2017 IEEE 33rd International Conference on Data Engineering

Scalable and Interactive Graph Clustering Algorithm on Multicore CPUs

Son T. Mai, Martin Storgaard Dieu, Ira Assent, Jon Jacobsen, Jesper Kristensen, and Mathias Birk
Department of Computer Science, Aarhus University, Aarhus, Denmark
Email: {mtson, martin, ira, jon, jesper, mathias}@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 structures consisting of the so-called *super-nodes* for building clusters. During its runtime, anySCAN can be suppressed for examining intermediate results and resumed for finding better result at arbitrary time points, making it an *anytime* algorithm which is capable to deal with 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 further speeding up the algorithm at each iteration. Consequently, anySCAN *uniquely* is an interactive and parallel algorithm at the same time. Experiments are conducted on very large real graph datasets for demonstrating the performance of anySCAN. It acquires very good approximate results early, leading to orders of magnitude speedup factor compared to SCAN and its variants. Using 16 threads, the acquired speed up factors are up to $\approx$ 13.5 times over its sequential version.**

Keywords*—Structural graph clustering, SCAN, anytime clustering, parallel algorithm, multicore CPUs

I. Introduction

Given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, graph clustering algorithms group vertices in V so that those in the same groups are highly connected and there are few connections among different groups. It has many applications, e.g., finding communities of people in social networks or detecting hidden structures in graphs. During the last decades, many graph clustering techniques have been introduced such as modularity-based methods [1], graph partitioning [2], and structural graph clustering methods [3]. 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 as well as examining all edges for labeling vertices. For very large graphs, these overheads obviously are a computational bottleneck that limits its applicability. Enhancing the performance of SCAN is thus an important task and is currently attracting considerable research efforts, e.g., [4], [5], [6]. 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? Or, 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 a single thread usage?

Contributions. In this paper, we focus on the problem of speeding up SCAN for very large graph datasets. Our algorithm, called *anytime SCAN* (anySCAN), has some *unique* properties as described below.

First, our algorithm, as an *anytime* technique, quickly produces an approximate result in the beginning and then iteratively refines it during its execution. Thus while it is running, users can suspend it for examining intermediate results and resume it for finding better ones 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 solving time consuming problems in many different fields, e.g., robotics [7], pattern recognitions [8], and data mining [9], none of existing variants of SCAN is specifically designed for this interesting and useful property.

Second, anySCAN is the first parallel extension of SCAN specifically designed for shared memory architectures such as multicore processors which are becoming ubiquitous nowadays. By maintaining an underlying cluster structure consisting of the so-called *super-nodes* and processing vertices in blocks for connecting these super-nodes to form clusters, anySCAN significantly reduce synchronizations 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 producing the exact results of SCAN faster.

Thirds, anySCAN is a *work-efficient* anytime and parallel method. Concretely, an anytime version of an algorithm usually ends up being slower than itself at the end due to additional costs for maintaining the anytime properties. Similarly, a parallel algorithm tries to ensure high throughputs for better utilizing all its threads. This often comes with a cost of increasing the overall workload, making it much slower than state-of-the-art sequential techniques on a single thread (or even multiple threads) usage. 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 thus is a *work-efficient* method, i.e., its final cumulative runtimes are much faster than SCAN and its variants even on a single thread.

Outline. The rest of the paper is organized as follows. In Section II, we review some basic notions of SCAN and extend them to cope with more general weighted graphs. In Section III, we describe our algorithm anySCAN. Section IV evaluates the performance of our algorithm. Section V reviews related works. And, Section VI concludes the paper and highlights future directions.

II. Some Basic Notions

A. The algorithm SCAN

SCAN [3] is originally designed for clustering undirected and unweighted graphs. However, in this work, we are more interested in weighted graphs, which are more general and therefore have wider applicability. Thus, we first extend the notion of SCAN to work with weighted graph in this Section.

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

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 replacing its structure similarity with the weighted one in Definition 1.

Definition 2: The structural neighborhood of a vertex p, denoted as 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 builts 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. Due to space limitation, interested reader please refer to [3] for details. The time complexity of SCAN is $O(|E|)$ (or $O(min(|N_p|, |N_q|) \cdot |E|)$) and is *worst-case optimal* as proven in [5].

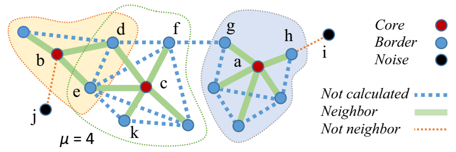

Fig. 1. Basic ideas of anySCAN

B. Anytime algorithms

Anytime algorithms are widely-used to cope with time consuming problems in many fields, e.g., [9], [10], [11], [12], [13], [7]. Their general idea is quickly producing an approximate result in the beginning, and then iteratively improving it further. 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 end-users during its execution.

In [7], the authors describe some important characteristics of anytime algorithms such as (1) the final results of anytime algorithms should be similar to those of batch ones and (2) the final cumulative runtimes of anytime algorithms should not be much larger than those of batch ones. Due to space limitation, interested readers please refer to [7] for more details.

III. The Algorithm Anytime SCAN

Figure 1 illustrates the basic ideas of anySCAN to process vertices such that the cluster structure quickly emerged and is refined without naively checking all vertices. Assume that with 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 ones. Obviously, all other vertices now belong to at least one cluster, e.g., g and h must be in the same cluster with a following Definition 4 and 5. We then try to find true clusters using these known information as guidelines. If we choose d for examining and discover that $core(d)$, then b, c and all their neighbors (N_b^ϵ and N_c^ϵ) must belong to the same cluster. Next, if we examine f and see that f is not a core, then N_a^ϵ and N_c^ϵ will surely belong to different clusters. Now, we can safely stop the algorithm and have the same clustering result as SCAN without further examining the rest of vertices, since the clusters will not change anymore regardless of additional neighborhood checks. This helps to save many structural similarity calculations, thereby reducing runtime. Moreover, anySCAN can be interrupted after some arbitrary neighborhood checks for producing roughly approximate results of SCAN. Besides that, if we process a *block* of vertices each time instead of a single vertex like SCAN and its variants [5], [4], [3], 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.

