ScaleSCAN: Scalable Density-Based Graph Clustering

Hiroaki Shiokawa[1,2(✉)], Tomokatsu Takahashi[3], and Hiroyuki Kitagawa[1,2]

[1] Center for Computational Sciences, University of Tsukuba, Tsukuba, Japan
{shiokawa,kitagawa}@cs.tsukuba.ac.jp
[2] Center for Artificial Intelligence Research,
University of Tsukuba, Tsukuba, Japan
[3] Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Japan
shihakata@kde.cs.tsukuba.ac.jp

Abstract. How can we efficiently find clusters (*a.k.a.* communities) included in a graph with millions or even billions of edges? Density-based graph clustering SCAN is one of the fundamental graph clustering algorithms that can find densely connected nodes as clusters. Although SCAN is used in many applications due to its effectiveness, it is computationally expensive to apply SCAN to large-scale graphs since SCAN needs to compute all nodes and edges. In this paper, we propose a novel density-based graph clustering algorithm named *ScaleSCAN* for tackling this problem on a multicore CPU. Towards the problem, ScaleSCAN integrates *efficient node pruning methods* and *parallel computation schemes* on the multicore CPU for avoiding the exhaustive nodes and edges computations. As a result, ScaleSCAN detects exactly same clusters as those of SCAN with much shorter computation time. Extensive experiments on both real-world and synthetic graphs demonstrate that the performance superiority of ScaleSCAN over the state-of-the-art methods.

Keywords: Graph mining · Density-based clustering
Manycore processor

1 Introduction

How can we efficiently find clusters (*a.k.a.* communities) included in a graph with millions or even billions of edges? Graph is a fundamental data structure that has helped us to understand complex systems and schema-less data in the real-world [1,7,13]. One important aspect of graphs is cluster structures where nodes in the same cluster have denser edge-connections than nodes in the different clusters. One of the most successful clustering method is density-based clustering algorithm, named *SCAN*, proposed by Xu *et al.* [20]. The main concept of SCAN is that densely connected nodes should be in the same cluster; SCAN excludes nodes with sparse connections from clusters, and SCAN classifies them

S. Hartmann et al. (Eds.): DEXA 2018, LNCS 11029, pp. 18–34, 2018.
https://doi.org/10.1007/978-3-319-98809-2_2

as either hubs or outliers. In contrast to most traditional clustering algorithms such as graph partitioning [19], spectral algorithm [14], and modularity-based method [15] that only study the problem of the cluster detection and so ignore hubs and outliers, SCAN successfully finds not only clusters but also hubs and outliers. As a result, SCAN has been used in many applications [5,12].

Although SCAN is effective in finding highly accurate results, SCAN has a serious weakness; it requires high computational costs for large-scale graphs. This is because SCAN has to find all clusters prior to identifying hubs and outliers; it finds densely connected subgraphs as clusters. It then classifies the remaining non-clustered nodes into hubs or outliers. This clustering procedure entails exhaustive density evaluations for all adjacent node pairs included in the large-scale graphs. Furthermore, in order to evaluate the density, SCAN employs a criteria, called *structural similarity*, that incurs a set intersection for each edge. Thus, SCAN requires $O(m^{1.5})$ in the worst case [3].

Existing Approaches and Challenges: To address the expensive time-complexity of SCAN, many efforts have been made for the recent few years, especially in the database and data mining communities. One of the major approaches is nodes/edge pruning: *SCAN++* [16] and *pSCAN* [3] are the most representative methods. Although these algorithms certainly succeeded in reducing the time complexity of SCAN for the real-world graphs, the computation time for large-scale graphs (*i.e.* graphs with more than 100 million edges) is still large. Thus, it is a challenging task to improving the computational efficiency for the structural graph clustering. Especially, most of existing approaches perform as a single-threaded algorithms; they do not fully exploit parallel computation architectures but this is time-consuming.

