Scalable Interactive Dynamic Graph Clustering on Multicore CPUs

Son Mai, Sihem Amer-Yahia, Ira Assent, Mathias Skovgaard Birk, Martin Storgaard Dieu, Jon Jacobsen, Jesper Kristensen

Scalable Interactive Dynamic Graph Clustering on Multicore CPUs

Son T. Mai, Sihem Amer-Yahia
CNRS, University of Grenoble Alpes, France
Email: {mtson, sihem.amer-yahia}@univ-grenoble-alpes.fr
Ira Assent, Mathias Birk, Martin Storgaard Dieu, Jon Jacobsen, Jesper Kristensen
Department of Computer Science, Aarhus University, Denmark
Email: ira@cs.au.dk

Abstract—The structural graph clustering algorithm SCAN is a fundamental technique for managing and analyzing graph data. However, its high runtime remains a computational bottleneck, which limits its applicability. In this paper, we propose a novel interactive approach for tackling this problem on multicore CPUs. Our algorithm, called anySCAN, iteratively processes vertices in blocks. The acquired results are merged into an underlying cluster structure consisting of the so-called *super-nodes* for building clusters. During its runtime, anySCAN can be suspended for examining intermediate results and resumed for finding better results at arbitrary time points, making it an *anytime* algorithm which is capable of handling very large graphs in an interactive way and under arbitrary time constraints. Moreover, its block processing scheme allows the design of a scalable parallel algorithm on shared memory architectures such as multicore CPUs for speeding up the algorithm further at each iteration. Consequently, anySCAN uniquely is a both *interactive* and *work-efficient* parallel algorithm. We further introduce danySCAN an efficient *bulk* update scheme for anySCAN on dynamic graphs in which the clusters are updated in bulks and in a parallel interactive scheme. Experiments are conducted on very large real graph datasets for demonstrating the performance of anySCAN. They show its ability to acquire very good approximate results early, leading to orders of magnitude speedup compared to SCAN and its variants. Moreover, it scales very well with the number of threads when dealing with both static and dynamic graphs.

Index Terms—Structural graph clustering, SCAN, anytime clustering, parallel algorithm, multicore CPUs, dynamic graphs.

1 Introduction

Given a graph $G = (V, E)$, where V is a set of vertices and E a set of edges, graph clustering algorithms group vertices in V so that those in the same group are highly connected and there are few connections among different groups. They have many applications, e.g., finding communities of people in social networks or detecting hidden structures in graphs. During the last decades, many techniques have been introduced such as modularity-based methods [1] and graph partitioning [2]. Among these techniques, the structural graph clustering algorithm SCAN [3] is not only able to discover clusters but also hubs connecting several clusters and outliers. SCAN, however, requires evaluating all $O(|E|)$ structural similarities for all pairs of adjacent vertices. For very large graphs, these overheads obviously are a computational bottleneck that limits applicability. Enhancing the performance of SCAN is thus an important task and is currently attracting considerable research efforts, e.g., [4]–[9]. However, many challenges still remain. For example, how can we produce clusters under arbitrary time constraints? How can we provide user interaction during the clustering process? How can we design a parallel algorithm that scales well with multiple threads while still efficient enough compared to state-of-the-art sequential techniques under single thread usage? Or, how can we efficiently deal with changes in dynamic graphs?

Contributions. In this paper, we focus on the problem of speeding up SCAN for very large graphs and deal with all questions above. Our algorithm, called *anytime SCAN* (anySCAN), has some unique properties described below.

First, anySCAN, as an *anytime* technique, quickly produces an approximate result and then iteratively refines it during its execution. Thus, while it is running, users can suspend it for examining intermediate results and resume it again at any time, until a satisfactory result is reached. Obviously, this *interactive* scheme of anySCAN is very useful for coping with very large graphs under arbitrary limited time constraints. Though anytime algorithms have been widely employed for time consuming problems in many fields, e.g. [10]–[12], no anytime variant of SCAN exists.

Second, anySCAN is the first parallel extension of SCAN specifically designed for shared memory architectures such as multicore CPUs. By maintaining an underlying cluster structure consisting of the so-called *super-nodes* and processing vertices in blocks to connect these super-nodes to form clusters, anySCAN significantly reduces synchronization among threads, thus making it a scalable parallel algorithm. Combined with its *anytime* property, anySCAN is an unique technique that can exploit multiple threads for approximating the results of SCAN as well as produce the exact results of the algorithm SCAN faster.

Third, anySCAN is a *work-efficient* anytime and parallel method. Concretely, an anytime version of an algorithm usually ends up being slower if run to the end than the original algorithm due to additional cost for maintaining

the anytime properties. Similarly, a parallel algorithm aims to ensure high throughput to better utilize all its threads. Typically high throughput comes at the cost of increased overall workload, making it slower on a single thread than sequential techniques. In contrast to these techniques, by examining the current cluster structure at each iteration and calculating the structural similarity only when it is necessary, anySCAN reduces redundant calculations and is thus a *work-efficient* method, i.e., its final cumulative runtimes are much faster than SCAN even on a single thread.

Extensions. In our previous work [13], we only consider static graphs. However, many real-world graphs such as social networks (e.g., Facebook or Twitter), communication networks, or biological networks are subject to frequent changes. For example, an edge is added or removed when two persons are connected or disconnected in Facebook, leading to changes in communities. These changes are even more volatile when considering actions taken in the network (such as sharing particular types of content) to identify communities. In these cases, the graph clustering should be updated immeditately to reflect these changes, instead of reclustering from scratch.

When graphs evolve over time, all existing techniques for SCAN update clusters in a batch mode for each change [8], which may not be efficient when the number of changes is large due to redundant calculations. In contrast, our algorithm, called danySCAN, processes changes in a bulk mode for multiple updates at once. It thereby achieves performance improvements as well as admits an interactive parallel processing scheme for updating clusters. To the best of our knowledge, danySCAN is the first dynamic clustering algorithm with interactive parallel bulk update for SCAN.

Experiments are conducted on very large graphs with up to 1.3 billion edges to demonstrate the performance of our algorithms. They are an order of magnitude faster than SCAN and its variants. Moreover, in its parallel mode, they scale very well with the number of threads on both static and dynamic graphs.

2 Some Basic Notions

The algorithm SCAN. SCAN [3] is originally designed for clustering undirected and unweighted graphs. Here, we study weighted graphs, which are more general and have wider applicability. Thus, we first extend the notion of SCAN to work with weighted graphs.

We are given an undirected and weighted graph $G = (V, E, W)$, where V, E, W are sets of vertices, edges, and their weights of G, respectively. Let N_p be the set of adjacency vertices of a vertex p, and $w_{pq} \in W$ be the weight of the edge (p, q). We extend the unweighted structural similarity notion of SCAN into a weighted one as follows.

Definition 1. The weighted structural similarity between two vertices p and q is defined as:

$$\sigma(p,q) = \Big(\sum_{r \in N_p \cap N_q} w_{pr} \cdot w_{qr}\Big) \Big/ \sqrt{\Big(\sum_{r \in N_p} w_{pr}^2\Big) \cdot \Big(\sum_{r \in N_q} w_{qr}^2\Big)}$$

Generally, $\sigma(p, q)$ indicates how strong the two vertices influence each other through their shared neighbors. The structural similarity of SCAN is a special case of Definition 1 where all the edge weights are 1. Similar to [3], $\sigma(p, q)$ can be calculated in $O(|N_p| + |N_q|)$ time following the sort-merge join style [5] or $O(min(|N_p|, |N_q|))$ if a hash structure is employed [5] since the length $l_p = \sum_{r \in N_p} w_{pr}^2$ of a vertex p is fixed and can be easily calculated in a preprocessing step in $O(|N_p|)$ time. Given the two parameters $\mu \in \mathbb{N}^+$ and $\epsilon \in \mathbb{R}^*$, the cluster notions of SCAN can be extended by using the weighted structural similarity in Definition 1.

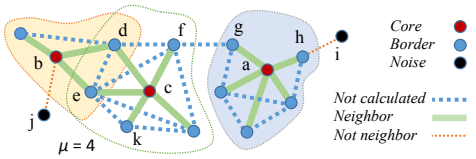


Fig. 1. Basic idea of anySCAN

Definition 2. The structural neighborhood of a vertex p (N_p^ϵ) is defined as $N_p^\epsilon = \{q \mid q \in N_p \wedge \sigma(p, q) \geq \epsilon\}$.

Definition 3. A vertex p is called a core vertex, denoted as $core(p)$, if $|N_p^\epsilon| \geq \mu$. If p has less than μ neighbors but one of its neighbors is a core vertex, then it is called a border (denoted as $border(p)$). Otherwise it is called noise (or outlier) (denoted as $noise(p)$).

Definition 4. A vertex p is directly density-reached from q, denoted as $p \triangleleft q$, if $core(q)$ and $p \in N_q^\epsilon$. Two vertices p and q are density-connected, denoted as $p \bowtie q$, if there exists a chain of vertices x so that $p \triangleleft x_1 \cdots \triangleleft x_i \triangleright \cdots \triangleright x_n \triangleright q$.

Definition 5. A cluster in SCAN is defined as a maximal set of vertices that are density-connected from each other.

SCAN builds clusters by randomly starting from an unprocessed core vertex p, finding its neighbors q, and expanding clusters by examining q's neighbors until all vertices are processed. Its time complexity is $O(min(|N_p|, |N_q|) \cdot |E|)$ and it is *worst-case optimal* [5].

Anytime algorithms. Anytime algorithms are widely used to cope with time consuming tasks in many fields, e.g., [10], [12], [14]–[16]. In contrast to *batch* algorithms, *anytime* ones can be interrupted to provide a *best-so-far* result and then resumed to produce better results at any time, thus allowing interactions with users during execution.

