measures with varying sample sizes.

4.2.4 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 5) portion of the clustering algorithm assigns each border non-core vertex to the same cluster as an arbitrary ε-similar core vertex, in order 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.

First, we need to find SCAN parameters that give a good clustering. Let Σ be the set of (μ, ε) parameter settings

$$\Sigma = \{2, 4, 8, 16, \dots, 2^{18}\} \times \{.01, .02, .03, \dots, .99\}.$$

We use the modularity as a heuristic measurement for clustering quality, treating unclustered vertices as each being in their own cluster. We find the parameters $(\mu_{\text{exact-cos}}, \varepsilon_{\text{exact-cos}}) \in \Sigma$ giving the clustering with the best modularity under exact cosine similarity. Then, for a fixed sample size, we similarly find the best clustering from Σ under approximate cosine similarity, and we also look at the clustering under $(\mu_{\text{exact-cos}}, \varepsilon_{\text{exact-cos}})$ with approximate cosine similarity. We repeat this process with Jaccard similarity. With varying sample sizes, we plot the resulting modularities in figures 4-7 and 4-8. Each modularity score with the approximate similarity measures is the mean of five trials with different pseudorandom seeds. Figures 4-9 and 4-10 plots the adjusted Rand index (ARI) of the clustering under approximate cosine similarity at $(\mu_{\text{exact-cos}}, \varepsilon_{\text{exact-cos}})$ (or $(\mu_{\text{exact-Jaccard}}, \varepsilon_{\text{exact-Jaccard}})$) versus the "ground-truth" clustering under exact cosine similarity (and likewise for Jaccard similarity). The best parameter settings in Σ often have high μ values, which is somewhat surprising considering that past SCAN work rarely considers μ values above 20.

The improved approximation accuracy in these plots as the sample size increases is not only attributable to better accuracy in locality-sensitive hashing with higher

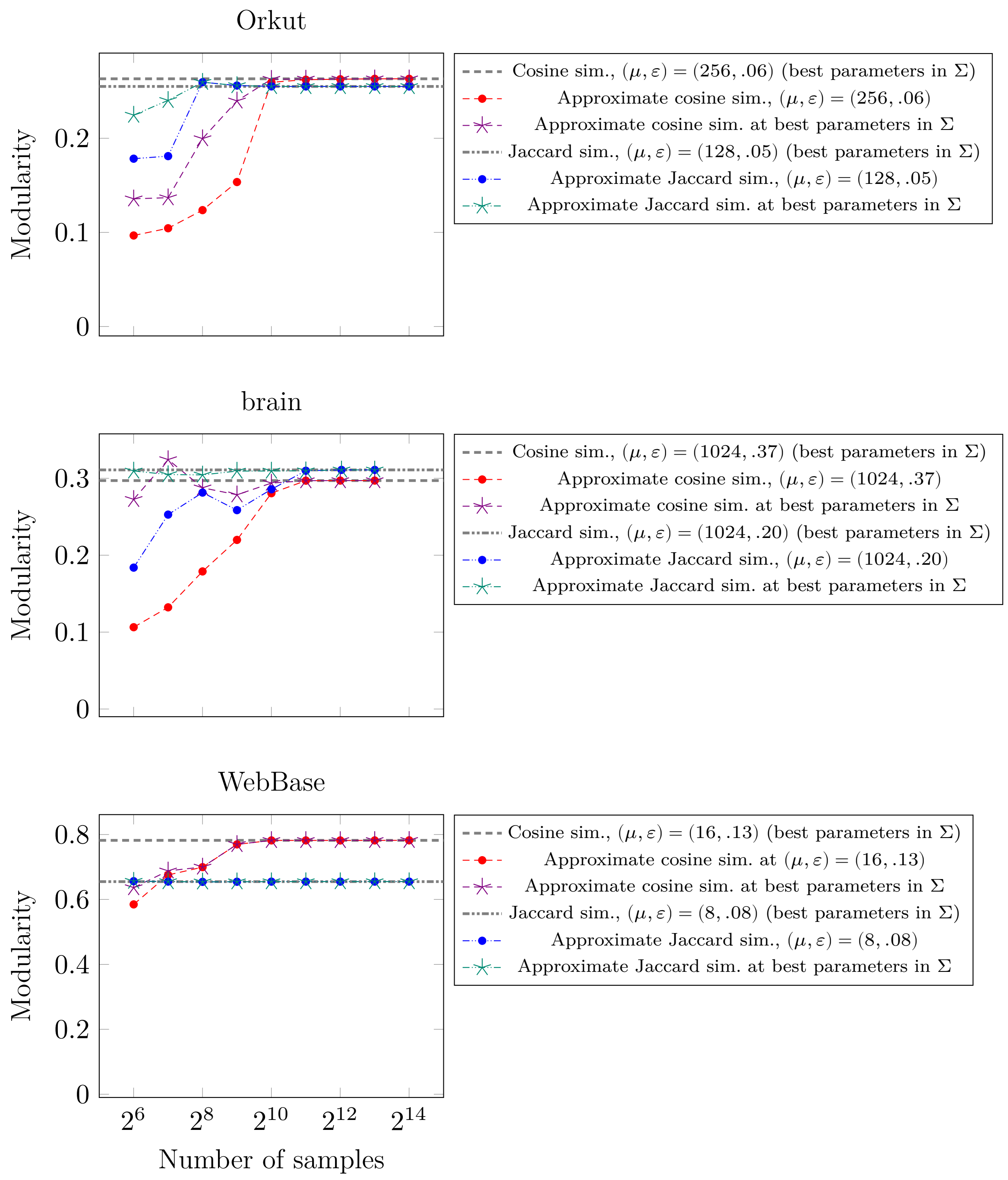

Figure 4-7: Modularity scores achieved by approximate similarity measures. We choose SCAN parameters (μ, ε) out of the set $\Sigma = \{2, 4, 8, 16, \ldots, 2^{18}\} \times \{0.01, 0.02, 0.03, \ldots, 0.99\}$.