Our Approaches and Contributions: We focus on the problem of speeding up SCAN for large-scale graphs. We present a novel parallel-computing algorithm, *ScaleSCAN*, that is designed to efficiently perform on shared memory architectures with the multicore CPU. The modern multicore CPU equips a lot of physical cores on a chip, and each core highlights vector processing units (VPUs) for powerful data-parallel processing, *e.g.,* SIMD instructions. Thus, ScaleSCAN employs *thread-parallel algorithm* and *data-parallel algorithm* in order to fully exploit the performance of the multicore CPU. In addition, we also integrates existing node pruning techniques [3] and our parallel algorithm. By pruning unnecessary nodes in the parallel computation manner, we attempt to achieve further improvement of the clustering speed. As a result, ScaleSCAN has the following attractive characteristics:

1. **Efficient:** Compared with the existing approaches [3,16,18], ScaleSCAN achieves high speed clustering by using the above approaches for density computations; ScaleSCAN can avoid computing densities for the whole graph.
2. **Scalable:** ScaleSCAN shows near-linear speeding up as increasing of the number of threads. ScaleSCAN is also scalable to the dataset size.
3. **Exact:** While our approach achieves efficient and scalable clustering, it does not to sacrifice the clustering accuracy; it returns exact clusters as SCAN.

Our extensive experiments showed that ScaleSCAN runs ×500 faster than SCAN without sacrificing the clustering quality. Also, ScaleSCAN achieved from ×17.3 to ×90.2 clustering speed improvements compared with the state-of-the-art algorithms [3,18]. In specific, ScaleSCAN can compute graphs, which have more than 1.4 billion edges, within 6.4 s while SCAN did not finish even after 24 h. Even though SCAN is effective in enhancing application quality, it has been difficult to apply SCAN to large-scale graphs due to its performance limitations. However, by providing our scalable approach that suits the identification of clusters, hubs and outliers, ScaleSCAN will help to improve the effectiveness of a wider range of applications.

Organization: The rest of this paper is organized as follows: Sect. 2 describes a brief background of this work. Section 3 introduces our proposed approach ScaleSCAN, and we report the experimental results in Sect. 4. In Sect. 5, we briefly review the related work, and we conclude this paper in Sect. 6.

2 Preliminary

We first briefly review the baseline algorithm SCAN [20]. Then, we introduce the data-parallel computation scheme that we used in our proposal.

2.1 The Density-Based Graph Clustering: SCAN

The density-based graph clustering SCAN [20] is one of the most popular graph clustering method; it successfully detects not only clusters but also hubs and outliers unlike traditional algorithms. Given an unweighted and undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, SCAN detects not only the set of clusters $\mathbb{C}$ but also the set of hubs H and outliers O at the same time. We denote the number of nodes and edges in G by $n = |V|$ and $m = |E|$, respectively.

SCAN extracts clusters as the sets of nodes that have dense internal connections; it identifies the other non-clustered nodes as hubs or outliers. Thus, prior to identifying hubs and outliers, SCAN finds all clusters in a given graph G. SCAN assigns two adjacent nodes into a same cluster according to how strong the two nodes are densely connected with each other through their shared neighborhoods. Let N_u be a set of neighbors of node u, so called *structural neighborhood* defined in Definition 1, SCAN evaluates *structural similarity* between two adjacent nodes u and v defined as follows:

Definition 1 (Structural neighborhood). *The structural neighborhood of a node u, denoted by N_u, is defined as $N_u = \{v \in V | (u, v) \in E\} \cup \{u\}$.*

Definition 2 (Structural similarity). *The structural similarity $\sigma(u, v)$ between node u and v is defined as $\sigma(u, v) = |N_u \cap N_v| / \sqrt{d_u d_v}$, where $d_u = |N_u|$ and $d_v = |N_v|$.*

Algorithm 1. Baseline algorithm: $\mathsf{SCAN}(G, \epsilon, \mu)$ [20]

```
1: for each edge (u, v) ∈ E do
2:     Compute σ(u, v) by Definition 2;
3: ℂ = ∅;
4: for each unvisited node u ∈ V do
5:     C = {u};
6:     for each unvisited node v ∈ C do
7:         if |N_v^ε| ≥ μ then
8:             C = C ∪ N_v^ε;
9:             Mark v as visited;
10:    if |C| ≥ 2 then
11:        ℂ = ℂ ∪ C;
```

We denote nodes u and v are *similar* if $\sigma(u, v) \geq \epsilon$; otherwise, the nodes are *dissimilar*.