3 The Algorithm Anytime SCAN

Figure 1 illustrates the basic idea of anySCAN. It processes vertices such that the cluster structure quickly emerges and is refined without naively checking all vertices. Assume that with the first five neighborhood checks on a, b, c, i, and j, we know that a, b, and c are core vertices, and i and j are noise. Following Definition 4 and 5, all other vertices now belong to at least one cluster, e.g., g and h must be in the same cluster with a. AnySCAN then looks for true clusters using this information as a guideline. If it chooses d and discovers that $core(d)$, then b, c and all their neighbors (N_b^ϵ and N_c^ϵ) must belong to the same cluster. Next, it examines f and sees that f is not a core, then N_a^ϵ and N_c^ϵ will surely belong to different clusters. It can then safely stop the algorithm and have the same clustering result as SCAN without further examining the rest of the vertices, since the clusters will not change anymore regardless of additional neighborhood checks. This saves many structural similarity

```
1  Function anySCAN (G, μ, ε, α, β)
2  BeginFunction
3    /* Step 1: Summarization */
4    while there still exist untouched vertices do
5      select a set X of α untouched vertices for examining
6      for all vertex p in X do
7        perform the range query on p and mark the state of p
8        if p is a core then
9          for all vertex q in N_p^ε do
10           mark the state and increase the number of neighbors of q
11           add sn(p) to the list of super nodes SN
12           if q is unprocessed-core or processed-core then
13             get the list SN_q of super nodes containing q
14             Union(sn(p), sn(g)), where g ϵ SN_q
15       else
16         for all vertex q in N_p^ε do
17           increase the number of neighbors nei(q) of q
18         add sn(p) to the noise list L
19   /* Step 2: Merging strongly-related super-nodes */
20   build the set S of shared unprocessed-border vertices
21   sort S in the descending order of the numbers of super-nodes
22   while S is not empty do
23     remove a set X of β fist vertices for examining
24     for all vertex p in X do
25       check if p should be pruned from the core check
26       perform the core check on p
27       if p is not a core then continue
28       get the list of super nodes SN_p containing p
29       for i = 0 to |SN_p| − 1 do
30         let sn(u) and sn(v) be super-nodes of p at i and i + 1 of SN_p
31         if Findset(sn(u)) ≠ Findset(sn(v)) then
32           Union( Findset(sn(u)), Findset(sn(v)))
33   /* Step 3: Merging weakly-related super-nodes */
34   find the representative super-nodes for all vertices
35   build the set T of unprocessed-border or core vertices
36   sort T in the descending order according to the vertex degrees
37   while T is not empty do
38     remove a set X of β fist vertices for examining
39     for all vertex p in X do
40       check if p should be pruned from the core check
41       perform the core check on p
42       if p is not a core then continue
43       for all vertex q in N_p do
44         if q is not a core then continue end if
45         if Findset(clu(p)) ≠ Findset(clu(q)) then
46           if σ(p, q) ≥ ε then
47             Union( Findset(clu(p)), Findset(clu(q)))
48   /* Step 4: Determining border vertices */
49   for all vertex p in L do
50     check if p is a border or a true noise
51 EndFunction
```

Fig. 2. Pseudocode for anySCAN

calculations and reduces runtime. Moreover, anySCAN can be interrupted after any neighborhood checks for producing approximate results of SCAN. Besides that, if it processes a *block* of vertices (with sizes α and β) each time instead of a single vertex like SCAN and its variants [3]–[5], each neighborhood query can be handled by a thread independently before being used to produce clusters, thus opening a way for designing a scalable parallel technique on shared memory architectures such as multicore processors.

3.1 Anytime SCAN

Concretely, anySCAN is built upon the concepts: summarization, selection, merging, and block processing which are summarized in Figure 2.

Step 1: Summarization. Step 1 (Line 3-18) summarizes vertices into homogeneous groups called *super-nodes*, which will be exploited to build clusters quickly in the next steps.

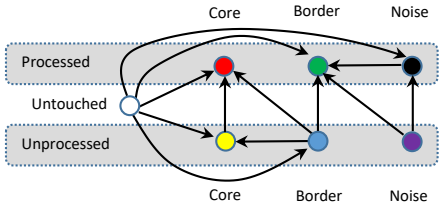


Fig. 3. The state transition schema for vertices

Assume that all vertices have *untouched* states in the beginning. We repeatedly and randomly choose an *untouched* vertice p for examining until there are no more *untouched* ones. If $noise(p)$ (Line 15-18), we mark it as *processed-noise* and store N_p^ϵ in a noise list L for a post processing step. If $core(p)$ (Line 8-14), N_p^ϵ is summarized in a *super-node* with p as a representative (denoted as $sn(p)$) and stored in the super-node list SN for further processing. We update the states of p to *processed-core* and its neighbors q to *processed-border* (if q was noise), or *unprocessed-border* (if q is not examined), or *unprocessed-core* (if q is untouched and is known to have more than μ neighbors)[1] following the vertex transition state schema in Figure 3 described below.

Figure 3 summarizes the state transition for all vertices during the execution of anySCAN. For example, if $|N_p| < \mu$, then we know that p is surely not a core without having to examine its neighbors. Thus $state(p)$ is *unprocessed-noise*. If we know that p is a neighbor of a core vertex q, then p will be a border of a cluster. Thus its state is changed to *processed-border*. If none of its neighbors is core, p is assigned *processed-noise* state. If an *unprocessed-border* object p has more than μ neighbors, it is surely a core and is changed to *processed-core*. Otherwise if it has fewer than μ neighbors, it is changed to *processed-border* since it already belongs to a cluster. If an *unprocessed-border* vertex p is not examined but p has more than μ neighbors then it is surely a core and is assigned *unprocessed-core* state. A *border* vertex will never become a *core*. A *core* vertex will not change to a *border* or a *noise* one. A *processed* vertex will not change to *unprocessed*.

Lemma 1. The state of each vertex in anySCAN changes according to the transition schema in Figure 3.

Lemma 1 can be verified through Definition 3 and 4 of SCAN. As demonstrated in Figure 1, after the summarization step, we have three super-nodes $sn(a)$, $sn(b)$, and $sn(c)$. Noise list L contains two vertices i and j. All other vertices are not examined and marked as *unprocessed-border*.

Lemma 2. All objects inside a super-node $sn(p)$ belong to the same cluster.

Lemma 2 is directly inferred from Definition 4. Following it, we only need to label all the super-nodes instead of labeling all vertices like SCAN. Since the number of super-nodes is much smaller than the number of vertices, the label propagation time is reduced. To do so, each super-node is initially placed in a single cluster. If we discover that $sn(p)$ and $sn(q)$ must belong to the same cluster, we merge them (Line 12-14). Here, a Disjoint-set data structure [17] is employed to keep track of the labels of super-nodes. It supports two operations including (1) *Findset*: for finding

1. To do so, we additionally store for each object q the number of neighbors it currently has, denoted as $nei(q)$.

which subset a particular super-node is in and (2) *Union*: for merging two subsets of super-nodes into a single subset.

At any time (e.g., after each iteration of anySCAN), the intermediate clustering result of anySCAN can be acquired by labeling a vertex according to the label of its super-node.

Step 2: Merging strongly-related super-nodes. In Step 2 (Line 19-32), anySCAN merges super-nodes to form clusters starting with ones that are more likely to be in the same cluster, e.g., vertices d and e in Figure 1.

Definition 6. Two super-nodes $sn(p)$ and $sn(q)$ are called *strongly-related* to each other if they share some vertices, i.e., $sn(p) \cap sn(q) \neq \emptyset$.

Intuitively, if $sn(p)$ and $sn(q)$ are strongly-related, they have a high chance to be in the same cluster. Thus, in this step, we will examine all pairs of strongly-related super-nodes to see if they should be merged together.

Lemma 3. If there exists a core vertex $u \in sn(p) \cap sn(q)$ (either in processed or unprocessed state), $sn(p)$ and $sn(q)$ belong to the same cluster.

Proof 1. Let a and b be two arbitrary vertices in $sn(p)$ and $sn(q)$, respectively. We have $a \triangleleft p$ and $q \triangleright b$ (Definition 4). Since $core(u)$ and $u \in N_p^\epsilon \cap N_q^\epsilon$, $p \triangleleft u$ and $u \triangleright q$, a and b are density-connected according to Definition 4.

Lemma 3 states that if two super-nodes share a core vertex, they are merged. We first collect a set S of all *unprocessed-border* vertices that belong to at least two super-nodes (Line 20), e.g., vertices d and e in Figure 1. The rest can be safely ignored since they do not help to determine the new connection of two super-nodes (*cores* has been processed in Step 1). We sort all vertices in S in descending order of the numbers of super-nodes they belong to (Line 21). Then, each vertex is extracted and processed to merge super-nodes until S is empty.

For each vertex p, if all its super-nodes already belong to the same cluster, we do not need to examine p anymore since it will not lead to any change in the result (Line 25). Otherwise, if p is an *unprocessed-border* vertex, we need to check if it is a core (Line 26). To do so, we explore its adjacency vertices $q \in N_p$ until we know that p is a core, i.e., it has more than μ neighbors, instead of calculating all structural similarities around p. This helps to reduce the runtime. If p is a core, we set its state to *unprocessed-core*. Otherwise, it is changed to *processed-border* vertex following the transition schema in Figure 3. Note that, we still mark the states for neighbors of p as in Step 1 and 3. Now, if p is a core, all of its super-nodes $g \in SN_p$ belong to the same cluster due to Lemma 3. Hence, anySCAN calls at most $|SN_p|-1$ *Union* operations for merging them together where SN_p is the set of super-nodes that contains p (Line 29-32). Here sorting (Line 21) can help to reduce the number of core checks since many super-nodes will be merged earlier.

Figure 1 shows an example of Step 2. Only d and e need to be examined. We first examine d and see that it is a core. Then $sn(b)$ and $sn(c)$ are merged into one cluster. Now, e can be ignored since all of its super-nodes have the same label. However, to finally conclude that the results are completely identical to SCAN, we need to check whether $sn(a)$ and $sn(c)$ should be merged in Step 3.

Step 3: Merging weakly-related super-nodes. Step 3 (Line 33-47) verifies connections that cannot be discovered in Step 2, e.g., $sn(a)$ and $sn(c)$.

Definition 7. Two super-nodes $sn(p)$ and $sn(q)$ are *weakly-related* if there exist two vertices $u \in sn(p)$ and $v \in sn(q)$ so that $u \in N_v$ and $v \in N_u v$, i.e., u and v are adjacent.

Lemma 4. If there exist two core vertices $u \in sn(p)$ and $v \in sn(q)$ so that u and v are adjacent (i.e., $(u, v) \in E$) and $\sigma(u, v) \geq \epsilon$ then two super-nodes $sn(p)$ and $sn(q)$ belong to the same cluster.

Proof 2. Let a and b be two arbitrary vertices in $sn(p)$ and $sn(q)$, respectively. We have $a \triangleleft p$ and $q \triangleright b$ (Definition 4). Since $core(u)$ and $core(v)$ and $\sigma(u, v) \geq \epsilon$, we have $u \bowtie v$, $p \triangleleft u$, and $v \triangleright q$ (Definition 4). Thus, a and b are density-connected (Definition 4).

Identifying the connection between *weakly-connected* super-nodes is much more difficult than for the *strongly-connected* case. Simply examining every pair of super-nodes $sn(p)$ and $sn(q)$, identifying the edge (p, q) that potentially connects them, and then checking their connection is clearly expensive. Therefore, we start with a set T of all *unprocessed-border*, *unprocessed-core*, or *processed-core* vertices (all *processed-border* and $noise$ can be safely ignored due to Lemma 4) (Line 35). We first sort T in descending order of vertex degrees (Line 36). Each vertex is extracted and processed until T is empty. The intuition behind this is that the higher the degree of a vertex p, the more likely it connects more super-nodes. Thus, examining it earlier helps to save core checks for the next ones.

For each vertex p, we scan all of its adjacent vertices q to see if they belong to the same cluster. If so, we can safely skip p since examining p will not lead to any change in the clustering result (Line 40). Otherwise, p may belong to an edge that connects two *weakly-related* super-nodes, and thus needs to be examined. If p is an *unprocessed-border* vertex, we need to check if it is a core one like in Step 2 (Line 41). If p is a core, we set its state to *unprocessed-core*. Otherwise, it is changed to *processed-border* following Figure 3. If p is a core, we again scan through its adjacent vertices $q \in N_p$. If q is not a core, we can skip it. Otherwise, if p and q belong to different clusters (Line 45), we calculate their structural similarity $\sigma(p, q)$. If $\sigma(p, q) \geq \epsilon$, we merge $sn(u)$ and $sn(v)$ by calling the *Union* operation, where $sn(u) = clu(p)$ and $sn(v) = clu(q)$ (Line 46-47), where $clu(p)$ is the cluster that contains p. Note that after Step 2, all *unprocessed-border* or *unprocessed-core* vertices only belong to one cluster.