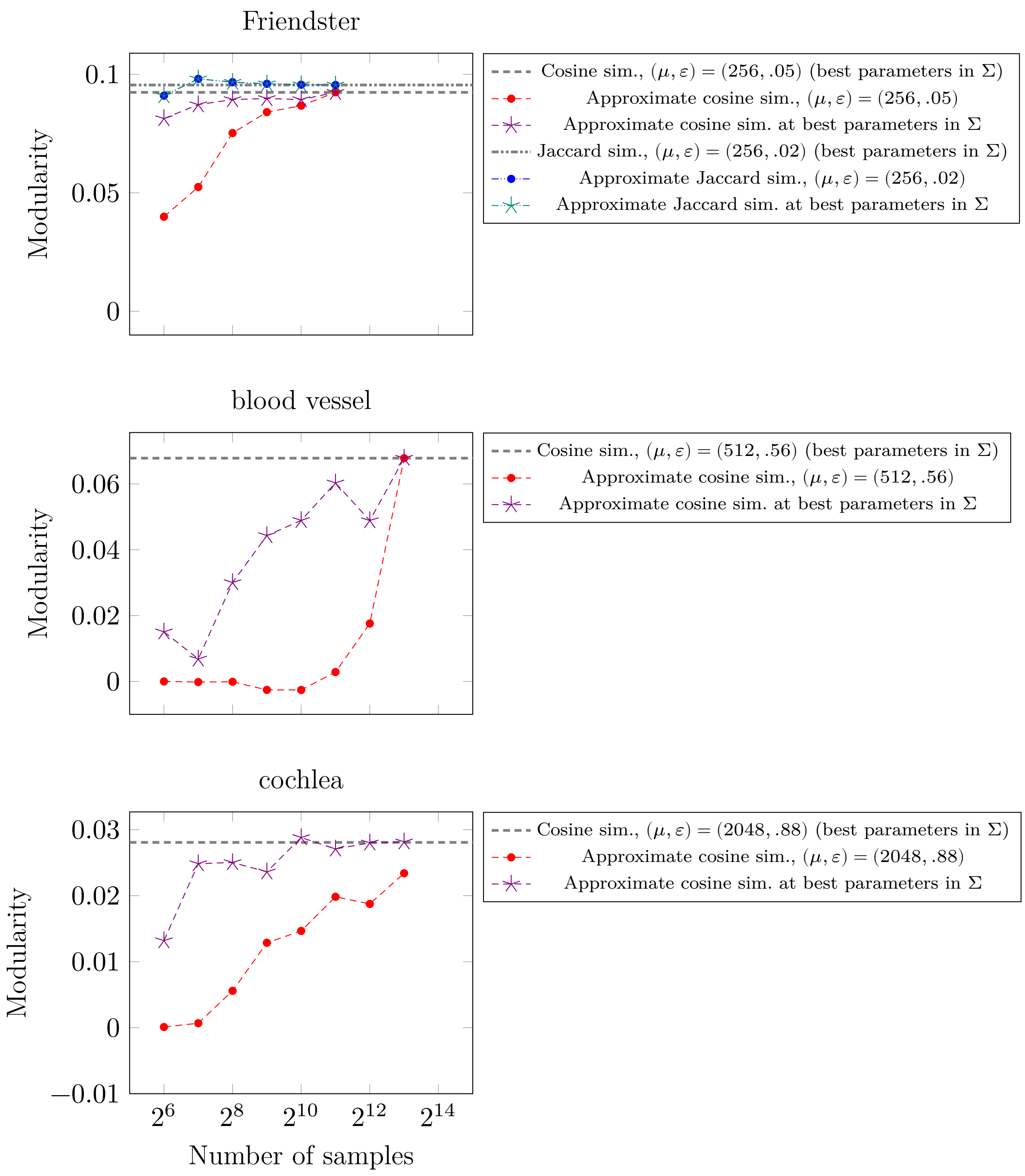

Figure 4-8: Continuation of figure 4-7.