SCAN detects a special class of node, called *core node*, that plays as the seed of a cluster, and SCAN then expands the cluster from the core node. Given a similarity threshold $\epsilon \in \mathbb{R}$ and a minimum size of a cluster $\mu \in \mathbb{N}$, core node is a node that has μ neighbors with a structural similarity that exceeds the threshold ϵ.

Definition 3 (Core node). *Given a similarity threshold $0 \leq \epsilon \leq 1$ and an integer $\mu \geq 2$, a node u is a core node iff $|N_u^\epsilon| \geq \mu$. Note that N_u^ϵ, so called ϵ-neighborhood of u, is defined as $N_u^\epsilon = \{v \in N_u | \sigma(u, v) \geq \epsilon\}$.*

When node u is a core node, SCAN assigns all nodes in N_u^ϵ to the same cluster as node u, and it expands the cluster by checking whether each node in the cluster is a core node or not.

Definition 4 (Cluster). *Let a node u be a core node that belongs to a cluster $C \in \mathbb{C}$, the cluster C is defined as $C = \{w \in N_v^\epsilon | v \in C\}$, where C is initially set to $C = \{u\}$.*

Finally, SCAN classifies non-clustered nodes (*i.e.* nodes that belong to no clusters) as hubs or outliers. If a node u is not in any clusters and its neighbors belong to two or more clusters, SCAN regards node u as a hub, and it is an outlier otherwise. Given the set of clusters, it is straightforward to obtain hubs and outliers in $O(n + m)$ time. Hereafter, we thus focus on only extracting the set of clusters in G.

Algorithm 1 overviews the pseudo code of SCAN. SCAN first evaluates structural similarities for all edges in G, and then constructs clusters by traversing all nodes. As proven in [3], Algorithm 1 is essentially based on the problem of triangle enumeration on G since each node $w \in \{N_u \cap N_v\} \backslash \{u, v\}$ forms a triangle with u and v when we compute $\sigma(u, v) = |N_u \cap N_v| / \sqrt{d_u d_v}$. This triangle enumeration basically involves $O(\alpha(G) \cdot m)$, where $\alpha(G)$ is the arboricity of G such that $\alpha(G) \leq \sqrt{m}$. Thus, the time complexity of SCAN is $O(m^{1.5})$ and is *worst-case optimal* [3].

2.2 Data-Parallel Instructions

In our proposed method, we employ the data-parallel computation schemes [17] for improving clustering speed. Thus, we briefly introduce the data-parallel instructions.

Data-parallel instructions are the fundamental instructions included in modern CPUs (*e.g.,* SSE, AVX, AVX2 in x86 architecture). By using the data-parallel instructions, we can perform the same operation on multiple data elements simultaneously. CPU usually loads only one element into for each CPU register in *non*-data-parallel computation scheme, whereas the data-parallel instructions enables to load *multiple elements* for each CPU register, and simultaneously perform an operation on the loaded elements.

The maximum number of elements that can be loaded on a register is determined by the size of the register and each element. For example, if a CPU supports 126-bit wide registers, we can load four 32-bit integers for each register. Also, CPUs with AVX2 and AVX-512 enable to perform eight and 16 integers simultaneously since the CPUs have 256-bit and 512-bit wide registers, respectively.

3 Proposed Method: ScaleSCAN

Our goal is to find exactly the same clustering results as those of SCAN from large-scale graphs within short runtimes. In this section, we present details of our proposal, ScaleSCAN. We first overview the ideas underlying ScaleSCAN and then give a full description of our proposed approaches.