In Figure 1, when we examine k, all of its adjacent vertices belong to the same cluster. Thus, k is skipped. However, g and adjacent vertex f belong to different clusters. Thus, g has to be checked. In this case, g is not a core. Thus, f and g belong to different clusters.

Step 4: Determining border vertices. Recall that in Step 1, all noise vertices are placed in a noise list L. However, some of them may be border vertices if they are connected to core ones. In Step 4 (Line 49-50), we detect this case. For all vertices $p \in L$, if $state(p)$ is *processed-noise*, we examine all $q \in N_p^\epsilon$. If q is a core either processed or unprocessed, p is a border of $clu(p)$. If q is *unprocessed-border*, we need to check if q is a core and $\sigma(p, q) \geq \epsilon$. If so, p is a border of

$clu(p)$. If p is *unprocessed-noise*, we examine all $q \in N_p$. If q is *processed-core* or *unprocessed-core* and $\sigma(p,q) \geq \epsilon$ then p belongs to the same cluster as q. If q is *unprocessed-border*, we check if q is a core and $\sigma(p,q) \geq \epsilon$. If so, p and q belong to the same cluster. By checking noise at the end, we can exploit previous results such as core vertices to save more on similarity calculation, thus enhancing performance.

3.2 Parallel Anytime SCAN

Block processing. A key idea of anySCAN is processing vertices in blocks. Concretely, in Step 1, we choose α vertices for summarizing at each iteration ($\alpha \gg 1$). Obviously, it increases the overlap between super-nodes. However, this overlap will lead to more super-nodes to be merged at Step 2 and will consequently reduce the number of structure similarity calculations in Step 3. Thus, the performance is improved. In Step 2 and 3, we choose β vertices for updating at each iteration. Though this leads to more redundant similarity calculations, it reduces the overhead of the anytime scheme of anySCAN by reducing the number of iterations.

Block processing also allows threads to perform core checks, the most expensive part of anySCAN, concurrently, thus leading to a *work-efficient* parallel algorithm. Specifically, it maintains the anytime property in parallel mode.

Parallelizing. Figure 4 shows the pseudocode for anySCAN using OpenMP [18]. Since vertices have different neighborhood sizes, we use dynamic scheduling for load balancing, e.g., Line 6, 10, 30, and 34.

Step 1: Summarization. Please note that because two super-nodes may overlap, assigning states for their shared vertices suffers may lead to a race condition, where threads could attempt to write different states for a vertice at the same time. To ensure consistency, a naive approach would need to lock the whole for loop (Line 9-14 Figure 2) for instance. However, this is clearly inefficient since this would sequentialize a large workload. To avoid that, we separate the for loop in Line 6 (Figure 2) into three different parts (Line 6-24 in Figure 4). First (Line 6-9), we calculate the neighborhood of all vertices p and store the results in a temporary buffer B. We also mark the state of p as described in Section 3.1. Obviously, vertices can be processed independently by threads. Second (Line 10-15), for each vertex p, we examine its neighbors q and mark its state if p is a core. If q is a *processed-noise* or *unprocessed-noise*, it is marked as *processed-border*. If q is *unknown*, it is marked as *unprocessed-border*. Now, to check if q is a core, we increase the number of neighbors of q by 1 (q has p as one of its neighbors) using an atomic operation (Line 14-15). If q is an *unprocessed-border* and has more than μ neighbors, it will be marked as *unprocessed-core*. Last (Line 16-24), we use a sequential algorithm for storing the super-nodes as well as merging some of them as described in Section 3.1, since these tasks are highly sequential. However, the overall runtime of this part is negligible and has a small effect on the scalability of anySCAN as demonstrated in Section 5.2.

Step 2: Merging strongly-related super-nodes. As for Step 1, we separate Step 2 into two parts for avoiding performance limiting synchronization. In Line 30-33, we perform the core check on each vertex p in a selected subset of vertices X independently using multiple threads and store the results. In Line 34-42, we check each vertex independently and merge related super-nodes when necessary as described in Section 3.1. The Union operation is not thread-safe. Thus, it must be locked inside critical sections (Line 41). However, since the number of super-nodes is much smaller than the number of vertices in our experiments, the number of Union operations is thus small and does not affect the scalability

```
1  Function anySCAN (G, μ, ε, α, β)
2  BeginFunction
3    /* Step 1: Summarization */
4    while there still exist untouched vertices do
5      select a set X of α untouched vertices for examining
6      #pragma omp parallel for schedule(dynamic)
7      for all vertex p in X do
8        calculate N_p^ε and store it into a buffer B
9        mark the state of p
10     #pragma omp parallel for schedule(dynamic)
11     for all vertex p in X do
12       for all vertex q in N_p^ε do
13         mark the state of q if p is a core
14         #pragma omp atomic
15         increase the number of neighbors nei(q) of q
16     for all vertex p in X do
17       if p is a core object then
18         add sn(p) to the list of super nodes SN
19         for all vertex q in N_p^ε do
20           if q is unprocessed-core or processed-core then
21             get the list SN_q of super nodes containing q
22             Union(sn(p), sn(g)), where g ∈ SN_q
23       else
24         add sn(p) to the noise list L
25   /* Step 2: Merging strongly-related super-nodes */
26   build the set S of shared unprocessed-border vertices
27   sort S in the descending order of the numbers of super-nodes
28   while S is not empty do
29     remove a set X of β fist vertices for examining
30     #pragma omp parallel for schedule(dynamic)
31     for all vertex p in X do
32       check if p should be pruned from the core check
33       perform the core check on p
34     #pragma omp parallel for schedule(dynamic)
35     for all vertex p in X do
36       if p is not a core then continue
37       get the list of super nodes SN_p containing p
38       for i = 0 to |SN_p| − 1 do
39         let sn(u) and sn(v) be super-nodes of p at i and i + 1 of SN_p
40         #pragma omp critical
41         Union( Findset(sn(u)), Findset(sn(v)))
42   /* Step 3: Merging weakly-related super-nodes */
43   find the representative super-nodes for all vertices
44   build the set T of unprocessed-border or core vertices
45   sort T in the descending order according to the vertex degrees
46   while T is not empty do
47     remove a set X of β fist vertices for examining
48     #pragma omp parallel for schedule(dynamic)
49     for all vertex p in X do
50       check if p should be pruned from the core check
51       perform the core check on p
52     #pragma omp parallel for schedule(dynamic)
53     for all vertex p in X do
54       if p is not a core then continue
55       for all vertex q in N_p do
56         if q is not a core then continue end if
57         if σ(p, q) ≥ ε then
58           #pragma omp critical
59           Union( Findset(clu(p)), Findset(clu(q)))
60   /* Step 4: Determining border vertices */
61   #pragma omp parallel for schedule(dynamic)
62   for all vertex p in L do
63     check if p is a border or a true noise
64 EndFunction
```

Fig. 4. Pseudocode for anySCAN using OMP

of anySCAN (see Figure 14 for more details). Note that, the pruning check needs to get the labels for vertices (which in turn is acquired from the labels of super-nodes via the Findset operation). Thus, if we did not split Step 2 into two parts, we would need to lock the Findset operation during the pruning check to ensure consistency. This would incur significant synchronization cost since the Findset operation is used more frequently than the Union operation.

Step 3: Merging weakly-related super-nodes. As for Step 2, we also separate Step 3 into two parts (Line 49 to 61). The first part is for checking the core properties of vertices, and the second part for merging super-nodes using Union.

Step 4: Determining border vertices. Each vertex p inside the noise list L can be examined independently to see if it is really an outlier (Line 63-65). Here some redundant calculation may happen if two noise vertices p and q share an *unprocessed-border* vertex due to the checking process in Step 4. However, this case very rarely happens in experiments. By accepting this, each vertex can be fully processed by a thread without having to wait for the results of other threads. Consequently, the scalability of anySCAN is improved.

3.3 Algorithm Analysis

Correctness. We first prove the correctness of anySCAN.

Lemma 5. The final results of anySCAN are identical to those of SCAN.

Proof 3. Assume that two vertices p and q belong to the same cluster in SCAN. There must exist a chain of core vertices $x_1 \cdots x_n$ so that $p \triangleleft x_1 \cdots \triangleleft x_i \triangleright \cdots \triangleright x_n \triangleright q$ (Definition 4). After Step 1, x_1 to x_n must belong to some super-nodes $sn(s_1) \cdots sn(s_m)$ ($m \leq n$) since only *unprocessed-noise* or *processed-noise* vertices are excluded to the noise list L. In Step 2 and 3, if $sn(s_1)$ to $sn(s_m)$ belong to different clusters, they will be connected by some other vertices or by some $x_i (1 \leq i \leq n)$ themselves (Lemmas 3 and 4). Thus, there exists a density-connected path from x_1 to x_n through $sn(s_1)$ to $sn(s_m)$ (some x_i may not be included in the path because they are not processed). Similarly, Step 4 guarantees that if p and q are in L, they are still density-connected to x_1 and x_n by some paths. Thus, p and q are also density-connected in anySCAN. Note that, in both anySCAN and SCAN, a shared-border vertex may be assigned to different clusters according to the examining order of vertices.

Complexity analysis. Let SN be the set of super-nodes, $T_{pq} = min(|N_p|, |N_q|)$ be the time for computing $\sigma(p, q)$, and c a constant. The time complexity of Step 1 is $O(\sum_{sn(p) \in SN \cup L}(\sum_{q \in N_p} T_{pq} + \sum_{q \in N_p} c))$ for calculating neighborhoods and marking the states of vertices. We have $O(\sum_{q \in C} |N_q^\epsilon|)$ Union-Find operations (since $SN(q) \subset N_q^\epsilon \subset N_q$ and each merge is done on new super-nodes only), where $C \subset V$ is the set of *core* vertices. Similarly, Step 2 and 3 need $O(\sum_{p \in S}(\sum_{q \in N_p} + \sum_{q \in N_p} T_{pq}))$ time for core checks and neighborhood calculations and $O(\sum_{p \in S} |N_p|)$ Union-Find operation. Step 4 needs $O(\sum_{p \in L} \sum_{q \in N_p} T_{pq})$ time for checking border states. Obviously, all of Steps 1 to 4 are bounded by $O(|E|T_{pq})$ neighborhood calculations and $O(|E|)$ Union-Find operations. Sorting requires $O(|V|)$ time using bin-sort since the values are all integer and are bounded by the vertex degrees. Thus, at the end, anySCAN has the same time complexity as pSCAN and SCAN with $O(min(N_p, N_q)|E| + |E|f(|V|))$, where $f()$ is the extremely slowly growing inverse of the single-valued Ackermann function [5], [17]. The algorithm anySCAN needs to store the super-node list SN as well as the noise list L. Thus, it incurs additional memory usage compared to SCAN. However, the overall size is still bounded by $O(|E|)$. Thus, in the end, it has $O(|V| + |E|)$ space complexity.