In Section III-A we present the four major Steps of the sequential version of anySCAN before discussing its parallelization in Section III-B.

A. Anytime SCAN

Concretely, anySCAN is built upon the concepts: summarization, selection, merging, and block processing which

```
1  Function anySCAN ($G, \mu, \varepsilon, \alpha, \beta$)
2  BeginFunction
3      /* Step 1: Summarization */
4      while there still exist untouched vertices do
5          select a set $X$ of $\alpha$ 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 $N_p^\varepsilon$ do
10                     mark the state of $q$ and increase the number of neighbors $nei(q)$ 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 $g \epsilon\ SN_q$
15             else
16                 for all vertex $q$ in $N_p^\varepsilon$ 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 unprocessed-border vertices that belong to at least two super nodes
21     sort $S$ in the descending order according to the number of super-nodes
22     while $S$ is not empty do
23         remove a set $X$ of $\beta$ 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 position $i$ and $i + 1$ of $SN_p$
31                 if Findset($sn(u)$) $\neq$ 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 unprocessed-core or processed-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 $\beta$ 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($clu(p)$) $\neq$ Findset($clu(q)$) then
46                     if $\sigma(p, q) \geq \varepsilon$ 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 (α and β is the block sizes described in Section III-B)

we describe in detail in the following. The pseudo code for anySCAN (non-parallel version) can be found 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. 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 into a noise list L for a post processing step. If $core(p)$ (Line 8-14), N_p^ϵ is summarized into 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. Due to space limitations, we only briefly describe some cases here. For example, if $|N_p| < \mu$, then we know that p is surely not a core without examining its neighbors. Thus $state(p)$ is *unprocessed-noise*. If later on, we know that p is a neighbor of a core vertex q, then p will be a border of a cluster. Thus its state will be changed to *processed-border*. If none of its neighbors is core, p is assigned *processed-noise* state. If an *unprocessed-border* object p is examined and has more than μ neighbors, it is surely a core. Thus its state changes to *processed-core*. Otherwise if it has less than μ neighbors, it is changed to *processed-border* instead of *processed-noise* since it already belongs to a cluster. If an *unprocessed-border* object p is not examined but p has more than μ neighbors then it is surely a core and is thus assigned *unprocessed-core* state. A *border* vertex will never become a *core*. Also, a *core* vertex will never be changed to a *border* or a *noise* vertex. An *processed* vertex will never be changed to *unprocessed*.

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

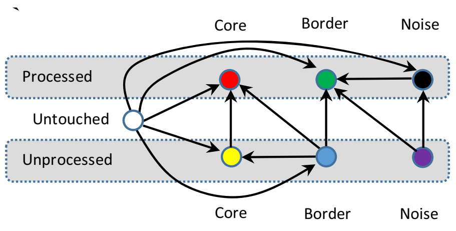

Fig. 3. The state transition schema for vertices

Theorem 1: During the execution of anySCAN, the state of each vertex changes according to the transition schema in Figure 3.

Proof: It can be verified through Definition 3 and 4 of SCAN in Section II. ■

We use the same colors in Figure 1 and 3 for denoting the states of vertices in the whole paper. As demonstrated in Figure 3, 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 1: All objects inside a super-node $sn(p)$ belong to the same cluster.

Proof: Directly inferred from Definition 4. ■

Following Lemma 1, 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, it will help to reduce the label propagation time. 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 following Lemma 2 described below). Here, a Disjoint-set data structure [14] is employed for keeping track of the labels of all super-nodes since each super-node only belongs to one cluster. It supports two important operations including (1) *Findset*: for finding which subset a particular super-node is in and (2) *Union*: for merging two subsets of super-nodes into a single subset. In the next steps, we will merge these super-nodes to produce the same final clustering result as SCAN.

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

Step 2: Merging strongly-related super-nodes. Step 2 to merges super-nodes to form cluster starting with nodes that are more likely to be in the same cluster.

Definition 6: Two super-nodes $sn(p)$ and $sn(q)$ are called *strongly-related* 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 2: 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: Let a and b be two arbitrary vertices in $sn(p)$ and $sn(q)$, respectively. We have $a \lhd p$ and $q \rhd b$ (Definition 4). Since $core(u)$ and $u \in N_p^\epsilon \cap N_q^\epsilon$, $p \lhd u$ and $u \rhd q$. Thus, a and b are density-connected according to Definition 4. ■

Lemma 2 states that if two super-nodes share a core vertex, they will be merged together. We start by collecting 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 play no role for determining the connection of two super-nodes[2]. We sort all vertices in S in descending order of the number of super-nodes they belong to (Line 21). Then, each vertex is extracted and processed 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 one (Line 26). To do so, we only need to 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 between p and its adjacent vertices. This helps to reduce the structural similarity evaluation, thereby reducing 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. Now, if p is a core, all of its super-nodes $g \in SN_p$ will belong to the same cluster due to Lemma 2. 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 two vertices 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, it is e's turn. However it can now be ignored since all of its super-nodes have the same label. Here we detect exactly two clusters of SCAN without examining all other vertices, thereby significantly reducing runtime. However, to finally conclude that the result are completely identical to SCAN, we need to check whether $sn(a)$ and $sn(c)$ should be merged. Since they share nothing, verifying their connection is much more challenging and will be performed in Step 3.

Step 3: Merging weakly-related super-nodes. Step 3 verifies additional connections that cannot be detected 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 vice versa, i.e., u and v are adjacent vertices. For example, in Figure 1, $sn(a)$ and $sn(c)$ are *weakly-related*, while $sn(a)$ and $sn(b)$ are not.

Obviously, two *weakly-related* super-nodes may be in the same cluster under certain conditions. In this step, we will deal with this kind of connections.

Lemma 3: 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 $sn(p)$ and $sn(q)$ belong to the same cluster.

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

In Figure 1, if f and g are core vertices and $\sigma(f,g) \geq \epsilon$, a and c will be density-connected, thus $sn(a)$ and $sn(c)$ will belong to the same cluster. Identifying the connection between *weakly-connected* super-nodes is much difficult than for *strongly-connected* cases. A simple approach is 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. This 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 3) (Line 35). Similar to step 2, 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 it 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 vertices. For every cluster C found in step 2, we choose a super-node $sn(u)$ as its representative. We use the term $clu(p)$ for denoting the cluster that contains vertex p.

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. Similar to the previous step, we only need to explore its adjacency vertices $q \in N_p$ until p exceeds the threshold making it core. We then stop to save structural similarity calculations (Line 41). If p is a core, we set its state to *unprocessed-core*. Otherwise, it is changed to *processed-border* (see the transition schema in 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 cluster (Line 45), we calculate the 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 $sn(v) = clu(q)$

[2]Note that all *core* vertices have been processed in Step 1 (Line 12-14) by merging their *super-nodes* together. This helps to reduce the number of synchronizations among threads for the parallel version of anySCAN in Section III-B (see Figure 4 line 41-42 and 60-61)

(Line 46-47). 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 one. Thus, f and g belong to different cluster.

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 this Step (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 noises at the end, we can exploit previous results such as core vertices to save more similarity calculations, thus enhancing performance.

B. Parallel Anytime SCAN

anySCAN can be efficiently parallelized in shared memory architectures. We outline the algorithmic principles below.

Block processing. Instead of processing vertices one-by-one as described in Section III-A for simplicity, another 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 helps to reduce the overhead of the anytime scheme of anySCAN by reducing the number of iterations, thereby increasing performance.

Block processing also allows multiple threads to perform core checks, which is the most expensive part of anySCAN, concurrently. Thus, it allows an efficient parallel process which uses much fewer structural similarity calculations than SCAN and thus is as fast as other extensions of SCAN using a single thread, while it scales well for multiple threads. Specifically, it maintains the anytime property even in parallel mode. In Section IV we will study the effect of the block sizes α and β on the performance of anySCAN.

Parallelizing. Figure 4 shows the pseudo code for the parallel version of anySCAN using OpenMP [15]. Since all vertices have different neighborhood sizes, we use dynamic scheduling for load balancing, e.g., Line 6, 10, 30, and 34. The Union operation is not thread-safe. Thus, it must be locked inside critical sections (Line 41 and 60). However since the number of super-nodes is much smaller than the number of vertices, the number of Union operations is therefore small and does not affect the scalability of anySCAN much (see Figure 12