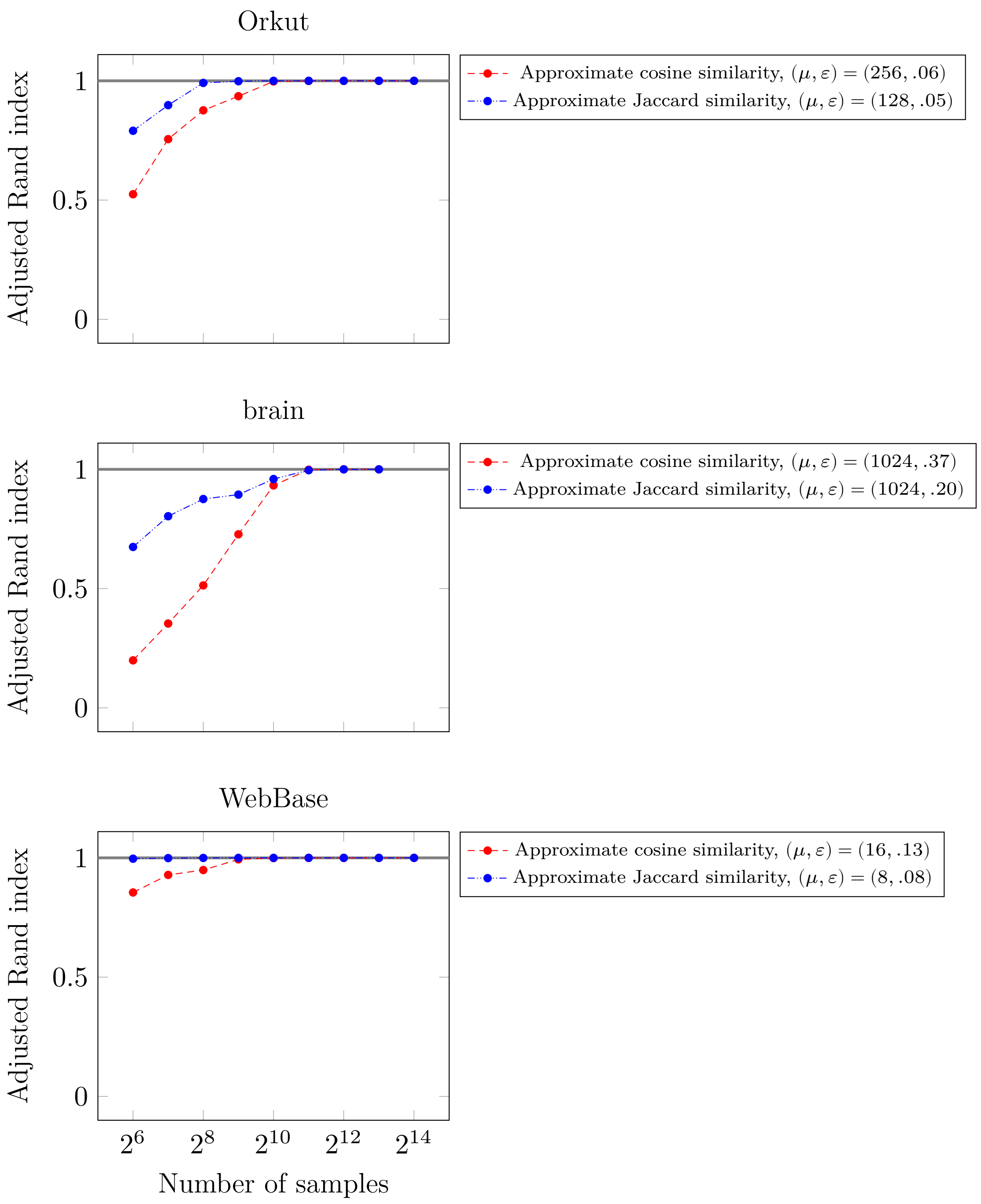

Figure 4-9: Accuracy of clusters using approximate similarity measures against a ground truth of the clusters given by the corresponding exact similarity measures. Parameters are the best SCAN parameters in Σ for the exact similarity measure.