3.4 Optimizations

We introduce some optimization techniques to speed up the similarity calculation of anySCAN inspired by [5].

Lemma 6. Given two vertices p and q, if $\hat{\sigma}(p, q)^2 < \epsilon^2 \cdot l_p \cdot l_q$, where $\hat{\sigma}(p, q) = min(|N_p|, |N_q|) \cdot max(w_p, w_q)$, and $w_p = max_{q \in N_p}(w_{pq})$, and $l_p = \sum_{r \in N_p} w_{pr}^2$, then $\sigma(p, q) < \epsilon$.

Proof 4. We have $\sum_{r \in N_p \cap N_q} w_{pr} \cdot w_{qr} \leq min(|N_p|, |N_q|) \cdot max(w_p, w_q) < \epsilon \cdot \sqrt{l_p \cdot l_q}$. Thus $\sigma(p, q) = (\sum_{r \in N_p \cap N_q} w_{pr} \cdot w_{qr}) / \sqrt{l_p \cdot l_q} < \epsilon$.

The equation $\hat{\sigma}(p, q)^2 < \epsilon^2 \cdot l_p \cdot l_q$ can be verified in $O(1)$ time (since w_p and l_p are fixed and can be calculated in a preprocessing step in $O(|N_p|)$ time). If it is *true*, we do not need to calculate the structural similarity $\sigma(p, q)$. Moreover, while calculating $\sum_{r \in N_p \cap N_q} w_{pr} \cdot w_{qr}$, if the intermediate result is bigger than $\sqrt{\epsilon^2 \cdot l_p \cdot l_q}$, then $\sigma(p, q)$ is bigger than ϵ. Thus, we can stop for further reducing runtime.

4 Dynamic Graph Clustering

Given a weighted dynamic graph G, possible changes in G are deletion, insertion and weight change on each edge. A naive approach would perform clustering from scratch, which is obviously very expensive. Here, we focus on an efficient incremental approach. Let $U = \{(u, v, w, t)\}$ be a set of changed edges (u, v, w), where t is the update type including insertion, deletion and change weight. Instead of updating clusters with each change [8], [19], [20], our method, called dynamic anySCAN (danySCAN), operates in a bulk scheme, especially when the number of updates $|U|$ is large. For simplicity, we skip w and t whenever it is clear from the context.

4.1 Dynamic Anytime SCAN

The general idea of danySCAN is that we exploit the existing super-node structure for incrementally updating clusters without having to start from scratch. We first present the general process of danySCAN with arbitrary graph updates before going into the optimization procedure of each special kind of update operations.

Definition 8. Given a set of updates U, the set of directly affected vertices A is defined as a set of vertices in U.

Definition 9. Given a set of updates U, the set of *affected edges* B is defined as a set of all edges (u, v) where $u \in A$ and/or $v \in A$.

Definition 10. The set of vertices A^k is called the *k-affected* of A if it consists of all vertices that are k-hop-away from any vertices in A.

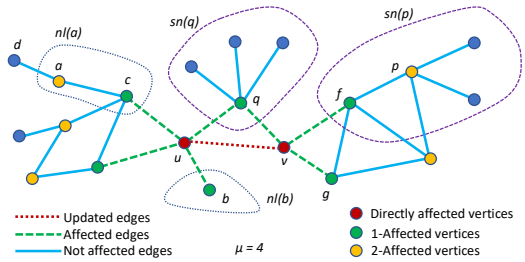


Fig. 5. Affected vertices, edges, super-nodes and noises w.r.t. a changed edge $(u, v) \in U$

Lemma 7. All edges (u, v) in B will change their structural similarities after the update.

Lemma 7 is directly inferred from Definition 1. Following it, all *1-affected* vertices in A^1 may change their core properties since their structural neighborhoods will be changed. Conversely, all edges $e \in E \setminus B$ will not be affected. Their similarities remain the same after the updates. Similarly, all vertices $v \in V \setminus A^1$ will not change their core properties.

Figure 5 shows an example when we change (e.g., delete) an edge (u, v). The structural similarity $\sigma(u, c)$ will be changed. Thus, vertex c may change its core properties. However, vertices a and d are not affected at all. Their core properties remain the same. Each super-node $sn(q)$ where $q \in A^1$ will also be affected by the updates following the similarity changes of $e \in B$. Some vertices may be removed from $sn(q)$ or inserted into $sn(q)$, e.g., u or v. If $|sn(q)| < \mu$, the super-node must be removed since q is not a core vertex anymore. We call $sn(q)$ a *directly affected super-node*. For $sn(p)$ where $p \in A^2 \setminus A^1$, since N_p^ϵ is not affected, its members will not change. However, since vertex $f \in sn(p)$ might change its core property, it will affect the connectivity between $sn(p)$ and its nearby super-nodes, thus leading to changes of current clusters. We call $sn(p)$ an *affected super-node*. All other super-nodes outside the *2-affected* set will not affect the cluster structure at all since all of their vertices do not change after the updates. Similarly, a noise vertex $nl(b) \in L$ is called a *directly affected noise* if $b \in A^1$ since its members can be changed into core vertex or border vertex of another cluster. A noise object $nl(a)$ is an *affected-noise* if $a \in A^2 \setminus A^1$. Even though its members remain the same, it may be assigned to another cluster following the change of the core property of one of its members, e.g., vertex c.

The algorithm danySCAN consists of several steps as shown in Figure 6 and described below. Generally, it will update the super-node structure w.r.t. updates before merging them again to build clusters.

Step 1. Updating the super-node structures. When changes occur, we need to rebuild the list of super-nodes and the noise list L as well as their relationships.

First, we need to update the graph G to reflect the changes and then recalculate the lengths of vertices for speeding up similarity evaluations (Line 4-5 in Figure 6).

Second, we need to check the state of each affected edge $e \in B$ (Line 6-8). To do so, we assign for each edge $(u, v) \in E$ a state including: *yes* if $\sigma(u, v) \geq \epsilon$, *no* if $\sigma(u, v) < \epsilon$ and *unknown* if $\sigma(u, v)$ has not been calculated yet. Now, for simplicity, we can set all edges $e \in B$ to the *unknown* state, indicating that their similarities are changed

```
1  Function danySCAN (G, μ, ε, α, β, U)
2  BeginFunction
3    /* Step 1: Updating the super-nodes */
4    #pragma omp parallel for schedule(dynamic)
5    Update the graph G and recalculate the lengths l of vertices
6    #pragma omp parallel for schedule(dynamic)
7    for all (u, v) in B do
8      set state of σ(u, v) to unknown (if it may change)
9    #pragma omp parallel for schedule(dynamic)
10   for each directly affected super-node sn(q) do
11     update and mark the states of its members if q is a core
12     otherwise put q into the noise list L
13   #pragma omp parallel for schedule(dynamic)
14   for each directly affected noise nl(q) do
15     update its members and check if q becomes a core
16     if so mark the states of its neighbors and
17       put q to the super-node list
18   #pragma omp parallel for schedule(dynamic)
19   find all orphan vertices and
20     build new super-nodes in parallel like anySCAN
21   find all affected clusters and put their super-nodes
22     into the affected super-node list
23   recreate the Disjointsets data structure
24   for each non-affected clusters C do
25     merge its super-nodes using Union operation
26   build the set of examined vertices A*
27   /* Step 2: Merging strongly-related super-nodes */
28   build the set S of unprocessed vertices that belong to
29     at least two super nodes from the examined set A*
30   merge super-nodes in parallel like anySCAN
31   /* Step 3: Merging weakly-related super-nodes */
32   find the representative super-nodes for all vertices
33   build the set T of unprocessed-border or core vertices
34     from the examined set A*
35   merge super-nodes in parallel like anySCAN
36   /* Step 4: Determining border vertices */
37   #pragma omp parallel for schedule(dynamic)
38   for all affected noises in L do
39     check if p is a border or a true noise
40 EndFunction
```

Fig. 6. Pseudocode for dynamic anySCAN

and need to be recalculated. However, for efficiency, we can use some optimization techniques for different kinds of update operations as presented in Section 4.2.

Third, we set the states of all vertices $p \in A$ and $p \in sn(q)$, where $sn(q)$ is a *directly affected super-node* to *unknown*. Then, for each *directly affected super-node* $sn(q)$, we update its members by adding a new vertex r if $\sigma(q, r) \geq \epsilon$ or removing r if $\sigma(q, r) < \epsilon$. If $|sn(q) < \mu$, we delete $sn(q)$ from the list of super-nodes and move it to the noise list L. Otherwise, we mark the state of $p \in sn(q)$ following the transition state diagram in Figure 3 like for Step 1 of anySCAN (Line 9-12). Similarly, for each *directly affected noise* vertex $nl(q) \in L$, we update its members. If q is a core, we put $sn(q)$ into the super-node list and mark the states of its neighbors accordingly (Line 13-17). At the end of this stage, if a vertex q is still an orphan (i.e., $state(q) = unknown$), we check if it actually belongs to some unaffected super-nodes. If so, q is marked as *unprocessed-border*.

Fourth, we build new super-nodes and new noise vertices over orphan vertices in a similar way to the summarization step of anySCAN described in Section 3.1, which we will not repeat here for clarity, until there are no more *unknown* ones (Line 19-20). After that we further extend the sets of *affected super-node* and *affected noises* by adding newly generated ones to them.

Fifth, in the worst case, an *affected super-node* $sn(p)$ may lose its connections to other super-nodes due to the

changes of core properties of its members or one of its *yes* connections to others is broken. This may split its cluster into smaller groups. Thus, this cluster needs to be re-verified to identify sub-clusters. Note that, we also have merge cases where clusters are merged into a larger one. But, these cases are easier due to the merging nature of anySCAN as presented in Section 3.1. Here a cluster C is an *affected cluster* if one of its super-nodes is *affected*. For each *affected cluster*, we put its super-nodes into the *affected super-node* list for rebuilding sub-clusters later (Line 21-22).

Seventh, we need to rebuilt the cluster structure if there is a split case. To do so, we recreate a new Disjointsets (Line 23). Then, for each non-affected cluster C with c super-nodes, we merge them by calling the Union operation exactly $c-1$ times for each adjacent pair for rebuilding C in the new Disjointsets (Line 24-25). Note that since the number of super-nodes is much smaller than the number of vertices, this task thus has negligible effect on the performance of danySCAN. Now we need to identify a special set of vertices A^* called *examined vertices* as follows before fully building final clusters in the next steps of danySCAN (Line 26).

Definition 11. The set of *examined vertices* A^* consists of all vertices in $\cup sn(p)$, where $sn(p)$ is an *affected super-node* and all vertices in A^2.

Since we need to merge *affected super-nodes* to form clusters in the next steps, it is straightforward to see that we only need to examine vertices in A^* for efficiency. A^2 ensures that super-nodes are merged due to the update. And $\cup sn(p)$ guarantees that if a cluster is not broken, it will be recreated correctly. When the number of updates is large, $A^* \approx V$. Thus, we can choose $A^* = V$ for simplicity.