```
1  Function anySCAN ($G, \mu, \varepsilon, \alpha, \beta$)
2  BeginFunction
3      /* Step 1: Summarization */
4      while there still exist untouched vertices do
5          select a set $X$ of $\alpha$ untouched vertices for examining
6          #pragma omp parallel for schedule(dynamic)
7          for all vertex $p$ in $X$ do
8              calculate $N_p^\varepsilon$ 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 $N_p^\varepsilon$ 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 $N_p^\varepsilon$ 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 $g \in SN_q$
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 unprocessed-border vertices that belong to at least two super nodes
27     sort $S$ in the descending order according to the number of super-nodes
28     while $S$ is not empty do
29         remove a set $X$ of $\beta$ 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 position $i$ and $i + 1$ of $SN_p$
40                 if $Findset(sn(u)) \neq Findset(sn(v))$ then
41                     #pragma omp critical
42                     $Union(\ Findset(sn(u)), Findset(sn(v)))$
43     /* Step 3: Merging weakly-related super-nodes */
44     find the representative super-nodes for all vertices
45     build the set $T$ of unprocessed-border or unprocessed-core or processed-core vertices
46     sort $T$ in the descending order according to the vertex degrees
47     while $T$ is not empty do
48         remove a set $X$ of $\beta$ fist vertices for examining
49         #pragma omp parallel for schedule(dynamic)
50         for all vertex $p$ in $X$ do
51             check if $p$ should be pruned from the core check
52             perform the core check on $p$
53         #pragma omp parallel for schedule(dynamic)
54         for all vertex $p$ in $X$ do
55             if $p$ is not a core then continue
56             for all vertex $q$ in $N_p$ do
57                 if $q$ is not a core then continue end if
58                 if $Findset(clu(p)) \neq Findset(clu(q))$ then
59                     if $\sigma(p, q) \geq \varepsilon$ then
60                         #pragma omp critical
61                         $Union(\ Findset(clu(p)), Findset(clu(q)))$
62     /* Step 4: Determining border vertices */
63     #pragma omp parallel for schedule(dynamic)
64     for all vertex $p$ in $L$ do
65         check if $p$ is a border or a true noise
66 EndFunction
```

Fig. 4. Pseudocode for anySCAN using OMP

in Section IV-B). In our experiments, the aggregate runtime for sequential parts of anySCAN is negligible and strongly dominated by that of the parallel parts. Thus, it achieves very good scalability w.r.t. the number of used threads. We will describe each step of anySCAN in details below.

Step 1: Summarization. Naively parallelizing Step 1 will result in many synchronizations among threads due to the overlaps of super-nodes. For avoiding this, 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 for next steps. We also mark the state of p as described in Section III-A. Obviously, vertices can be processed independently by threads. Thus, there is no conflict and consequently no explicit synchronization required here, except a barrier at the end. Second (Line 10-15), for each vertices p, we examine their neighbors

q and mark their 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*. Obviously, there will be no conflict assigning the state of q at this point. 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) (which is roughly 200 times faster than a lock or a critical section in OpenMP). Moreover, the overlapping among super-nodes is very small, which means that the effect of atomic operation here is negligible. If q is an *unprocessed-border* and has more than μ neighbors, it will be marked as *unprocessed-core*. We can easily see that conflict will also not happen. Last (Line 16-24), we use sequential algorithm for storing the super-nodes as well as merging some of them as described in Section III-A since these works are highly sequential. However, the overall runtime of this part is very negligible thus has very small effect on the scalability of anySCAN as demonstrated in Section IV-B. Concretely, we only need one atomic operation and one barrier at the end, which is much more efficient than using locks for synchronizing the states of objects. Note that the atomic operation is only used during the processing of blocks, and thus has very negligible effect on anySCAN.