3.1 Overview

The basic idea underlying ScaleSCAN is to reduce the computational cost for the structural similarity computations from algorithmic and parallel processing perspectives. Specifically, we first integrate the node pruning algorithms [3] into massively parallel computation scheme on the modern multicore CPU. We then propose the data-parallel algorithm for each structural similarity computation for further improving the clustering efficiency. By combining the node pruning and parallel computing nature, we design ScaleSCAN so as to compute only necessary pairs of nodes.

Algorithm 2 shows the pseudocode of ScaleSCAN. For efficiently detecting nodes that can be pruned, we maintain two integer values sd (*similar-degree*) [3] and ed (*effective-degree*) [3]. Formally, sd and ed are defined as follows:

Definition 5 (Similar-degree). *The similar-degree of node u, denoted* `sd[u]`*, is the number of neighbor nodes in N_u that have been determined to be structure-similar to node u, i.e., $\sigma(u, v) \geq \epsilon$ for $v \in N_u$.*

Definition 6 (Effective-degree). *The effective-degree of node u, denoted* `ed[u]`*, is d_u minus the number of neighbor nodes in $N[u]$ that have been determined to be not structure-similar to node u, i.e., $\sigma(u, v) < \epsilon$ for $v \in N_u$.*

Algorithm 2. Proposed algorithm: ScaleSCAN(G, ϵ, μ)

▷ **Step 0: Initialization**
1: **for each** node $u \in V$ **do in thread-parallel**
2: $sd[u] \leftarrow 0$, and $ed[u] \leftarrow d_u$;

▷ **Step 1: Pre-pruning**
3: **for each** edge $(u, v) \in E$ **do in thread-parallel**
4: Get $L[(u, v)]$ by Definition 7;
5: **if** $L[(u, v)] \neq$ *unknown* **then** UpdateSdEd$(L[(u, v)])$;
6: $E^{\text{unknown}} \leftarrow \{(u, v) \in E | L[(u, v)] = \textit{unknown}\}$

▷ **Step 2: Core detection**
7: **for each** $(u, v) \in E^{\text{unknown}}$ **do in thread-parallel**
8: **if** $sd[u] < \mu$ and $ed[u] \geq \mu$ **then**
9: $L[(u, v)] \leftarrow$PStructuralSimilarity$((u, v), \epsilon)$;
10: UpdateSdEd$(L[(u, v)])$;
11: $E^{\text{core}} \leftarrow \{(u, v) \in E | sd[u] \geq \mu$ and $sd[v] \geq \mu\}$;

▷ **Step 3: Cluster construction**
12: **for each** $(u, v) \in E^{\text{core}}$ **do in thread-parallel**
13: **if** find$(u) \neq$ find(v) **then**
14: **if** $L[(u, v)] =$ *unknown* **then** $L[(u, v)] \leftarrow$PStructuralSimilarity$((u, v), \epsilon)$;
15: **if** $L[(u, v)] =$ *similar* **then** cas_union(u, v);
16: $E^{\text{border}} \leftarrow \{(u, v) \in E \backslash E^{\text{core}} | sd[u] \geq \mu$ or $sd[v] \geq \mu\}$;
17: **for each** $(u, v) \in E^{\text{border}}$ **do in thread-parallel**
18: **if** find$(u) \neq$ find(v) **then**
19: **if** $L[(u, v)] =$ *unknown* **then** $L[(u, v)] \leftarrow$PStructuralSimilarity$((u, v), \epsilon)$;
20: **if** $L[(u, v)] =$ *similar* **then** cas_union(u, v);

In the beginning of ScaleSCAN shown in Algorithm 2 (Lines 1–2), ScaleSCAN first initializes sd and ed for all nodes. By comparing the two values sd and ed, we determine whether a node should be prune or not in the thread-parallel manner. We describe the details of the node pruning techniques based on sd and ed in Sect. 3.3.