Step 2: Merging strongly-related super-nodes. This step (Line 27-30) works in a similar way to anySCAN with some major modifications: (1) we only consider objects in the examined set A^* and (2) we consider both *unprocessed-border* and *unprocessed-core* since we have not processed *unprocessed-core* in Step 1 (Line 27-30).

Step 3: Merging weakly-related super-nodes. This step (Line 31-35) also works exactly as in anySCAN with a restriction on the examined set A^* (Line 31-35).

Step 4: Determining border vertices. It is easy to prove that if a noise vertex $p \in L$ is not an *affected noise*, it will never be changed. Note that if p is a shared border vertex, it may be assigned different labels according to the examining order. Thus, we only take out all *affected noises* and examine them using the same procedure of anySCAN.

4.2 Optimization techniques for danySCAN

The above procedure of danySCAN works regardless of the update types. However, for efficiency, we can reformulate the second phase of Step 1 to cope with different update types by exploiting some similarity check as in [8]. For example, if an edge (u, v) is inserted, for each vertex $w \in N_u \setminus N_v$, $\sigma(w, u)$ will be decreased. Thus, if $state(u, v) = no$, it will never change to *yes*. Thus, we do not need to re-evaluate the structural similarity between u and v anymore. We can update edge states in Step 1 in two ways: (1) sequentially with each updates which incurs redundant calculations when several updated edges share a vertex; or (2) check all *affected vertices* in chunks for avoiding shared vertices. In the latter case, we must modify the similarity check of [8] for ensuring the correctness of the algorithm by only considering $w \in N_u \setminus N_v$ if w is not an *affected vertex* as in Lemma 8 below. However, compared to (1), (2) may have weaker pruning power in case we have big chunks.

Graph	Vertices	Edges	$\overline{d}$	c
Ego-Gplus (GR01)	107,614	13,673,453	127.06	0.4901
Soc-LiveJournal1 (GR02)	4,847,571	68,993,773	14.23	0.2742
Soc-Poket (GR03)	1,632,803	30,622,564	18.75	0.1094
Com-Orkut (GR04)	3,072,441	117,185,083	38.14	0.1666
Kron_g500-logn21 (GR05)	2,097,152	182,082,942	86.82	0.1649
Twitter (GR06)	41,652,230	1,369,000,750	32.8	0.0730

TABLE 1
REAL GRAPH DATASETS ($\overline{d}$ IS AVERAGED VERTEX DEGREES AND c IS AVERAGED CLUSTER COEFFICIENTS)

Lemma 8. Given an inserted edge (u, v), for each vertex $w \in N_u \setminus N_v \wedge N_w \cap A = \emptyset$, $\sigma(u, w)$ will be decreased.

Proof 5. Since $N_w \cap A = \emptyset$, its length l_w is not changed while l_u is increased due to the insertion. Thus, $\sigma(u, w)$ is decreased following Definition 1.

Here, we introduce two more cases to cope with weighted graphs: (1) increase weight which is similar to the edge insertion and (2) decrease weight which is similar to the edge deletion in [8]. The detailed extension for all update types is straightforward but lengthly, and thus is omitted.

4.3 Dynamic AnySCAN on Multicore CPUs

The extension of danySCAN for multicore CPUs is exactly the same as anySCAN with the block processing scheme in the fourth phase of Step 1, Step 2 and 3. The pseudocode of danySCAN can be found in Figure 6.

4.4 Algorithm Analysis

Correctness. We first prove the correctness of danySCAN.

Lemma 9. The final results of danySCAN are identical to those of SCAN.

Proof 6. In Step 1, danySCAN checks each super-node to add or remove members. Each new super-node p is also built by a neighborhood query. Thus, this guarantees that $\forall sn(p) : sn(p) = N_p^\epsilon$. Step 1 also ensures that none of the vertices in L is actually a core. Consequently, if two vertices p and q are density-connected in SCAN by a chain of core vertices $x_1 \cdots x_n$ (Definition 4), all x_i must belong to some super-nodes $sn(s_j)$ after Step 1 of danySCAN. At the end of Step 1, all possibly changed connections among super-nodes are removed to check them again in Step 2 and 3. Following Lemmas 3 and 4, all $sn(s_j)$ will be put into the same cluster via x_i or other core vertices. Due to Lemma 2, p and q are connected via a chain going through $sn(s_j)$ (including s_j itself). Step 4 guarantees that if a border vertex is added to the noise list L initially, it is surely discovered as a border here. Thus, the results of danySCAN and SCAN are identical.

Complexity analysis. The time complexity of danySCAN is similar to that of anySCAN, pSCAN and SCAN. It also consumes $O(|E| + |V|)$ space like others.

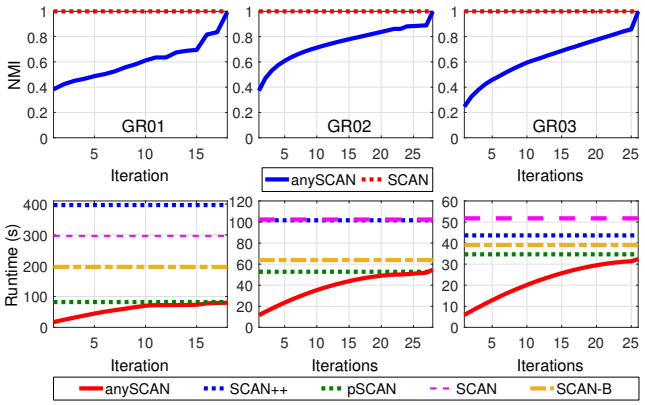


Fig. 7. NMI scores and cumulative runtimes of anySCAN during its execution in comparison with other *batch* algorithms (represented by horizontal lines) for GR01 to GR03

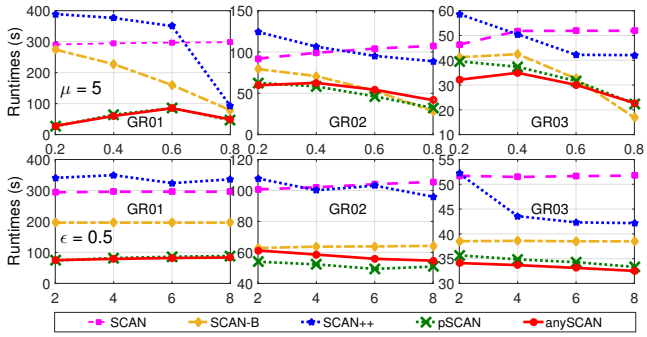


Fig. 8. Final runtimes of different algorithms w.r.t. parameters ϵ (top) and μ (bottom) for GR01 to GR03

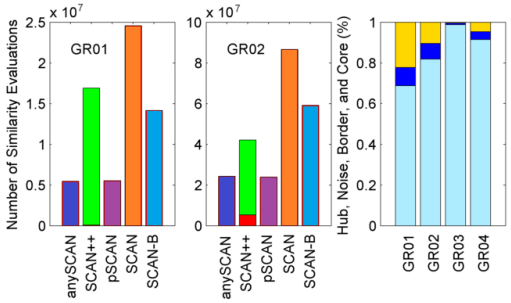


Fig. 9. (Left) Numbers of structural similarity calculations for all algorithms. For SCAN++, numbers of true similarity (bottom) and similarity sharing (top) evaluations are plotted. (Right) Numbers of hub and outlier (light blue), border (dark blue), and core (yellow) vertices (from bottom to the top) for some datasets

5 Experiments

All experiments are conducted on a Linux workstation with two 3.1 GHz Intel Xeon CPUs with 64 GB local RAM each using g++ 4.8.3 (-O3 flag) and OpenMP[2] 3.1. We use 6 large datasets (see Table 1) acquired from the Stanford Network Analysis Project (SNAP)[3] [21], The UF Sparse Matrix Collection[4], and the Laboratory of Web Algorithmics[5] [22].

5.1 Anytime SCAN

We compare anySCAN with the original algorithm SCAN and its fastest variants pSCAN [5] and SCAN++ [4]. Since these state-of-the-art techniques are originally designed to work with unweighted graphs, we extend them to work with weighted ones as described in Section 2. We also study SCAN-B, our extension of SCAN using the optimization techniques described in Section 3.4. For evaluating the anytime property of anySCAN, we use the results of SCAN as ground truth and Normalized Mutual Information (NMI) scores [23] for assessing how close the intermediate result is compared to that of SCAN. NMI is defined as the geometric mean of shared information between the clustering result C and the ground truth T and their conditional entropy. Its score is in $[0, 1]$ where 1 means both results are identical. Unless otherwise stated, we use default parameters $\mu = 5$, $\epsilon = 0.5$, and $\alpha = \beta = 8192$. Due to space limitations, we omit some graphs here. Interested readers please refer to [7] for more results on all graphs.

Anytime properties. Figure 7 shows the cumulative runtimes and NMI scores of anySCAN for GR01 to GR03 (see Table 1) measured at different iterations of Steps 1 to 3 of anySCAN. As we can see, the clustering qualities of anySCAN improve over time and converge toward the results of SCAN at the end (indicating by $NMI \approx 1.0$). The longer it is run, the better the NMI scores it obtains, i.e., its results are more similar to those of SCAN. Additionally, since anySCAN is an anytime algorithm, it has the benefit that it can be stopped at arbitrary time points for approximating results as well as saving computation cost. For example, one can stop anySCAN with good NMI scores ≈ 0.5 after 24.08, 13.94, and 8.91 seconds for GR01 to GR03 and acquire acceleration factors of up to 14.55 times compared to SCAN. This property makes anySCAN an interactive algorithm that fits well in systems with limited time constraints and that require fast response times.

Overall performance. In Figure 8, we further compare the final cumulative runtimes of anySCAN and others w.r.t. parameters μ ($\epsilon = 0.5$) and ϵ ($\mu = 5$) for GR01 to GR03 (GR04 to GR05 are omitted due to space constraints). pSCAN is slightly faster than anySCAN on GR02 ($\bar{d} = 14.2$) and GR05 ($\bar{d} = 15.8$), and is slightly slower than anySCAN on GR01 ($\bar{d} = 127.0$), GR03 ($\bar{d} = 18.7$), and GR04 ($\bar{d} = 38.1$). SCAN++ does not work well when ϵ and μ are small due to its two-hop-away-node (DTAR) expansion scheme. The bigger the number and the neighborhood sizes of core vertices, the more structural similarity evaluations it must perform, thus making SCAN++ slower than SCAN in some cases due to its additional overhead of calculating and maintaining the DTARs. Similar results are also observed in [5]. Interestingly, SCAN-B works quite well despite its simplicity. For sparse graphs, e.g. GR02 and GR03, and high values of ϵ, e.g. $\epsilon = 0.8$, it is sometimes slightly faster than pSCAN and anySCAN. The reason is that most structure similarity calculations are skipped due to the filtering property described in Lemma 6, especially when ϵ is very high. For GR06 with 1.3 billion edges ($\mu = 5, \epsilon = 0.5$), pSCAN and anySCAN need 2806.49 and 2770.94 seconds, respectively, while SCAN cannot finish after 12 hours.