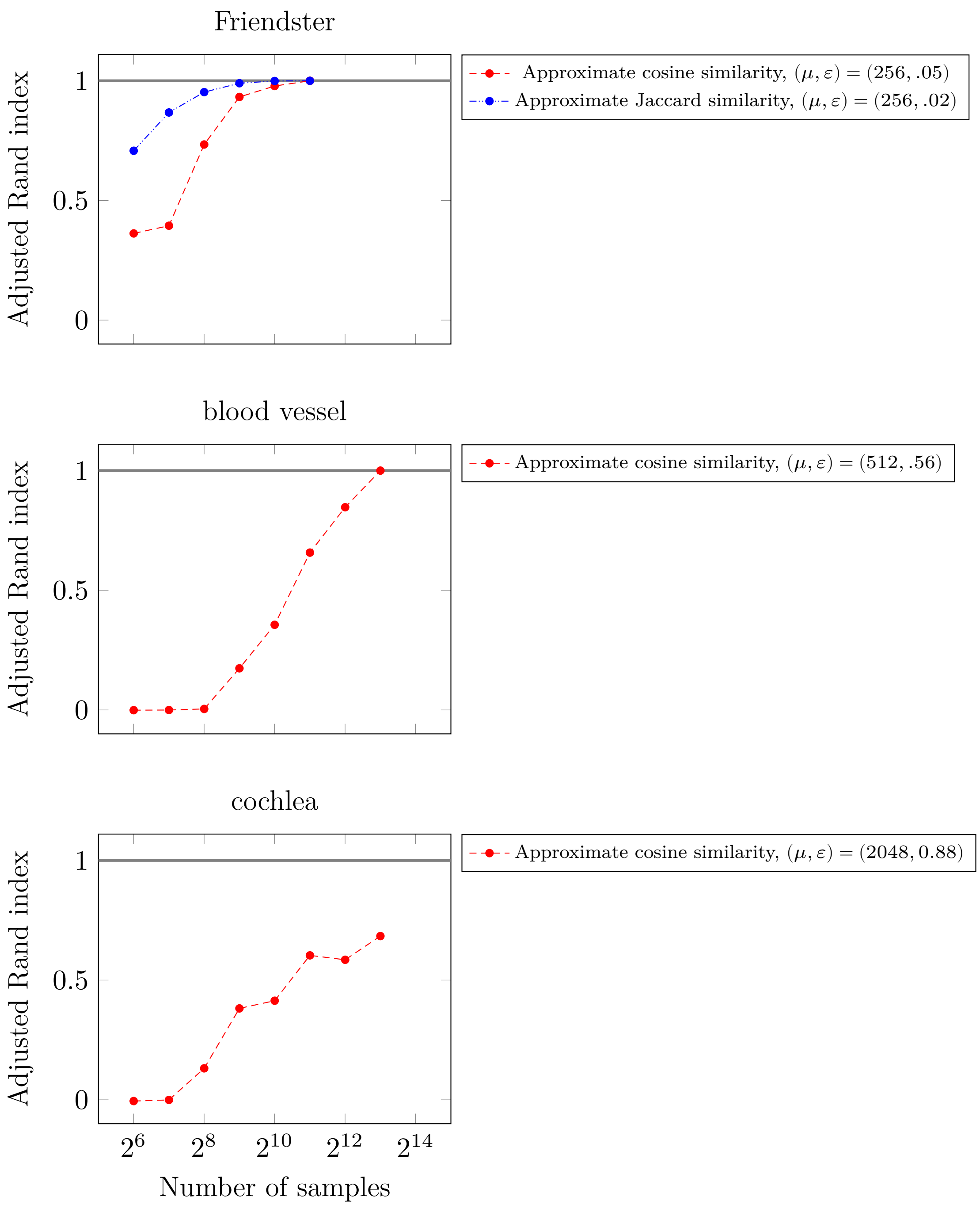

Figure 4-10: Continuation of figure 4-9.

numbers of samples but also to the heuristic described in section 3.3.3 that reverts to computing exact similarity for vertices that have low degree relative to the number of samples.

The approximate Jaccard clusterings approach the clustering quality of the corresponding exact similarity clustering 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 [53]. The accuracy bound in theorem 3.2.3 compared to the bound in theorem 3.2.2 also suggests that MinHash might give closer approximations than SimHash does assuming that the bounds are tight.

(Though we use modularity as a measurement of quality, if a clustering practitioner's goal is solely to find a clustering that maximizes modularity, then it is likely be better to use a clustering algorithm tailored for maximizing modularity. For instance, our experiments show SCAN getting a peak modularity of less than .1 on the Friendster graph across parameters in Σ, whereas LaSalle and Karypis report that their modularity-maximizing clustering tool Nerstrand finds a clustering with modularity around .6 [33]. The low peak modularity of .028 that SCAN finds for the cochlea graph, however, is perhaps more indicative of the cochlea graph not containing good clusters — Nerstrand only achieves a modularity of .06927 on the cochlea graph, and community-el [47], another modularity-maximizing clustering algorithm, achieves a modularity of .007, which is even lower than what SCAN finds.)

Figure 4-11 takes the modularity scores from figures 4-7 and 4-8 and plots them against the corresponding approximate index construction times from figure 4-6 on the horizontal axis rather than the number of samples. Likewise, figure 4-12 crosses ARI scores from figures 4-9 and 4-10 on the vertical axis against the approximate index construction times from figure 4-6 on the horizontal axis. These two plots, figure 4-11 and figure 4-12, are trade-off curves between the time to construct the SCAN index versus the quality of the clusterings resulting from that index. Points to the top and to the left represent sample sizes that give good quality as well as low index construction times. The plots also include the times to construct the index with the exact similarity measures from figure 4-1 with the assumption that the times for exact Jaccard similarity are the same as those measured for exact cosine similarity.

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

All in all, the timing experiments show that GBBS-IndexSCAN achieves high parallelism and performs competitively against ppSCAN, and the approximation experiments suggest that locality-sensitive hashing can speed up index construction significantly on dense graphs without sacrificing clustering quality.