Step 2: Merging strongly-related super-nodes. Similar to Step 1, we separate Step 2 into 2 parts for avoiding synchronizations. From Line 30-33, we perform the core check on each object p in X independently using multiple threads and store the results for next parts. From Line 34-42, we check each object independently and merge related super-nodes when it is necessary as described in Section III-A. Here, the Union operation is not thread-safe and must be executed using a lock or a critical section of OMP. However, as described above, the number of calls to Union of anySCAN is very small and thus it still has very good scalability at the end.

Step 3: Merging weakly-related super-nodes. Similar to 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 operation inside a critical section. Since this is similar to Step 2, we skip the details for saving space.

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

At the end of this Step, all noise vertices are examined to see if they are hubs or outliers. Obviously, these checks can be executed in parallel by multiple threads without explicit synchronizations.

C. Algorithm Analysis

Correctness. We first prove the correctness of anySCAN.

Lemma 4: The final results of anySCAN are identical to those of SCAN.

Id	Graph	Vertices	Edges	$\bar{d}$	c
GR01	ego-Gplus	107,614	13,673,453	127.06	0.4901
GR02	soc-LiveJournal1	4,847,571	68,993,773	14.23	0.2742
GR03	soc-Poket	1,632,803	30,622,564	18.75	0.1094
GR04	com-Orkut	3,072,441	117,185,083	38.14	0.1666
GR05	kron_g500-logn21	2,097,152	182,082,942	86.82	0.1649

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

Proof: (Sketch) 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 $p \lhd x_1 \cdots \lhd x_i \rhd \cdots \rhd x_n \rhd q$ (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 (Lemma 2 and Lemma 3). 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. Consequently, 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. The time complexity of anySCAN is $O(min(N_p, N_q) \cdot |E| + t \cdot f(t))$ in the worst case, where $t \ll |E|$ (see Figure 12 for experimental results) is the number of Findset and Union Operation and $f(t)$ is the extremely slowly growing inverse of the single-valued Ackermann function [5], [14]. Thus, AnySCAN also has the same worst-case complexity as pSCAN and SCAN as proven in [5]. However, since anySCAN uses much fewer structural similarity calculations than SCAN, it is consequently faster than SCAN. 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.

D. Optimizations

We introduce some optimization techniques for further speeding up anySCAN inspired by [5].

Lemma 5: 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: 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 $\sigma(p,q) = (\sum_{r \in N_p \cap N_q} w_{pr} \cdot w_{qr}) / \sqrt{(\sum_{r \in N_p} w_{pr}^2) \cdot (\sum_{r \in N_q} w_{qr}^2)} < \epsilon$. ■

For each vertex p, w_p and l_p are fixed and can be calculated in a preprocessing step in $O(|N_p|)$ time. The equation $\hat{\sigma}(p,q)^2 < \epsilon^2 \cdot l_p \cdot l_q$ can be verified in $O(1)$ time. If it is *true*, we do not need to calculate the structural similarity $\sigma(p,q)$ since $\sigma(p,q)$ will be surely smaller than ϵ. 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 immediately for further reducing runtime.

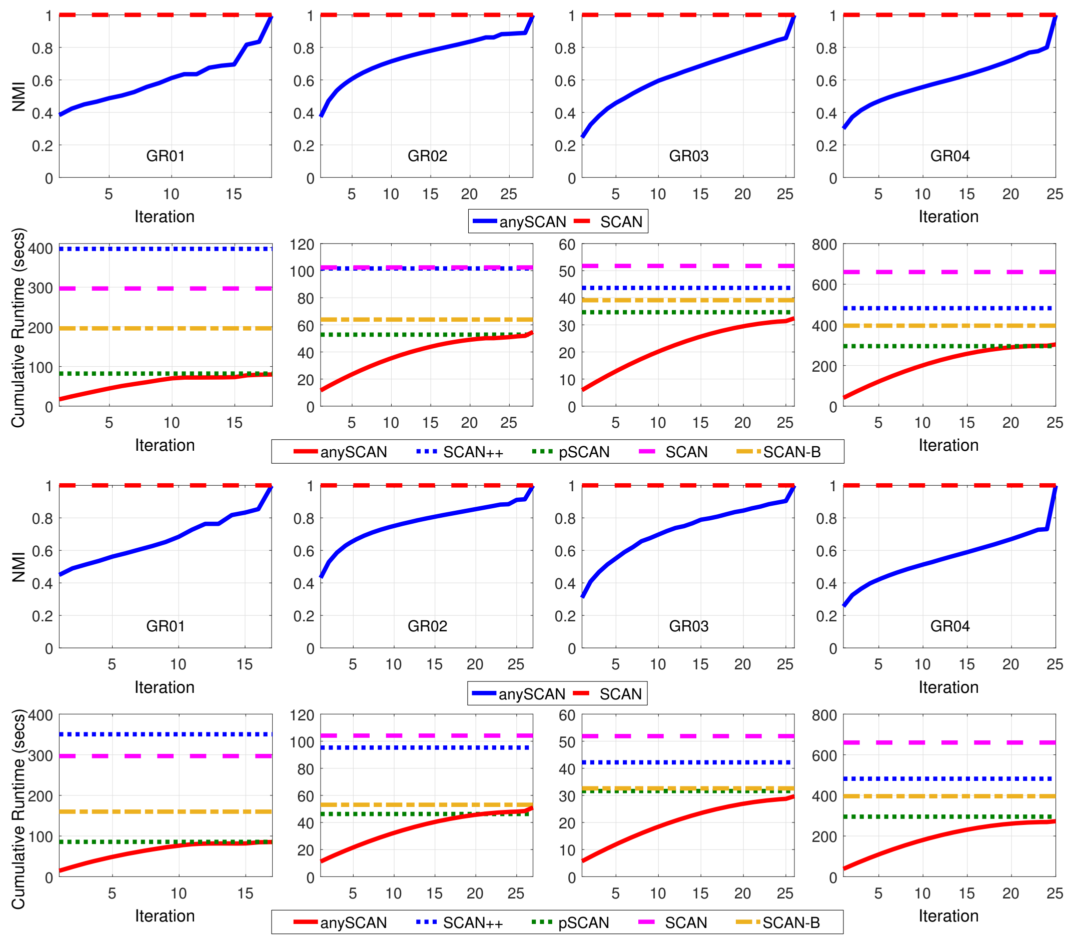

Fig. 5. NMI scores and runtimes of the *anytime* algorithm anySCAN during its execution in comparison with other *batch* algorithms (represented by horizontal lines) for GR01 to GR04 with $\epsilon = 0.5$ (top) and $\epsilon = 0.6$ (bottom)

IV. 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[3] 3.1. We use five large datasets acquired from the Stanford Network Analysis Project (SNAP)[4] [16], The UF Sparse Matrix Collection[5], and the Laboratory of Web Algorithmics[6] [17] for evaluating our algorithms. These datasets are summarized in Table I. Unless otherwise stated, we use default parameters $\mu = 5$, $\epsilon = 0.5$, and $\alpha = \beta = 8192$.