Similarity evaluations. Figure 9 shows the number of structural similarity evaluations for all algorithms and thus further clarifies the acquired results above. For all datasets,

2. http://www.openmp.org/
3. https://snap.stanford.edu/
4. http://www.cise.ufl.edu/research/sparse/matrices/
5. http://law.di.unimi.it/datasets.php

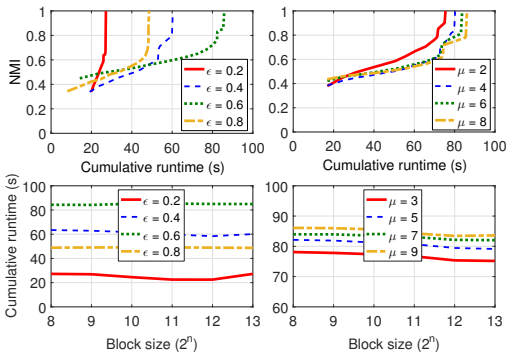


Fig. 10. The effect of parameters μ and ϵ (left) and block sizes $\alpha = \beta$ (right) for GR01

pSCAN and anySCAN use almost the same number of similarity calculations, which is much smaller than those of other methods. The number of similarity sharing calculations of SCAN++ is clearly correlated with the numbers of core vertices. The higher the number of core vertices, the more similarity sharing SCAN++ uses, meaning that the similarity evaluation time will be reduced. However, it also means that SCAN++ incurs more overhead for expanding its DTAR clusters. Sometimes, this overhead surpasses the similarity sharing benefit, thus making SCAN++ slower than SCAN-B (even though it uses fewer calculations), e.g. on GR02.

Parameter analysis. The effect of parameter ϵ on anySCAN is shown in Figure 10 (top). Due to its summarization scheme, too small ϵ, e.g. 0.2, means many super-nodes are created earlier, thus leading to better approximate results earlier. Too high ϵ, e.g. 0.8, creates many noise vertices in the beginning, thus making NMI higher (than medium values of ϵ) since they could be regarded as members of a special cluster. In contrast to ϵ, the effect of μ is quite straightforward: lower values of μ mean better approximate results since there are more core vertices to be discovered at each iteration of anySCAN. This means super nodes are merged earlier and thus makes anySCAN reach the final result of SCAN faster.

The effect of block size parameters α and β is also clear in Figure 10 (bottom). Too small values make anySCAN slower due to its anytime overhead at each iteration. When we increase the block size, there are more super-nodes. Their overlap helps to reduce the runtime by connecting more super-nodes earlier, thus reducing the number of similarity evaluations at Step 2 and Step 3 of anySCAN. For example, with $\mu = 4$, anySCAN decreases from 82.1 to 80.6 and 79.1 seconds when $\alpha = \beta$ increase from 256 to 2048 and 8192, respectively. However, when $\alpha = \beta$ are too large, redundant similarity calculations may appear during Step 1. Thus, the runtime of anySCAN may slightly increase. For example, with $\epsilon = 0.2$, anySCAN requires 27.2, 22.5 seconds, and 28.1 seconds when $\alpha = \beta = 256, 2048$, and 8192, respectively. The changes, however, are very small. This means that the performance of anySCAN is stable w.r.t. the block sizes.

Performance on synthetic graphs. Table 2 summarizes some synthetic graphs created by LFR bench mark graphs [24]. We set the number of vertices to 1,000,000 and vary the number of edges in terms of average vertex degrees and average cluster coefficients. The maximum degree is set to 100.

Figure 11 shows the performance of pSCAN and anySCAN on these synthetic graphs. When the number of edges (indicated by the average vertex degree) increases, the runtimes of both algorithm increase since more structural similarity needs to be computed. However, anySCAN tends to perform better than pSCAN on denser graphs. When the average cluster coefficient increases from 0.2 to 0.6 the runtimes of both methods decreases. Again, anySCAN tends to perform better than pSCAN on datasets with higher average cluster coefficients. This can be explained by the way anySCAN performs clustering. The denser the graphs and the better separated cluster structures, the more vertices will be put in each super-node, thus reducing the efforts for connecting them together. This leads to an improvement of the overall performance.

Id	Vertices	Edges	$\bar{d}$	c
LFR01	1,000,000	22,283,773	44.567	0.4017
LFR02	1,000,000	25,064,820	50.129	0.4007
LFR03	1,000,000	27,599,929	55.199	0.4022
LFR04	1,000,000	29,937,286	59.874	0.4011
LFR05	1,000,000	32,527,885	65.055	0.4004
LFR11	1,000,000	25,064,820	50.129	0.2012
LFR12	1,000,000	25,064,820	50.129	0.3029
LFR13	1,000,000	25,064,820	50.129	0.4168
LFR14	1,000,000	25,064,820	50.129	0.5012
LFR15	1,000,000	25,064,820	50.129	0.6003

TABLE 2
SYNTHETIC GRAPH DATASETS ($\bar{d}$ IS AVERAGED VERTEX DEGREE AND c IS AVERAGED CLUSTER COEFFICIENT)

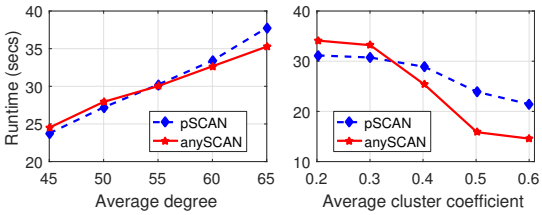


Fig. 11. Performance on synthetic graphs

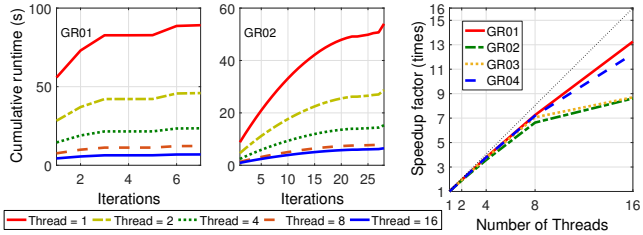


Fig. 12. Cumulative runtimes of anySCAN after each iteration of its *anytime* scheme for different numbers of threads (left) and the final runtime scalability (right)

5.2 Multicore Anytime SCAN

Anytime properties. Since anySCAN is an anytime and parallel algorithm at the same time, we study how it scales with the number of threads. As shown in Figure 12 (left), anySCAN scales very well for each iteration of its anytime scheme, using $\alpha = \beta = 32768$ as a default parameter in this section. For GR01, the speedup factors at all examined time points are very high (around 13.25 for 16 threads). More interestingly, for most datasets, the scalability of anySCAN slightly declines at each iteration. For GR02 and 16 threads as an example, at the first iteration, it achieves 9.52 times speedup factor. However, at the end, the speed up factor

Fig. 13. Speed up factors of anySCAN and an ideal algorithm w.r.t. different numbers of threads

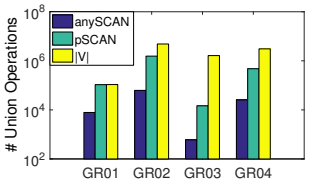


Fig. 14. Number of performed Union operations for GR01 to GR04

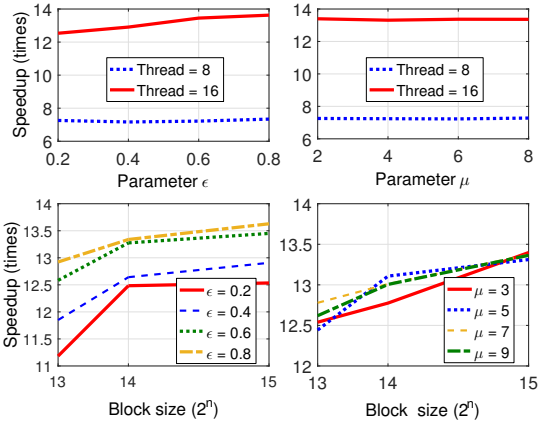


Fig. 15. The effect of parameters μ, ϵ, and block sizes on the scalability of anySCAN for the dataset GR01

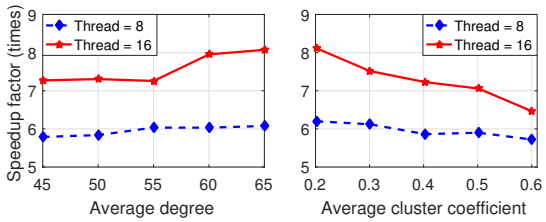


Fig. 16. Scalability on synthetic graphs

reduces to 8.61. Thus, the earlier a user stops the algorithm, the higher the speed up factor she enjoys.

Figure 12 (right) shows the final speedup factors of anySCAN w.r.t. different numbers of threads for GR01 to GR04 (GR05 and GR06 are omitted for clarity). anySCAN scales very well with the numbers of threads, e.g., almost linearly for GR01, GR04, and GR05. For GR01, the speedup factors over single thread are 1.93, 3.78, 7.24, and 13.25 for 2, 4, 8, and 16 threads, respectively. Using 16 threads, the speed up factors are 12.25, 11.41 and 9.35 for GR04, GR05 and GR06, respectively. The speed up factors on GR02 and GR03 are worse than those of the others. Beside the common NUMA effect (threads are run on two different CPUs with 64 GB local memory each), one reason clearly is the sparseness of the graph. GR02 and GR03 are much sparser than GR01, GR04 and GR05 (indicated by the averaged vertex degrees $\overline{d}$). Moreover, the degrees of vertices vary significantly on GR02 and GR03. These make the workloads of threads very unbalanced, thus reducing the scalability of anySCAN (see also Figure 13 for further analysis). In this case, increasing the block size values will help to solve the problem (see Figure 15 for the parameter analysis). Sorting vertices and processing ones with higher degrees first might also balance the workloads better.

Performance comparison. Since anySCAN is the first parallel version of SCAN on multicore processors, we compare it with an ideal parallel algorithm for further assessing its performance in Figure 13. The ideal algorithm only calculates the structural similarities (without optimizations) of all edges of G which is the most expensive part of SCAN and ignores the label propagation process among vertices to build clusters. Obviously, it does not require synchronizations among threads and thus has an ideal scalability w.r.t. the number of threads. Its performance is calculated by the speedup factor between the single thread and multiple-thread usages. Note that pSCAN [5] and SCAN++ [4] are highly sequential and are non-trivial problems for parallelizing efficiently. As we can see, anySCAN acquires very close performance to the ideal algorithm for most datasets, especially GR02. For GR03, the performance difference between anySCAN and the ideal method is larger than for the other data sets for 16 threads. Here, optimization techniques for speeding up the structural similarity calculation cause the problem. Most calculations are filtered out earlier following Lemma 6. Even though this makes the algorithm much more efficient, it lowers the overall workload for threads at each iteration of anySCAN, thus making it more sensitive to the load balancing problem, as well as reducing the ratio of sequential and parallel parts of anySCAN and thus the scalability following Amdahl's law for scalability.