After the initialization, the algorithm consists of three main thread-parallel steps: (Step 1) pre-pruning, (Step 2) core detection, and (Step 3) cluster construction. In the pre-pruning, ScaleSCAN first reduces the size of given graph G in the thread-parallel manner; it prunes edges from E what are obviously either similar or dissimilar without computing the structural similarity. Then, ScaleSCAN extracts all core nodes in the core detection step that is the most time-consuming part in the density-based graph clustering. In order to reduce the computation time for the core detection, ScaleSCAN combines the nodes pruning techniques proposed by Chang *et al.* [3] and the thread-parallelization using the multicore processor. In addition, for further improving the efficiency of the core detection step, we also propose a novel structural similarity computation technique, named PStructuralSimilarity, by using the data-parallel instructions. Finally, in the cluster construction step, ScaleSCAN finds clusters based on

Algorithm 3. UpdateSdEd($L[(u,v)]$)

```
1: if L[(u,v)] = similar then
2:     sd[u] ← sd[u] + 1 with atomic operation;
3:     sd[v] ← sd[v] + 1 with atomic operation;
4: else if L[(u,v)] = dissimilar then
5:     ed[u] ← ed[u] − 1 with atomic operation;
6:     ed[v] ← ed[v] − 1 with atomic operation;
```

Definition 4 by employing union-find tree shown in Sect. 3.4. In the following sections, we describe the details of each thread-parallel step.

3.2 Pre-pruning

In this step, ScaleSCAN reduces the size of graph G by removing $(u,v) \in E$ what can be either $\sigma(u,v) \geq \epsilon$ or $\sigma(u,v) < \epsilon$ without computing the structural similarity defined in Definition 2. Specifically, let $(u,v) \in E$, we always have $\sigma(u,v) \geq \epsilon$ when $\frac{2}{\sqrt{d_u d_v}} \geq \epsilon$ since $|N_u \cap N_v| \geq 2$ from Definition 1. Meanwhile, we also have $\sigma(u,v) < \epsilon$ when $d_u < \epsilon^2 d_v$ (or $d_v < \epsilon^2 d_u$), because if $d_u < \epsilon^2 d_v$ then $\sigma(u,v) < \frac{d_u}{\sqrt{d_u d_v}} < \epsilon$. Clearly, we can check both $\frac{2}{\sqrt{d_u d_v}} \geq \epsilon$ and $d_u < \epsilon^2 d_v$ (or $d_v < \epsilon^2 d_u$) in $O(1)$. Thus, we can efficiently remove such edges from a given graph.

Based on the above discussion, we maintain *edge similarity label* $L[(u,v)]$ for each edge $(u,v) \in E$; an edge (u,v) takes one of the three edge similarity labels, *i.e., similar*, *dissimilar*, and *unknown*.

Definition 7 (Edge similarity label). *Let $(u,v) \in E$, ScaleSCAN assigns the following edge similarity label $L[(u,v)]$ for (u,v):*

$$L[(u,v)] = \begin{cases} similar & (if\ \frac{2}{\sqrt{d_u d_v}} \geq \epsilon) \\ dissimilar & (if\ d_u < \epsilon^2 d_v\ \ or\ d_v < \epsilon^2 d_u) \\ unknown & (Otherwise) \end{cases} \tag{1}$$

If an edge (u,v) is determined to have $\sigma(u,v) \geq \epsilon$ or $\sigma(u,v) < \epsilon$, we assign $L[(u,v)]$ as *similar* or *dissimilar*, respectively; otherwise, we label the edge as *unknown*. If $L[(u,v)] = unknown$, we can not verify the edge becomes $\sigma(u,v) \geq \epsilon$ or not without computing its structural similarity. Thus, we compute the structural similarity only for $E^{\text{unknown}} = \{(u,v) \in E | L[(u,v)] = unknown\}$ in the subsequent procedure.

The pseudocode of the pre-pruning step is shown in Algorithm 2 (Lines 3–6). In this step, we assign each edge to each thread on the multicore CPU. For each edge (u,v) (Line 3), we first apply Definition 7, and obtain the edge similarity label $L[(u,v)]$ (Line 4). If $L[(u,v)] \neq unknown$, we invoke UpdateSdEd($L[(u,v)]$) (Line 5) for updating sd and ed values according to $L[(u,v)]$ (Lines 1–6 in Algorithm 3). Note that sd and ed are shared by all threads, and thus UpdateSdEd($L[(u,v)]$) has a possibility to cause write conflicts. Hence, to avoid the write

conflicts, we use atomic operation (*e.g.,* `omp atomic` in OpenMP) for updating sd and ed values (Lines 2–3 and Lines 5–6 in Algorithm 3). After the pre-pruning procedure, we extract a set of edges E^{unknown} whose edge similarity label are *unknown* (Line 6).