A. 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 II. We also introduce SCAN-B, an extension of SCAN using optimization techniques described in Section III-D. Extension details are omitted due to space limitation. For evaluating the anytime property of anySCAN, we use the results of SCAN as ground truths and Normalized Mutual Information (NMI) scores [18] 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.

Anytime properties. Figure 5 shows the cumulative runtimes and NMI scores of anySCAN for GR01 to GR04 measured at some arbitrary iterations of Steps 1 to 3 (arbitrary time points) of anySCAN. As we can see, the clustering qualities of anySCAN improve overtime and converge toward the results of SCAN at the end (indicating by $NMI \approx 1.0$). Taking GR02 ($\epsilon = 0.5$) as an example, anySCAN acquires the NMI scores of 0.53, 0.71, 0.8 after 17.79, 35.39, and 48.2 seconds, respectively, and ends with the same result as SCAN after 54.7 seconds. The longer it is run, the better the NMI scores it obtains, i.e., its results are more similar to those of SCAN.

Since anySCAN is an anytime one, it can be stopped at arbitrary time points for approximating results as well as saving computation costs. For example, one can stop anySCAN with good NMI scores ≈ 0.5 after 24.08, 13.94, 8.91, and 168.06 seconds for GR01 to GR04 and acquire acceleration factors of up to 14.55 times compared to other batch algorithms

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

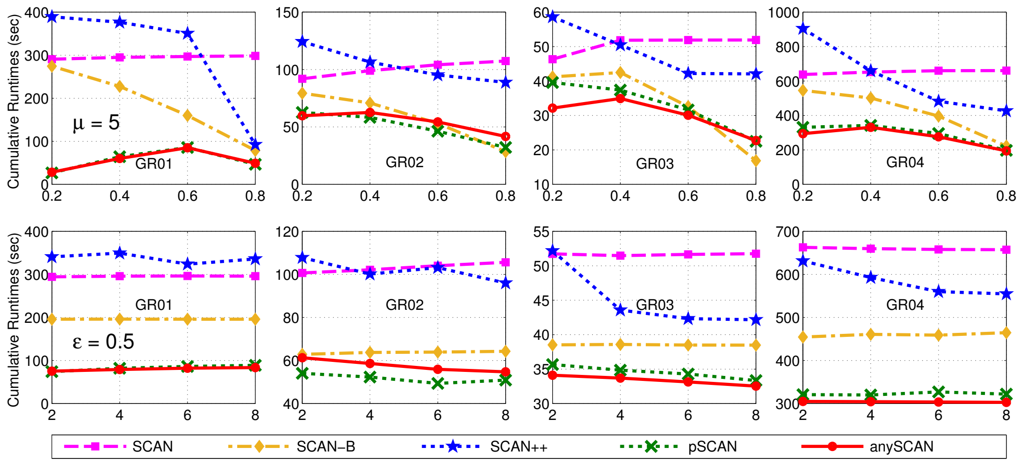

Fig. 6. Final runtimes of different algorithms w.r.t. different parameters ϵ (top) and μ (bottom)

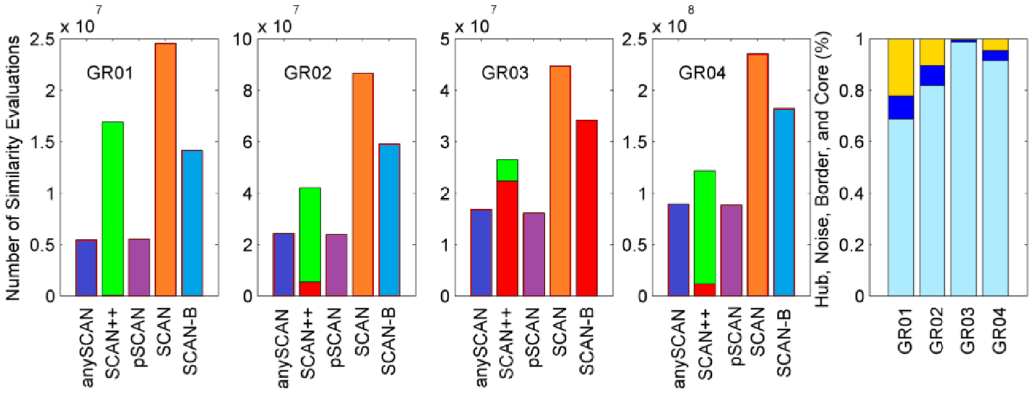

Fig. 7. (Left) Numbers of structural similarity calculations for all algorithms. For SCAN++, numbers of true similarity (bottom) and similarity sharing (top) evaluations are both plotted. (Right) Numbers of hub and outlier (light blue), border (dark blue), and core (yellow) vertices (from bottom to the top) for all datasets

with $\epsilon = 0.6$. As described in previous Sections, this anytime property makes anySCAN an interactive algorithm. During its execution, users can suppress it for analyzing the intermediate results, resume it for finding better results, or stop it whenever they are satisfied with acquired results. Moreover, anySCAN will fit well in many information systems with limited time constraints or those with fast response requirements. To the best of our knowledge, none of the existing techniques for speeding up SCAN has these anytime properties.

Compared with the batch algorithms, even if anySCAN is run to the end, its final cumulative runtimes are slightly faster than the fastest *batch* algorithm pSCAN in most cases. Taking GR04 ($\epsilon = 0.6$) as an example, anySCAN requires 273.0 seconds at the end, which is slightly faster than pSCAN with 295.2 seconds. This is interesting since anytime algorithms are usually slower (sometimes much slower) than batch ones because they incur additional costs for maintaining their anytime properties. We will study the overall performances of anySCAN and others below.

Overall performance. In Figure 6, we further compare the final cumulative runtimes of anySCAN and others w.r.t. different parameters μ ($\epsilon = 0.5$) and ϵ ($\mu = 5$). 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 ($\bar{d} = 38.1$). Overall, 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 it even slower than SCAN in some cases due to its additional overheads of calculating and maintaining the DTARs. The 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 quite simple, most structure similarity calculations are skipped due to the filtering property described in Lemma 5, especially when ϵ is very high.

Why is anySCAN more efficient than others? Figure 7 shows the numbers of structural similarity evaluations for all algorithms and thus further clarifies the acquired results above. For all datasets, pSCAN and anySCAN use almost the same number of similarity calculations, which are much smaller than those of other methods. The numbers of similarity sharing calculations of SCAN++ are clearly correlated with the numbers of core vertices. The higher the numbers of core vertices, the higher the numbers of similarity sharing SCAN++ uses, meaning that the similarity evaluation time

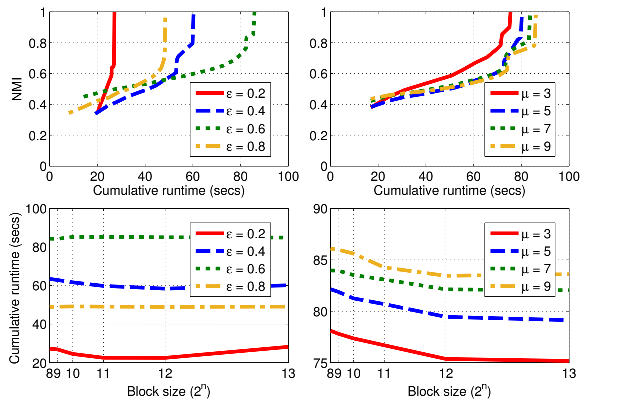

Fig. 8. The effect of parameters μ and ϵ (left) and block sizes $\alpha = \beta$ (right) for GR01