AnySCAN needs to perform the Union operations inside critical areas for merging super-nodes inside Step 2 and 3. Thus, the numbers of Union operations strongly affect its scalability and are shown in Figure 14. Since pSCAN uses the Disjoint Set data structure like anySCAN, we include it here for a comparison even though it is not a parallel algorithm. As we see, pSCAN uses much fewer Union operations than the numbers of vertices $|V|$ of G. And, anySCAN uses even fewer operations (up to 25 times and 2725 times compared to pSCAN and $|V|$, respectively). Moreover, most of them (7685/7844, 31440/62351, 268/599, and 19969/25426 operations for GR01 to GR04, respectively) are executed sequentially in Step 1 of anySCAN, leaving only few to be executed in Step 2 and 3 inside critical sections. And fewer operations means better scalability of anySCAN as shown in Figure 12.

Parameter analysis. Processing times of different core vertices vary significantly depending on their neighborhood sizes, and are obviously more expensive than those of noise ones. Balancing the workloads for threads is therefore harder if there are more core vertices. Moreover, the number of merged super-nodes is higher. Thus, increasing μ and ϵ will lead to greater speed up factors as shown in Figure 15 (top), since the number of core vertices is reduced. On the other hand, increasing the block size will provide more work for threads at each iteration, therefore increasing the workload balance and thus increasing the scalability of anySCAN as shown in Figure 15 (bottom).

Performance on synthetic graphs. Generally, when the

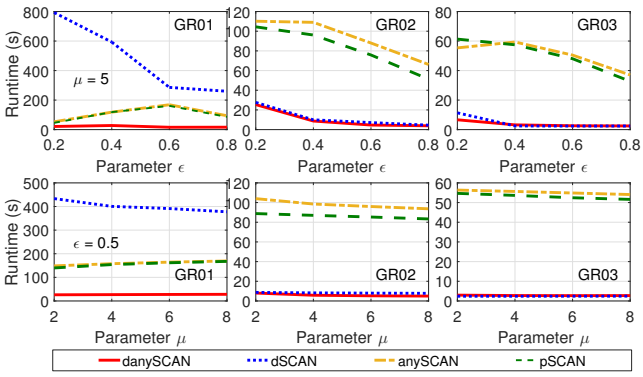


Fig. 17. Performance of danySCAN and others for GR01 to GR03 with different parameters $\epsilon(\mu = 5)$ and $\mu(\epsilon = 0.5)$

average degree grows, the overall scalability of anySCAN improves since the number of structural similarity calculations and the time for evaluating them both grow as well. On the other hand, when the average cluster coefficient is high, the overlap among the neighborhoods of vertices is also large, thus leading to more conflicts during Step 2 and 3 of anySCAN. This reduces the overall scalability. Figure 16 shows the scalability of anySCAN using multiple threads when varying the average degrees and cluster coefficients. Though there are some small fluctuations, the above trends are generally observed in most cases.

5.3 Dynamic Anytime SCAN

Since danySCAN is the first approach for weighted dynamic structural graph clustering, we compare it with dSCAN [8], an extension of pSCAN for dynamic graphs, also in its non-weighted mode with two operations including insertion and deletion. Moreover, since it processes updates in bulks, we compare it with pSCAN [5] and anySCAN [7] for fully reclustering the whole graphs. Unless otherwise stated, we use default parameters $\mu = 5$, $\epsilon = 0.5$, and $\alpha = \beta = 8192$.

Performance comparison. Figure 17 shows the performance comparisons among danySCAN and competitors for GR01 to GR03 for different parameters $\epsilon(\mu = 5)$ and $\mu(\epsilon = 0.5)$ and 4000 updates (2000 delete and 2000 insert operations). Clearly, danySCAN is much more efficient (up to 35.2 times faster) than pSCAN and anySCAN since it only updates the cluster structures rather than starting from scratch. Compared to dSCAN, danySCAN runs much faster than dSCAN on GR01 and GR02 and is comparable to dSCAN on GR03. The reason is that dSCAN reclusters with each update, thus incurring costs for repeatedly traversing the graph, while danySCAN only needs to update clusters one time using its bulk processing scheme. Moreover, the denser the graphs, the better the performance of danySCAN compared to dSCAN due to the higher graph traversal costs. For example, dSCAN runs much slower (24.8 times) than danySCAN on GR01 ($\bar{d} = 127$) and is slightly slower (1.5 times) than danySCAN on GR02 ($\bar{d} = 14.2$).

Figure 18 shows the cumulative runtimes of different methods when the number of updates changes from 2000 to 10000. While the runtimes of pSCAN and anySCAN remains stable, those of danySCAN and dSCAN increase. However, the performance changes for dSCAN are much larger than for danySCAN. Again, this is because the bulk update scheme of danySCAN incurs less overhead than the batch update scheme of dSCAN.

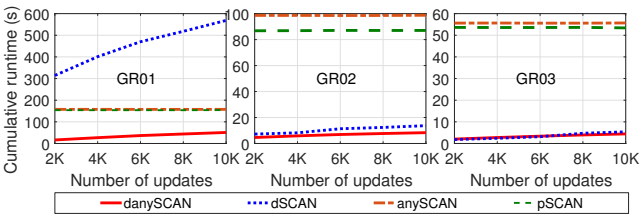


Fig. 18. Performance of danySCAN and others w.r.t. the number of updates for different datasets

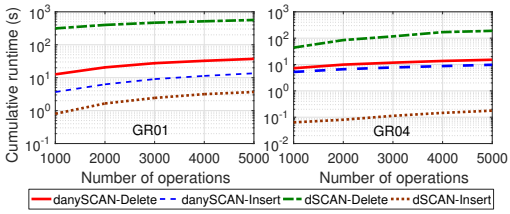


Fig. 19. Performance for different update operations

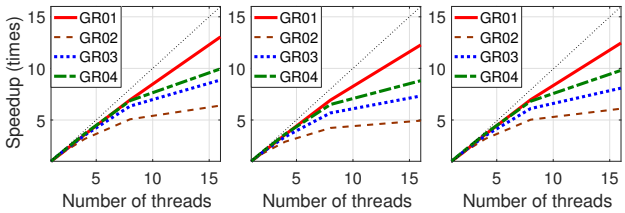


Fig. 20. Scalability of danySCAN w.r.t. the numbers of threads for different update operations including weight change (left), insertion (middle) and deletion (right), respectively

Parameter analysis. As seen from Figure 17, when ϵ increases, the runtime of danySCAN decreases. The main reason is that with lower values of ϵ, we have more core vertices. Thus, when there is a change in the graph, the number of *affected clusters* is higher. Thus, both dSCAN and danySCAN have to traverse a larger part of the graph for rebuilding clusters. Consequently, the overhead is higher, leading to performance degradation. Similarly, with lower values of μ, we have more core vertices, leading to worse performance than with higher values of μ.

Performance for different update operations. Figure 19 shows the cumulative runtimes of danySCAN and dSCAN for 1000 to 5000 insertion and deletion operations for GR01 and GR04. For both methods, deletion is more expensive than insertion. This is due to the fact that when an edge (u, v) is deleted, it more likely will lead to a split of clusters. Thus, the algorithm must traverse the graph for rebuilding these clusters. In the insertion, it usually leads to a merge of clusters which is much cheaper due to the merging approach of both dSCAN and danySCAN. Moreover, while dSCAN processes the deletion cases much slower than danySCAN, it deals with the insertion ones very efficiently. For GR01 and 5000 insertion operations, dSCAN needs 3.67 seconds while danySCAN requires 13.58 seconds. However, with 5000 deletion operations, dSCAN consumes 564.85 seconds compared to 37.30 seconds of danySCAN. danySCAN is designed to cope with mixed operations in its bulk update scheme and not optimized for each specific operation like dSCAN. However, since the deletion time is much larger than the insertion time, danySCAN still outperforms its

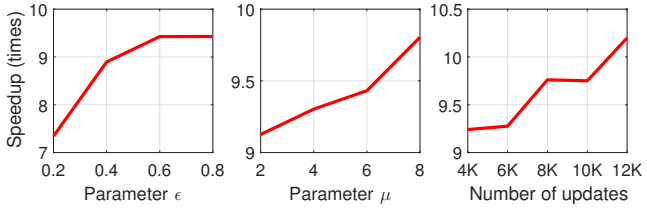


Fig. 21. Effect of parameters μ, ϵ and the number of weight changes on the scalability of danySCAN on 16 threads for GR04 with 8000 updates

competitor if the number of insertions is not much larger than the number of deletions as demonstrated in Figure 17. For both types of operations, when the number of updates is larger, the better the performance of danySCAN is compared to that of dSCAN as observed from Figure 18 above. The weight change case of danySCAN has the performance between the insertion and the deletion cases since it naturally consists of these two.

5.4 Dynamic Anytime SCAN on Multicore CPUs

Scalability. Figure 20 shows the scalability of danySCAN w.r.t. the number of threads for different datasets and 10,000 different update operations using default parameters $\mu = 5, \epsilon = 0.5$ and $\alpha = \beta = 32768$. Overall, danySCAN scales very well with the number of threads. For example, using 16 threads, the speedup factor on GR01 is up to 13 times. Among different datasets, it has the worst performance on GR02. The reason has been examined in Section 5.2. GR02 is the sparsest graph with significantly varied vertex degrees, thus making workload balancing harder, which is reflected in the relatively weaker performance of danySCAN. The denser the graph, the better the scalability of danySCAN due to the larger overall workload for threads.

Weight change and deletion operations show better performance than insertion. The reason is simply that the more edges are deleted, the more clusters we need to rebuild and the heavier the update tasks which leads to better scalability.

Parameter analysis. As seen in Figure 21, smaller values of ϵ and μ will increase the number of core vertices, thus leading to workload imbalance and more super-node merges. This typically decreases the scalability as discussed in Section 5.2. When the number of updates is large, more vertices are affected in the bulk scheme. In turn, this increases the overall workload of threads. And consequently, it raises the scalability. The blocksize has similar effect on danySCAN compared to anySCAN like μ and ϵ (thus omitted here). The same results are observed for the other update cases.

6 Related Work and Discussion

Graph clustering techniques. Due to the ubiquitousness of graph like structures such as social networks, graph clustering techniques are becoming more and more important. There are graph clustering models such as modularity-based methods [1], graph partitioning [2], and structural graph clustering methods, which are our main focus here.

Structural graph clustering. Structural graph clustering, in particular the density-based approach of SCAN [3], is an active research topic with many extensions. For example, SCOT [25], HintClus [25], and gSkeletonClu [20] aim to solve the parameter setting problem of SCAN. DHSCAN [26] and AHSCAN [27] are divisive and agglomerative hierarchical algorithms, respectively, using the structural similarity notion of SCAN. In this work, we focus on techniques that speed up the algorithm SCAN [3].

LinkScan* [28] improves the efficiency of SCAN using an edge sampling technique for reducing the number of similarity evaluations. However, it only approximates the result of SCAN unlike pSCAN [5] and SCAN++ [4].

pSCAN [5] is a recent state-of-the-art technique. Instead of calculating the full neighborhood of a vertice p, it only checks if p is a core and then tries to connect p to other core vertices from other clusters. This scheme significantly reduces structural evaluation and makes pSCAN one of the fastest variants of SCAN. The final cumulative runtimes of anySCAN are almost similar to those of pSCAN. However, anySCAN has the power of approximation and exact techniques at the same time in its anytime scheme.