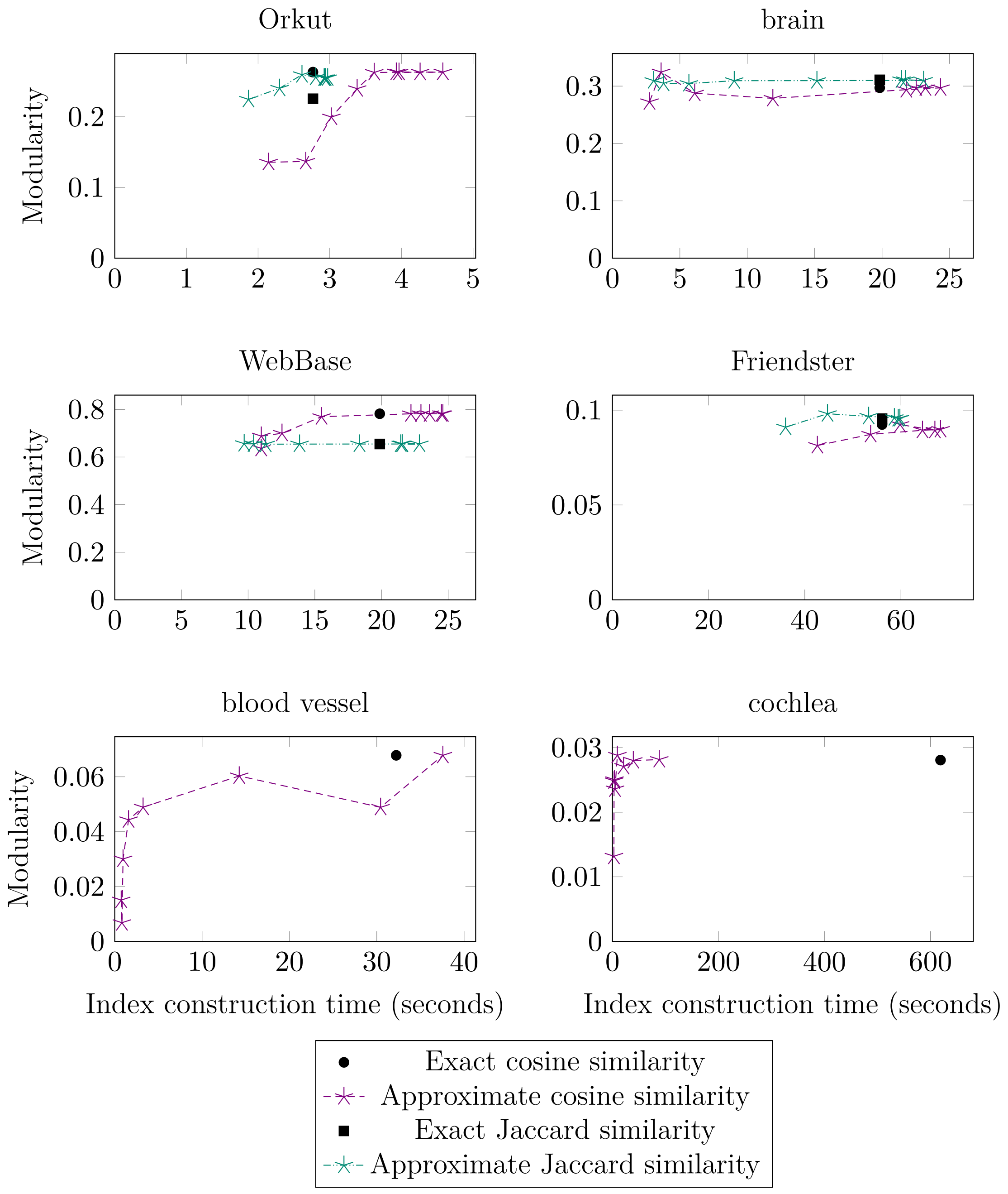

Figure 4-11: Comparison of GBBS-IndexSCAN (96 threads) approximate index construction times with varying numbers of samples versus the best modularity score found using any parameters in Σ. These plots use the running times from figure 4-6 on the horizontal axis and the modularities from figures 4-7 and 4-8 on the vertical axis.