will be reduced. However, it also means that SCAN++ incurs more overhead for expanding its DTAR clusters. Sometimes, this overhead overwhelms the similarity sharing benefit, thus making SCAN++ slower than SCAN-B (though it uses fewer calculations), e.g. on the datasets GR02 and GR04.

Parameter analysis. The effect of parameter ϵ on anySCAN is shown in Figure 8 (top) for the dataset GR01. 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 makes super nodes to be merged earlier, and thus makes the intermediate results of anySCAN to come closer to the final one earlier.

The effect of block size parameters α and β is also quite clear as demonstrated in Figure 8 (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 $\mu = 5$, 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, is very small for all datasets as well as different values of μ and ϵ. This means that the performance of anySCAN is very stable w.r.t. the block sizes α and β.

Performance on synthetic graphs. Table II summarizes some synthetic graphs created by LFR bench mark graphs [19]. Here 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 as 100.

Figure 9 shows the performance of pSCAN and anySCAN

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 II
SYNTHETIC GRAPH DATASETS ($\bar{d}$ IS AVERAGED VERTEX DEGREES AND c IS AVERAGED CLUSTER COEFFICIENTS)

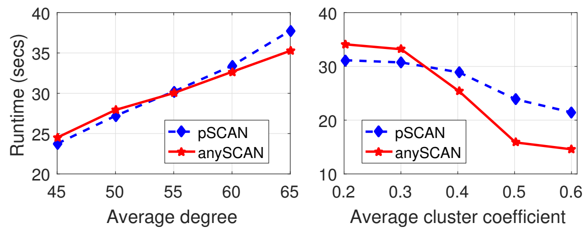

Fig. 9. Performance on synthetic graphs

on these synthetic graphs. When the number of edges (indicated by the average vertex degree) increases, the runtimes of both algorithm increase (see Figure 9 left) since more structural similarity needs to be performed. 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 (see Figure 9 right). 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 the improvement of the overall performance.

B. *Multicore Anytime SCAN*

To the best of our knowledge, anySCAN is the first parallel approach for SCAN on shared memory architecture. Moreover, it is *uniquely* a parallel and anytime algorithm at the same time. In this Section, we will study these aspects in details for real datasets GR01 to GR04 with the default parameters $\alpha = \beta = 32768$. We measure the scalability of anySCAN with multiple threads over its single thread version. Note that since the parallel overhead of anySCAN is negligible, the final cumulative runtimes of non-parallel version and a single thread version of anySCAN are almost the same.

Anytime properties. Since anySCAN is *uniquely* an anytime and parallel algorithm at the same time, it is interesting to see how it scales with the number of threads during its execution. As shown in Figure 10 (left), anySCAN scales very well at each iteration of its anytime scheme. Taking the dataset GR01 as an example, 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 reduces to 8.61. Thus,

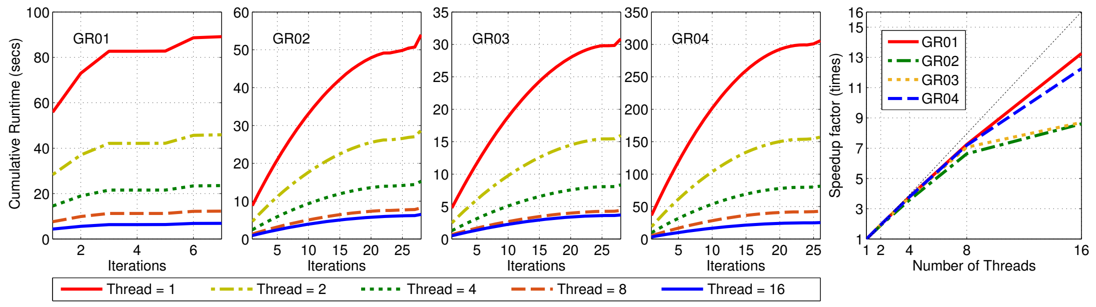

Fig. 10. Cumulative runtimes of anySCAN after each iteration of its *anytime* scheme for different numbers of threads (left) and the final runtime scalability (right)

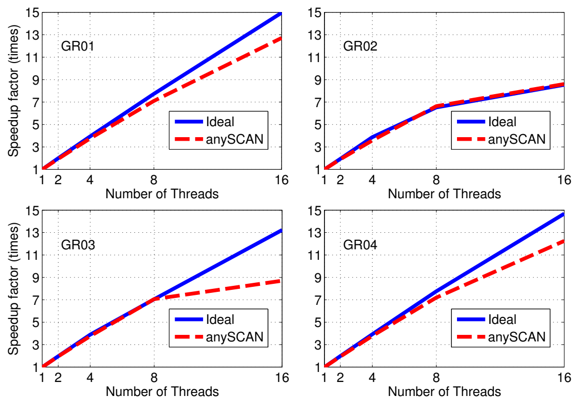

Fig. 11. Speed up factors of anySCAN and an ideal algorithm w.r.t. different numbers of threads

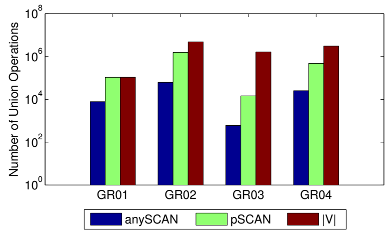

Fig. 12. The numbers of performed Union operations for GR01 to GR04

the earlier a user stops the algorithm, the higher the speed up factor she enjoys.

Figure 10 (right) shows the final speedup factors of anySCAN w.r.t. different numbers of threads. As we can see, anySCAN scales very well with the numbers of threads during its execution, e.g., almost linear for GR01, GR04, and GR05. For GR01 as an example, the speedup factors over single thread are 1.93, 3.78, 7.24, and 13.25 for 2, 4, 8, and 16 threads, respectively. For GR04 and GR05 using 16 threads, the speed up factors are 12.25 and 11.41, respectively. The speed up factors on GR02 and GR03 are worse than those on GR01, GR04, and GR05. 