SCAN++ [4] is the closest related work to anySCAN. It builds a set of $pivots$ by performing neighborhood calculations for a vertex p, called a pivot, and expanding pivots for all nodes that are two-hop-away from p in the same way as SCAN until it is converged. Then, it tries to connect pivots by examining and pruning $bridge$ vertices that connect them. In this way, the number of similarity calculations is reduced. The goals of these steps bear some similarity with Step 1 and 2 of anySCAN, though anySCAN has a completely different angle. First, anySCAN randomly draws vertices for summarization and only keeps core vertices as *super-nodes* for further processing, thus limiting redundant similarity calculations since the number of super-nodes in anySCAN is much smaller than the number of pivots in SCAN++. Second, it connects super-nodes in a different manner as SCAN++ by examining two different kinds of connections ($strong$ and $weak$ ones) separately as well as processing only core vertices as super-nodes. Last, noise vertices are examined in the post processing step separately from the whole algorithm. Thus, it has better pruning power than SCAN++. And thus, it is more efficient.

Parallelizing SCAN. There exist some efforts for parallelizing SCAN. PSCAN [6] is a parallel version of SCAN using MapReduce for distributed computing. In [4], the authors also briefly introduce a MapReduce framework for SCAN++. The distributed model of MapReduce differs significantly from parallel computing in shared memory ones where memory is a contested resource, and latencies are small [29], [30]. Thus naively tranforming distributed algorithms to shared memory architectures will obviously be very inefficient as pointed out by many previous studies, e.g., [31]. To the best of our knowledge, anySCAN is the first parallel algorithm for SCAN on shared memory architectures such as multicore CPUs. Moreover, existing techniques such as pSCAN [5] and SCAN++ [4] incur many synchronizations that leave threads idle in a naive parallelization, which significantly reduces the scalability w.r.t. the number of threads. Recently, Takahashi et al. [9] introduce SCAN-XP, a parallel version of SCAN specifically designed for Intel Xeon Phi Coprocessors. This method, however, requires all structural similarities to be calculated. Thus, it is not a work-efficient parallel technique like anySCAN. It also is not designed to work with weighted graphs like anySCAN.

Dynamic structural graph clustering. There exist some dynamic structural algorithms. DENGRAPH [19] is an incremental clustering algorithm designed to detect communities in large and dynamic social networks. Incorder [32] is an incremental version of gSkeletonClu [20]. Chang et al. [8] also introduces an incremental version of pSCAN, called dSCAN, which is closest to danySCAN. All these techniques update clusters in a batch mode with each insertion or deletion. For large number of updates, this scheme incurs many redundant calculations, thus decreasing their performance. On the other hand, danySCAN processes updates in a bulk mode, which is more efficient. Moreover, updates can be done in an anytime parallel way. In addition, danySCAN can handle dynamic weighted graphs.

7 Conclusion

In this paper, we propose an approach for accelerating the structural graph clustering algorithm SCAN. Our anytime algorithm, called anySCAN, quickly produces an approximate result in the beginning and continuously refines it for acquiring better results within arbitrary time constraints. This anytime scheme provides an efficient way for coping with large graphs. More interestingly, anySCAN is, at the same time, a parallel algorithm. Each iteration of its anytime scheme can be performed in parallel, thus further accelerating performance. To the best of our knowledge, anySCAN is the first anytime and parallel structural graph clustering algorithm. We additionally introduce an extension of anySCAN, called danySCAN, to parallelize processing of dynamic graphs. Experiments show that anySCAN has very good performance on large graph datasets in its anytime scheme. It also scales very well with the number of threads under shared memory architectures such as multicore CPUs.

In future work, we aim for a distribute extension of anySCAN to handle massive graph datasets on different machines, and applying our techniques to the study of patient data with sleep disorder symptoms.

Acknowledgments

Special thank to Prof. Lijun Chang for providing us with the source code of pSCAN, Anh Le, Kenneth S. Bøgh, Panagiotis Bouros, and Prof. Hiroaki Shiokawa for their help and useful discussions. We also thank our anonymous reviewers for their fruitful comments. Part of this research was funded by a Villum postdoc fellowship and the CDP Life Project.

References

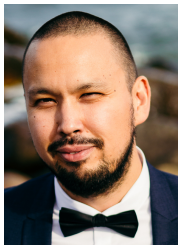
[1] R. Guimer and L. A. N. Amaral, "Functional cartography of complex metabolic networks," *Nature*, 2005.
[2] J. Shi and J. Malik, "Normalized Cuts and Image Segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, 2000.
[3] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger, "SCAN: a structural clustering algorithm for networks," in *KDD*, 2007, pp. 824–833.
[4] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-scale Graphs," *PVLDB*, vol. 8, no. 11, pp. 1178–1189, 2015.
[5] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang, "pSCAN: Fast and Exact Structural Graph Clustering," in *ICDE*, 2016, pp. 481–490.
[6] W. Zhao, V. S. Martha, and X. Xu, "PSCAN: A Parallel Structural Clustering Algorithm for Big Networks in MapReduce," in *AINA*, 2013, pp. 862–869.
[7] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. Birk, "Scalable and Interactive Graph Clustering Algorithm on Multicore CPUs," in *ICDE*, 2017, pp. 349–360.
[8] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang, "pSCAN: Fast and Exact Structural Graph Clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 2, pp. 387–401, 2017.
[9] T. Takahashi, H. Shiokawa, and H. Kitagawa, "SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors," in *NDA@SIGMOD*, 2017, pp. 6:1–6:7.
[10] S. Zilberstein, "Using Anytime Algorithms in Intelligent Systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.
[11] T. Kobayashi, M. Iwamura, T. Matsuda, and K. Kise, "An Anytime Algorithm for Camera-Based Character Recognition," in *ICDAR*, 2013, pp. 1140–1144.
[12] N. Begum, L. Ulanova, J. Wang, and E. J. Keogh, "Accelerating Dynamic Time Warping Clustering with a Novel Admissible Pruning Strategy," in *KDD*, 2015, pp. 49–58.
[13] S. T. Mai, I. Assent, and A. Le, "Anytime OPTICS: An Efficient Approach for Hierarchical Density-Based Clustering," in *DASFAA*, 2016, pp. 164–179.
[14] S. T. Mai, I. Assent, and M. Storgaard, "AnyDBC: An Efficient Anytime Density-based Clustering Algorithm for Very Large Complex Datasets," in *KDD*, 2016, pp. 1025–1034.
[15] S. T. Mai, X. He, J. Feng, and C. Böhm, "Efficient Anytime Density-based Clustering," in *SDM*, 2013, pp. 112–120.
[16] S. T. Mai, X. He, J. Feng, C. Plant, and C. Böhm, "Anytime density-based clustering of complex data," *Knowl. Inf. Syst.*, vol. 45, no. 2, pp. 319–355, 2015.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
[18] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
[19] T. Falkowski, A. Barth, and M. Spiliopoulou, "DENGRAPH: A Density-based Community Detection Algorithm," in *Web Intelligence*, 2007, pp. 112–115.
[20] H. Sun, J. Huang, J. Han, H. Deng, P. Zhao, and B. Feng, "gSkeletonClu: Density-Based Network Clustering via Structure-Connected Tree Division or Agglomeration," in *ICDM*, 2010, pp. 481–490.
[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.
[22] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW*. Manhattan, USA: ACM Press, 2004, pp. 595–601.
[23] M. J. Zaki and W. M. Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. New York, NY, USA: Cambridge University Press, 2014.
[24] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, p. 046110, Oct 2008.
[25] D. Bortner and J. Han, "Progressive clustering of networks using Structure-Connected Order of Traversal," in *ICDE*, 2010, pp. 653–656.
[26] N. Yuruk, M. Mete, X. Xu, and T. A. J. Schweiger, "A Divisive Hierarchical Structural Clustering Algorithm for Networks," in *ICDM*, 2007, pp. 441–448.
[27] ——, "AHSCAN: Agglomerative Hierarchical Structural Clustering Algorithm for Networks," in *ASONAM*, 2009, pp. 72–77.
[28] S. Lim, S. Ryu, S. Kwon, K. Jung, and J. Lee, "LinkSCAN*: Overlapping community detection using the link-space transformation," in *ICDE*, 2014, pp. 292–303.
[29] S. Tatikonda and S. Parthasarathy, "Mining Tree-Structured Data on Multicore Systems," *PVLDB*, vol. 2, no. 1, pp. 694–705, 2009.
[30] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. D. Nguyen, Y. Chen, and P. Dubey, "A Characterization of Data Mining Workloads on a Modern Processor," in *DaMoN*, 2005.
[31] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. keng Liao, F. Manne, and A. N. Choudhary, "A new scalable parallel DBSCAN algorithm using the disjoint-set data structure," in *SC*, 2012, p. 62.
[32] H. Sun, J. Huang, X. Zhang, J. Liu, D. Wang, H. Liu, J. Zou, and Q. Song, "IncOrder: Incremental density-based community detection in dynamic networks," *Knowl.-Based Syst.*, vol. 72, pp. 1–12, 2014.

Son T. Mai is working at University of Grenoble Alpes, Grenoble, France. Before, he was at Aarhus University, Aarhus, Denmark and Ludwig-Maximilians-University of Munich (LMU), Munich, Germany.

Sihem Amer-Yahia is a CNRS Research Director. Her interests are at the intersection of large-scale data management and social data exploration. Sihem held positions at QCRI, Yahoo! Research and AT&T Labs. She served on the SIGMOD Executive Board, the VLDB Endowment, and the EDBT Board. She is Editor-in-Chief of the VLDB Journal. Sihem serves as co-chair of PVLDB 2018, WWW 2018 Tutorials and ICDE 2019 Tutorials.

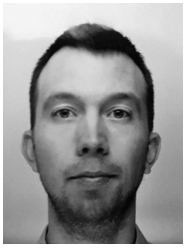
Ira Assent is an associate professor and head of the Data-Intensive Systems research group at the Department of Computer Science at Aarhus University, Denmark. Prior to joining Aarhus University in 2010, she was an assistant professor at Aalborg University, Denmark. Ira received her Ph.D. in Computer Science with distinction from RWTH Aachen University, Germany in 2008.

Mathias Skovgaard Birk received the MSc degree in computer science in 2017 from Aarhus University, Denmark, after finishing his thesis on incremental structural clustering of dynamic networks. His research interests include clustering, machine learning and design and development of distributed systems. He does not currently work in academia, but holds a position as software developer in a fintech company in Aarhus.



Martin Storgaard Dieu received the BSc degree in computer science at Southern University of Denmark in 2013 and his MSc degree in computer science at Aarhus University, Denmark in 2016. His main research interests are applied machine learning including graph clustering and anytime algorithms.

Jon Jacobsen received the BSc and MSc degree in computer science in 2014 and 2017 respectively from Aarhus University, Denmark. He studied as an exchange student at Nanyang Technological University's School of Computer Science and Engineering from July 2015 to December 2015. He is currently working as a software developer in Getinge Cetrea's R&D department.



Jesper M. Kristensen received the MSc degree in computer science in 2016 from Aarhus University, Denmark. He is currently an IT consultant at Netcompany – IT and Business Consulting.