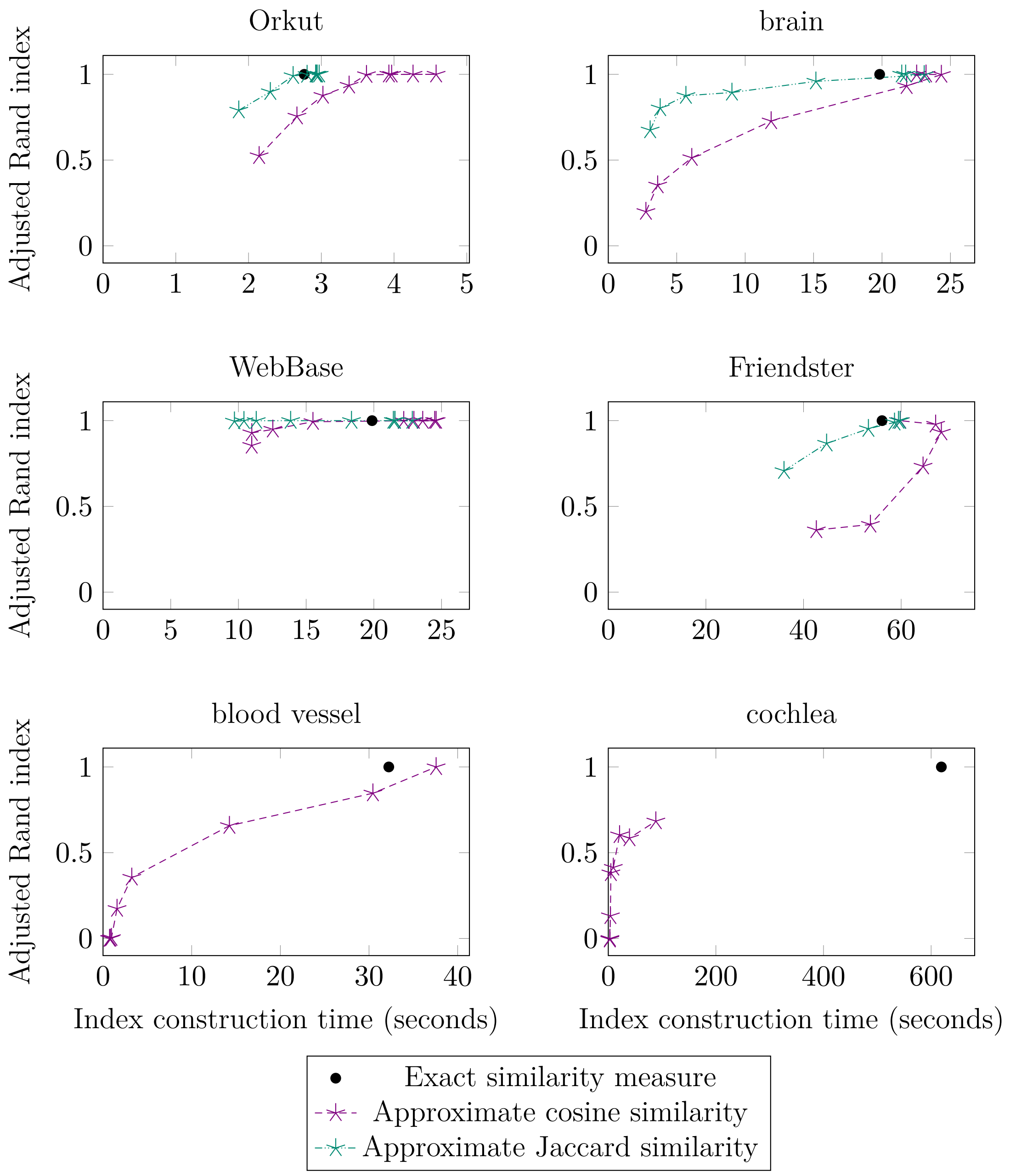

Figure 4-12: Comparison of GBBS-IndexSCAN (96 threads) approximate index construction times versus the accuracy of clusterings against a ground truth of the clustering given by exact similarity measures. The parameters used are the best SCAN parameters in Σ for the exact similarity measure. These plots use the running times from figure 4-6 on the horizontal axis and the adjusted Rand scores from figures 4-9 and 4-10 on the vertical axis.

Chapter 5

Related Work

Xu et al. introduced the original SCAN algorithm [61], using ideas from the popular point clustering algorithm DBSCAN [22]. There is much research in developing variants of SCAN that make it more usable and more efficient.

One of the inconvenient aspects of SCAN is that it is difficult to find good values for its two user-selected parameters, μ and ε. GS*-Index alleviates this issue by creating an index upon which future SCAN queries with arbitrary parameters are quick [59]. SCOT [8] and gSkeletonClu [30] also essentially compute indices for SCAN 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 appear together in the ordering. gSkeletonClu computes a spanning tree upon potential core vertices. Though we chose in this paper to focus attention on GS*-Index, more work is needed to compare these different indices in terms of computation time and space usage. gSkeletonClu in particular seems like it could give even faster clustering query times than GS*-Index, though gSkeletonClu is less flexible in the sense that it requires the user to operate on a fixed μ value.

SHRINK [29], DHSCAN [62], and AHSCAN [63] all borrow ideas from 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 [60] 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, though on some level having some tunable parameters can be helpful if they allow the user to explore other reasonable clusterings on a graph.

Other work building on SCAN focuses on making SCAN scale to large graphs. LinkSCAN* [36] reduces computation time at the cost of accuracy by operating on a sampled subgraph of the original graph. We present a different approximation idea using locality-sensitive hashing in this thesis, but it would be worthwhile in the future to compare the efficiency and clustering quality of these two different approximation approaches. Zhao et al. [64] and Mai et al. [39] 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 quick as possible so that this anytime functionality is unnecessary.

SCAN++ [52], pSCAN [11], and ppSCAN [13], 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. Because there are fewer similarity score computations to prune when building an index applicable to queries on arbitrary SCAN parameters and because it keeps the algorithm design simpler, we do not consider pruning similarity computations for our algorithm and instead achieve speed through precomputing an index.

For distributed systems, Chen et al. [14] and Zhao et al. [65] present MapReduce [17] parallelizations of SCAN, and SparkSCAN [66] is a Spark parallelization of SCAN. In contrast, we use a shared-memory, single multicore machine programming model in this thesis, which is faster and tends to be easier to program in. Though massive graphs may necessitate using distributed systems, there exist commodity multi-core machines with enough RAM to process fairly large graphs. GPUSCAN [56] explores using GPUs to speed up SCAN, whereas we focus on CPU-based algorithms in this thesis. SCAN-XP [57] is a parallel SCAN algorithm for multicore machines, but we do not compare our algorithm against SCAN-XP since ppSCAN is faster according to Che et al.'s experiments.

There are many other graph clustering algorithms besides SCAN and its variants. We will not attempt to list those other algorithms here, but interested readers may wish to refer to existing surveys written by others: [49, 23]. A practitioner interested in graph clustering should, of course, make a careful choice about what graph clustering algorithm is appropriate for their needs based on each algorithm's computational performance and on what kind of clusters each algorithm produces.

Chapter 6

Conclusion

This thesis presented a index-based SCAN algorithm that achieves significant parallel speed-up and allows a user to query efficiently for SCAN clusterings at arbitrary parameter settings. The algorithm is work-efficient in expectation relative to the sequential algorithm that it is based on, GS*-Index, and has logarithmic span with high probability. We also present a highly optimized, multicore implementation of the algorithm that runs well in practice. Moreover, we demonstrate that locality-sensitive hashing is a viable approximation scheme to speed up the computationally expensive component of index construction on dense graphs.