3.3 Core Detection

As we described in Sect. 2, core detection step is the most time-consuming part since the original algorithm SCAN needs to compute all edges in E. Thus, to speed up the core detection step, we propose a thread-parallel algorithm with the node pruning and data-parallel similarity computation method PStructuralSimilarity.

(1) Thread-Parallel Node Pruning: The pseudocode of the thread-parallel node pruning is shown in Algorithm 2 (Lines 7–12). Algorithm 2 (Lines 7–12) detects all core nodes included in G by using the node pruning technique in the thread-parallel manner. As shown in (Line 7) in Algorithm 2, we first assign each edge in E^{unknown} to each thread. In the threads, we compute the structural similarity only for the nodes such that (1) they have not been core or non-core, and (2) they have a possibility to be a core node. Clearly, if $sd[u] \geq \mu$ then node u satisfies the core node condition shown in Definition 3, and also if $ed[u] < \mu$ then node u never satisfies the core node condition; otherwise, we need to compute structural similarities between node u and its neighbor nodes to determine whether node u is core node or not. Hence, once we determine node u is either core or non-core, we stop to compute structural similarities between node u and its neighbor nodes (Line 6). Meanwhile, in the case of $sd[u] < \mu$ and $ed[u] \geq \mu$ (Line 6), we compute structural similarities for node u by PStructuralSimilarity (Line 7), and we finally update sd and ed by UpdateSdEd according to $L[(u, v)]$ (Line 8).

(2) Data-Parallel Similarity Computation: In the structural similarity computation, we propose a novel algorithm PStructuralSimilarity for further improving the efficiency of the core detection step. As we described in Sect. 2.2, each physical core on the modern multicore CPU equips the data-parallel instructions [17] (*e.g.,* SSE, AVX, AVX2 in x86 architecture); data-parallel instructions enable to compute multiple data elements simultaneously by using a single instruction. Our proposal, PStructuralSimilarity, reduces the computation time consumed in the structural similarity computations by using such data-parallel instructions.

Algorithm 4 shows the pseudocode of PStructuralSimilarity. For ease of explanation, we hereafter suppose that 256-bit wide registers are available, and we use 32-bit integer for representing each node in Algorithm 4. That is, we can pack eight nodes into each register. In addition, we suppose that nodes in N_u are stored in ascending order, and we denote $N_u[i]$ to specify i-th element in N_u. Given an edge (u, v) and the parameter ϵ, Algorithm 4 returns whether

$L[(u,v)]$ = *similar* or *dissimilar* based on the structural similarity $\sigma(u,v)$. In the structural similarity computations, the set intersection (*i.e.,* $|N_u \cap N_v|$) is obviously the most time-consuming part since it requires $O(\min\{d_u, d_v\})$ for obtaining $\sigma(u,v) = \frac{|N_u \cap N_v|}{\sqrt{d_u d_v}}$ while the other part (*i.e.,* $\sqrt{d_u d_v}$) can be done in $O(1)$. Hence, in PStructuralSimilarity, we employ the data-parallel instructions to improve the set intersection efficiency.

Algorithm 4 (Lines 6–11) shows our *data-parallel set intersection* algorithm that is consisted of the following three phases:

Phase 1. We load α and β nodes from N_u and N_v as blocks, respectively, and pack the blocks into the 256-bit wide registers, $\texttt{reg}_u$ and $\texttt{reg}_v$ (Lines 7–8). Since we need to compare all possible $\alpha \times \beta$ pairs of nodes in the data-parallel manner, we should select α and β so that $\alpha \times \beta = 8$. That is, we have only two choices: $\alpha = 8$ and $\beta = 1$, or $\alpha = 4$ and $\beta = 2$. Thus, we set $\alpha = 8$ and $\beta = 1$ if d_u and d_v are significantly different, otherwise $\alpha = 4$ and $\beta = 2$ (Lines 2–5). dp_load_permute permute nodes in the blocks in the order of permutation arrays π_α and π_β.