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 and GR04 (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 11 for further studies). In this case, increasing the block size values will help to solve the problem (see Figure 13 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 11. The ideal algorithm only calculates the structural similarities (without optimizations) of all edges of G which is the most expensive part of SCAN and obviously can be calculated independently with each other by multiple threads, and ignore the label propagation process among vertices. Obviously, it does not require synchronizations among threads and thus has an ideal scalability w.r.t. the number of threads. Note that existing efficient techniques for enhancing SCAN such as 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 including GR01, GR04, and especially GR02. Since the degrees of vertices vary significantly in GR02 and the NUMA effect, both the ideal algorithm and anySCAN suffer from performance degradation due to the load balancing problem as explained above. Thus, they end up having the same scalability. GR03 is an exception case where the performance of anySCAN suddenly drops on 16 threads while the idea algorithm still goes well. Here optimization techniques for speeding up the structural similarity calculation cause the problem. Most calculations are filtered out earlier following Lemma 5. Though this makes the algorithm much more efficient, this lowers the workloads 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, consequently reducing the scalability following the Amdahl's law.

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 12 for dataset GR01 to GR04. Since pSCAN uses the Disjoint Set data structure like anySCAN, we include it here for a comparison though it is not a parallel algorithm. As we see, pSCAN uses much less numbers of Union operations than the numbers of vertices $|V|$ of G. And, anySCAN uses even much 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,

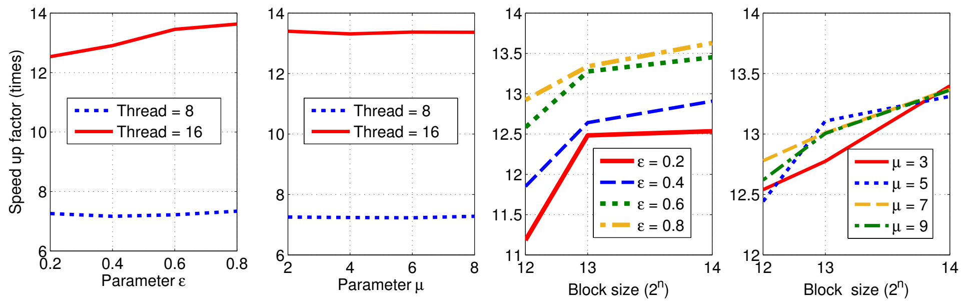

Fig. 13. The effect of parameters μ, ϵ, and block sizes on the scalability of anySCAN for the dataset GR01

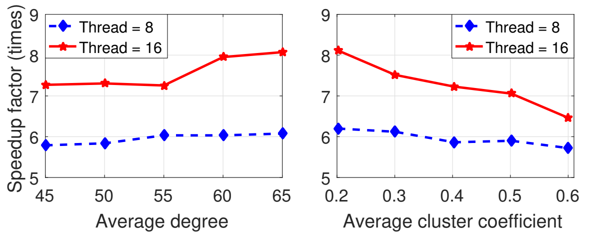

Fig. 14. Scalability on synthetic graphs

leaving only fewer ones to be executed in Step 2 and 3 inside critical sections. And fewer operations means better scalability anySCAN has as shown in Figure 10. This also implies that naively parallelizing pSCAN might lead to worse performance than anySCAN due to its much larger overheads of the Union operations.

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. Thus, increasing μ and ϵ will lead to increasing speed up factors as shown in Figure 13 (left) 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 13 (right).

Performance on synthetic graphs. Generally, when the average degree become larger, the overall scalability of anySCAN also goes up since the number of structural similarities as well as the time for evaluating them are both increased. On the other hand, when the average cluster coefficient is high, the overlap among the neighborhoods of vertices is also increased, thus leading to more conflicts during Step 2 and 3 of anySCAN. This reduces the overall scalability. Figure 14 shows the scalability of anySCAN on multiple threads when varying the average degrees and cluster coefficients. Thought there are some small fluctuations, the above trends are generally observed in most cases.

V. Related works and discussions

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 attractive research topic with many proposed techniques and extensions. For example, SCOT [20], HintClus [20], and gSkeletonClu [21] aim to solve the parameter setting problem of SCAN. The algorithm DENGRAPH [22] is an incremental clustering algorithm designed to detect communities in large and dynamic social networks. DHSCAN [23] and AHSCAN [24] are not density-based algorithms, but divisive hierarchical 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* [25] improves the efficiency of SCAN by using an edge sampling technique for reducing the number of structural similarity evaluations. However, it only approximates the result of SCAN. Recent techniques like SCAN++ [4] and pSCAN [5] not only acquire close performance [4] but also produce the exact clustering results of SCAN.

pSCAN [5] is a state-of-the-art technique proposed recently. 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 helps to reduce the calculation, thus making pSCAN one of the fastest variants of SCAN so far. The final cumulative runtimes of anySCAN are almost similar to those of pSCAN in many cases. Moreover, anySCAN has the power of approximation techniques and exact techniques at the same time in its anytime scheme. In additional, anySCAN can be efficiently parallelized, thus making it a *work-efficient* scalable parallel method while parallelizing pSCAN is a non-trivial problem.