There are many potential future avenues of research and exploration based on the work in this thesis. For instance,

- GS*-Index can handle dynamic edge insertions and deletions to the graph. Is it possible to parallelize this dynamic component as well and efficiently process batches of edge updates at once?
- gSkeletonClu is another index-based SCAN algorithm that may be parallelizable. What ideas can we take from gSkeletonClu to get an even faster parallel SCAN algorithm? It seems like gSkeletonClu may be able to achieve lower clustering query times than GS*-Index does.
- In order to reduce the cost of running SCAN, LinkSCAN* samples edges from its input graph and runs SCAN on the sampled graph. How does this ap-

proximation scheme compare to locality-sensitive hashing in running time and clustering quality?

- Despite there being a significant amount of prior work in performance optimizations for SCAN, it is still unclear in what situations, if any, a practitioner would prefer SCAN over other graph clustering algorithms. How do various graph clustering algorithms compare? What qualities do practitioners prioritize most in graph clustering? Based on the answers to these questions, is SCAN still relevant or is it obsolete?

Bibliography

[1] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering points to identify the clustering structure. *SIGMOD Record*, 28(2):49–60, 1999.

[2] K. Aydin, M. Bateni, and V. Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, page 387–396. Association for Computing Machinery, 2016.

[3] A. Bellogín and J. Parapar. Using graph partitioning techniques for neighbour selection in user-based collaborative filtering. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, page 213–216. Association for Computing Machinery, 2012.

[4] C. Biemann. 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*, page 73–80. Association for Computational Linguistics, 2006.

[5] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In M. J. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*, volume 2, chapter 25. Chapman & Hall/CRC, 2nd edition, 2010.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, page 587–596. Association for Computing Machinery, 2011.

[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, page 595–602. Association for Computing Machinery, 2004.

[8] D. Bortner and J. Han. Progressive clustering of networks using structure-connected order of traversal. In *IEEE 26th International Conference on Data Engineering*, pages 653–656. IEEE, 2010.

[9] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[10] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of SEQUENCES*, pages 21–29. IEEE, 1997.

[11] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang. pSCAN: Fast and exact structural graph clustering. *IEEE Transactions on Knowledge and Data Engineering*, 29(2):387–401, 2017.

[12] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 380–388. Association for Computing Machinery, 2002.

[13] Y. Che, S. Sun, and Q. Luo. Parallelizing pruning-based graph structural clustering. In *Proceedings of the 47th International Conference on Parallel Processing*. Association for Computing Machinery, 2018.

[14] J.-J. Chen, J.-M. Chen, J. Liu, and V.-L. Huang. PSCAN: A parallel structural clustering algorithm for networks. In *International Conference on Machine Learning and Cybernetics*, volume 2, pages 839–844. IEEE, 2013.

[15] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[16] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, page 393–404. Association for Computing Machinery, 2018.

[19] L. Dhulipala, C. Hong, and J. Shun. ConnectIt: A framework for static and incremental parallel graph connectivity algorithms. *arXiv preprint arXiv:2008.03909*, 2020.

[20] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun. The graph based benchmark suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics*. Association for Computing Machinery, 2020.

[21] Y. Ding, M. Chen, Z. Liu, D. Ding, Y. Ye, M. Zhang, R. Kelly, L. Guo, Z. Su, S. C. Harris, F. Qian, W. Ge, H. Fang, X. Xu, and W. Tong. atBioNet–an integrated network analysis tool for genomics and biomarker discovery. *BMC Genomics*, 13, 2012.

[22] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. 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*, page 226–231. AAAI Press, 1996.

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

[24] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, 1991.

[25] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, page 698–710. IEEE Computer Society, 1991.

[26] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[27] C. S. Greene, A. Krishnan, A. K. Wong, E. Ricciotti, R. A. Zelaya, D. S. Himmelstein, R. Zhang, B. M. Hartmann, E. Zaslavsky, S. C. Sealfon, D. I. Chasman, G. A. FitzGerald, K. Dolinski, T. Grosser, and T. O. G. Understanding multicellular function and disease with human tissue-specific networks. *Nature Genetics*, 47:569–576, 2015.

[28] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[29] J. Huang, H. Sun, J. Han, H. Deng, Y. Sun, and Y. Liu. SHRINK: A structural clustering algorithm for detecting hierarchical communities in networks. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 219–228. Association for Computing Machinery, 2010.

[30] J. Huang, H. Sun, Q. Song, H. Deng, and J. Han. Revealing density-based clustering structure from the core-connected tree of a network. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1876–1889, 2013.

[31] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[32] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[33] D. LaSalle and G. Karypis. Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, 76:66–80, 2015.

[34] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

[35] P. Li, A. Owen, and C.-H. Zhang. One permutation hashing. In *Advances in Neural Information Processing Systems 25*, pages 3113–3121. Curran Associates, Inc., 2012.

[36] S. Lim, S. Ryu, S. Kwon, K. Jung, and J.-G. Lee. LinkSCAN*: Overlapping community detection using the link-space transformation. In *IEEE 30th International Conference on Data Engineering*, pages 292–303. IEEE, 2014.