Example. If we have sets of loaded nodes $\{u_1, u_2, u_3, u_4\}$ and a permutation array $\pi_\alpha = [4,3,2,1,4,3,2,1]$, dp_load_permute$(\pi_\alpha, \{u_1, u_2, u_3, u_4\})$ loads $[u_4, u_3, u_2, u_1, u_4, u_3, u_2, u_1]$ into $\texttt{reg}_u$. Also, dp_load_permute$(\pi_\beta, \{v_1, v_2\})$ loads $[v_2, v_2, v_2, v_2, v_1, v_1, v_1, v_1]$ into $\texttt{reg}_v$ for $\{v_1, v_2\}$ and $\pi_\beta = [2,2,2,2,1,1,1,1]$.

Phase 2. We compare the $\alpha \times \beta$ pairs of nodes by dp_compare in the data-parallel manner. dp_compare compares each pair of nodes in the corresponding position of $\texttt{reg}_u$ and $\texttt{reg}_v$. If each pair of nodes has same node it then outputs 1, otherwise 0.

Example. Let $\texttt{reg}_u = [u_4, u_3, u_2, u_1, u_4, u_3, u_2, u_1]$ and $\texttt{reg}_v = [v_2, v_2, v_2, v_2, v_1, v_1, v_1, v_1]$, where $u_1 = v_1$ and $u_2 = v_2$, dp_compare outputs $[0,0,1,0,0,0,0,1]$.

Phase 3. We update the blocks (Lines 10–11) and repeat these phases until we can not load any blocks from N_u or N_v (Line 6).

After the termination, we count the number of common nodes $\triangle$ by (Line 12) in Algorithm 4. Finally, we obtain $L[(u,v)]$ based on $\triangle \geq \epsilon\sqrt{d_u d_v}$ or not (Lines 13–16).

3.4 Cluster Construction

ScaleSCAN finally constructs clusters in the thread-parallel manner. For efficiently maintaining clusters, we use *union-find tree* [4], which can efficiently keep set of nodes partitioned into disjoint clusters. The union-find tree supports two fundamental operations: find(u) and union(u,v). find(u) is an operation to check which cluster does node u belong to, and union(u,v) merges two clusters, which are node u and v belong to. It is known that each operation can be done in $\Omega(A(n))$ where A is Ackermann function, thus we can check and merge clusters efficiently.

Algorithm 4. PStructuralSimilarity$((u, v), \epsilon)$

▷ **Step 0: Initialization**
1: $\triangle \leftarrow 0$, $p_u \leftarrow 0$, $p_v \leftarrow 0$, and $\texttt{reg}_{add} \leftarrow$ dp_load$([0,0,0,0,0,0,0,0])$;
2: **if** $d_u > 2d_v$ (or $d_v > 2d_u$) **then**
3: $\quad \alpha = 8$, $\beta = 1$, $\pi_\alpha \leftarrow [1,2,3,4,5,6,7,8]$, and $\pi_\beta \leftarrow [1,1,1,1,1,1,1,1]$;
4: **else**
5: $\quad \alpha = 4$, $\beta = 2$, $\pi_\alpha \leftarrow [4,3,2,1,4,3,2,1]$, and $\pi_\beta \leftarrow [1,1,1,1,2,2,2,2]$;