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. These steps somehow have similar goals as Step 1 and 2 of anySCAN, though anySCAN has a completely different twist. 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 $weak$ separably as well as processing only core vertices as super-nodes and leaving the noise vertices to a post processing step for efficiency.

Last, noise vertices are examined in the post processing step separatedly from the whole algorithm. In the end, anySCAN is not only more efficient but also is both $anytime$ and $parallel$ technique. Note that the first step of SCAN++ is highly sequential and is responsible for most of the runtime of SCAN++, thus making it hard for parallelizing efficiently on multi-core CPUs.

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 [26], [27]. Thus naively tranforming distributed algorithms to shared memory architectures will obviously be very inefficient as pointed out by many previous researches, e.g., [28]. To the best of our knowledge, there is no parallel algorithm for SCAN on shared memory architectures such as multicore CPUs proposed in the literature so far. 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.

VI. Conclusion

In this paper, we propose a unique approach for accelerating the structural graph clustering algorithm SCAN. Our technique, called anySCAN, is an anytime method, which 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 the performance. To the best of our knowledge, anySCAN is the first anytime and parallel structural graph clustering algorithm. Experiments show that anySCAN has very good performance on large graph datasets in terms of its anytime scheme. It also scales very well with the number of threads under shared memory architectures such as multicore CPUs.

Acknowledgment

We special thank Prof. Lijun Chang for providing us with the binary file of pSCAN, Kenneth S. Bøgh, Panagiotis Bouros, and Prof. Hiroaki Shiokawa for their helps and/or useful discussions. We also special thanks to anonymous reviewers for their fruitful comments. Part of this research was funded by a Villum postdoc fellowship.

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. Zilberstein, "Using Anytime Algorithms in Intelligent Systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.

[8] T. Kobayashi, M. Iwamura, T. Matsuda, and K. Kise, "An Anytime Algorithm for Camera-Based Character Recognition," in *ICDAR*, 2013, pp. 1140–1144.

[9] 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.

[10] S. T. Mai, I. Assent, and A. Le, "Anytime OPTICS: An Efficient Approach for Hierarchical Density-Based Clustering," in *DASFAA*, 2016, pp. 164–179.

[11] 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.

[12] S. T. Mai, X. He, J. Feng, and C. Böhm, "Efficient Anytime Density-based Clustering," in *SDM*, 2013, pp. 112–120.

[13] 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.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[15] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[17] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[18] M. J. Zaki and W. M. Jr, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[19] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, p. 046110, Oct 2008.

[20] D. Bortner and J. Han, "Progressive clustering of networks using Structure-Connected Order of Traversal," in *ICDE*, 2010, pp. 653–656.

[21] 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.

[22] T. Falkowski, A. Barth, and M. Spiliopoulou, "DENGRAPH: A Density-based Community Detection Algorithm," in *Web Intelligence*, 2007, pp. 112–115.

[23] 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.

[24] ——, "AHSCAN: Agglomerative Hierarchical Structural Clustering Algorithm for Networks," in *ASONAM*, 2009, pp. 72–77.

[25] 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.

[26] S. Tatikonda and S. Parthasarathy, "Mining Tree-Structured Data on Multicore Systems," *PVLDB*, vol. 2, no. 1, pp. 694–705, 2009.

[27] 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.

[28] 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.