[37] C. X. Lin, Y. Yu, J. Han, and B. Liu. 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*, page 222–229. Springer, 2010.

[38] Z. Liu, Q. Shi, D. Ding, R. Kelly, H. Fang, and W. Tong. Translating clinical findings into knowledge in drug safety evaluation-drug induced liver injury prediction system (DILIps). *PLOS Computational Biology*, 7(12), 2011.

[39] S. T. Mai, S. Amer-Yahia, I. Assent, M. S. Birk, M. S. Dieu, J. Jacobsen, and J. M. Kristensen. Scalable interactive dynamic graph clustering on multicore CPUs. *IEEE Transactions on Knowledge and Data Engineering*, 31(7):1239–1252, 2019.

[40] V.-S. Martha, Z. Liu, L. Guo, Z. Su, Y. Ye, H. Fang, D. Ding, W. Tong, and X. Xu. Constructing a robust protein-protein interaction network by integrating multiple public databases. In *BMC Bioinformatics*, volume 12. Springer, 2011.

[41] M. Mete, F. Tang, X. Xu, and N. Yuruk. A structural approach for finding functional modules from large biological networks. In *BMC Bioinformatics*, volume 9. Springer, 2008.

[42] M. E. J. Newman. Analysis of weighted networks. *Physical Review E*, 70:056131, 2004.

[43] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[44] S. Papadopoulos, Y. Kompatsiaris, and A. Vakali. Leveraging collective intelligence through community detection in tag networks. In *Proceedings of Workshop on Collective Knowledge Capturing and Representation*. Citeseer, 2009.

[45] S. Papadopoulos, Y. Kompatsiaris, and A. Vakali. A graph-based clustering scheme for identifying related tags in folksonomies. In *Data Warehousing and Knowledge Discovery*, pages 65–76. Springer-Verlag, 2010.

[46] S. Papadopoulos, C. Zigkolis, G. Tolias, Y. Kalantidis, P. Mylonas, Y. Kompatsiaris, and A. Vakali. Image clustering through community detection on hybrid image similarity graphs. In *IEEE International Conference on Image Processing*, pages 2353–2356. IEEE, 2010.

[47] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader. Parallel community detection for massive graphs. In *Parallel Processing and Applied Mathematics*, pages 286–296. Springer, 2011.

[48] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, page 4292–4293. AAAI Press, 2015.

[49] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[50] M. Schinas, S. Papadopoulos, Y. Kompatsiaris, and P. A. Mitkas. 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*, pages 203–210. Association for Computing Machinery, 2015.

[51] M. Schinas, S. Papadopoulos, G. Petkos, Y. Kompatsiaris, and P. A. Mitkas. Multimodal graph-based event detection and summarization in social media streams. In *Proceedings of the 23rd ACM International Conference on Multimedia*, page 189–192. Association for Computing Machinery, 2015.

[52] H. Shiokawa, Y. Fujiwara, and M. Onizuka. SCAN++: Efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment*, 8(11):1178–1189, 2015.

[53] A. Shrivastava and P. Li. In defense of MinHash over SimHash. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 886–894, 2014.

[54] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, page 96–107. Association for Computing Machinery, 2014.

[55] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE 31st International Conference on Data Engineering*, pages 149–160. IEEE, 2015.

[56] T. R. Stovall, S. Kockara, and R. Avci. GPUSCAN: GPU-based parallel structural clustering algorithm for networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3381–3393, 2015.

[57] T. Takahashi, H. Shiokawa, and H. Kitagawa. SCAN-XP: Parallel structural graph clustering algorithm on Intel Xeon Phi coprocessors. In *Proceedings of the 2nd International Workshop on Network Data Analytics*. Association for Computing Machinery, 2017.

[58] D. A. Tolliver and G. L. Miller. 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*, volume 1, page 1053–1060. IEEE Computer Society, 2006.

[59] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin. Efficient structural graph clustering: An index-based approach. *Proceedings of the VLDB Endowment*, 11(3):243–255, 2017.

[60] C. Wu, Y. Gu, and G. Yu. DPSCAN: Structural graph clustering based on density peaks. In *Database Systems for Advanced Applications*, pages 626–641. Springer, 2019.

[61] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: A structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 824–833. Association for Computing Machinery, 2007.

[62] N. Yuruk, M. Mete, X. Xu, and T. A. J. Schweiger. A divisive hierarchical structural clustering algorithm for networks. In *Proceedings of the Seventh IEEE International Conference on Data Mining Workshops*, pages 441–448. IEEE Computer Society, 2007.

[63] N. Yuruk, M. Mete, X. Xu, and T. A. J. Schweiger. AHSCAN: Agglomerative hierarchical structural clustering algorithm for networks. In *Proceedings of the International Conference on Advances in Social Network Analysis and Mining*, page 72–77. IEEE Computer Society, 2009.

[64] W. Zhao, G. Chen, and X. Xu. AnySCAN: an efficient anytime framework with active learning for large-scale network clustering. In *IEEE International Conference on Data Mining*, pages 665–674. IEEE Computer Society, 2017.

[65] W. Zhao, V. Martha, and X. Xu. 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*, pages 862–869. IEEE Computer Society, 2013.

[66] Q. Zhou and J. Wang. SparkSCAN: A structure similarity clustering algorithm on Spark. In *Big Data Technology and Applications*, pages 163–177. Springer, 2016.