▷ **Step 1: Data-parallel set intersection**
6: **while** $p_u < d_u$ and $p_v < d_v$ **do**
7: $\quad \texttt{reg}_u \leftarrow$ dp_load_permute$(\pi_\alpha, [N_u[p_u], \cdots, N_u[p_u + \alpha - 1]])$;
8: $\quad \texttt{reg}_v \leftarrow$ dp_load_permute$(\pi_\beta, [N_u[p_v], \cdots, N_u[p_v + \beta - 1]])$;
9: $\quad \texttt{reg}_{add} \leftarrow$ dp_add$(\texttt{reg}_{add},$ dp_compare$(\texttt{reg}_u, \texttt{reg}_v))$;
10: $\quad$ **if** $N_u[p_u + \alpha - 1] \geq N_v[p_v + \beta - 1]$ **then** $p_v \leftarrow p_v + \beta$;
11: $\quad$ **if** $N_u[p_u + \alpha - 1] \leq N_v[p_v + \beta - 1]$ **then** $p_u \leftarrow p_u + \alpha$;

▷ **Step 2: Edge similarity label assignment**
12: $\triangle \leftarrow \triangle +$ dp_horizontal_add$(\texttt{reg}_{add})$;
13: **if** $\triangle < \epsilon\sqrt{d_u d_v}$ **then** $\triangle \leftarrow \triangle + |\{N_u[p_u], \ldots, N_u[d_u]\} \cap \{N_v[p_v], \cdots, N_v[d_v]\}|$;
14: **if** $\triangle \geq \epsilon\sqrt{d_u d_v}$ **then** $L[(u,v)] = \textit{similar}$;
15: **else** $L[(u,v)] = \textit{dissimilar}$;
16: **return** $L[(u,v)]$;

Algorithm 2 (Lines 12–20) shows our parallel cluster construction. We first constructs clusters by using only core nodes (Lines 12–15), and then we attach non-core nodes to the clusters (Lines 16–20). Recall that this clustering process is done in the thread-parallel manner. For avoiding conflicts among multiple threads, we thus propose a multi-threading aware union operation, cas_union(u, v). can_union employs *compare-and-swap (CAS)* atomic operation [8] before merging two clusters.

4 Experimental Analysis

We conducted extensive experiments to evaluate the effectiveness of ScaleSCAN. We designed our experiments to demonstrate that:

- **Efficient and Scalable:** ScaleSCAN outperforms the state-of-the-art algorithms pSCAN and SCAN-XP by over one order of magnitude for all datasets. Also, SacaleSCAN is scalable to the number of threads and edges (Sect. 4.2).
- **Effectiveness:** The key techniques of ScaleSCAN, parallel node-pruning and data-parallel similarity computation, are effective for improving the clustering speed on large-scale graphs (Sect. 4.3).
- **Exactness:** Regardless of parallel nodes pruning techniques, ScaleSCAN always returns exactly same clustering results as those of SCAN (Sect. 4.4).

Table 1. Statistics of real-world datasets

Dataset name	# of nodes	# of edges	Data source
DB	317,080	1,049,866	com-DBLP [9]
LJ	4,847,571	68,993,773	soc-livejournal1 [9]
OK	3,072,441	117,185,083	com-orkut [9]
FS	65,608,366	141,874,960	com-friendster [9]
WB	118,142,155	1,019,903,190	webbase-2001 [2]
TW	41,652,230	1,468,365,182	twitter-2010 [2]

4.1 Experimental Setup

We compared ScaleSCAN with the baseline method SCAN [20], the state-of-the-art *sequential* algorithm pSCAN [3], and the state-of-the-art *thread-parallel* algorithm SCAN-XP [18]. All algorithms were implemented in g++ using `-O3` option[1]. All experiments were conducted on a CentOS server with an Intel(R) Xeon(R) E5-2690 2.60 GHz GPU and 128 GB RAM. The CPU has 14 physical cores, we thus used threads for up to 14 in the experiments. Since each physical core equips 256-bit wide registers, 256-bit wide data-parallel instructions were also available. Unless otherwise stated, we used default parameters $\epsilon = 0.4$ and $\mu = 5$.

Datasets: We evaluated the algorithms on six real-world graphs, which are downloaded from the Stanford Network Analysis Platform (SNAP) [9] and the Laboratory for Web Algorithmics (LAW) [2]. Table 1 summarizes the statistics of real-world datasets. In addition to the real-world graphs, we also used synthetic graphs generated by LFR benchmark [6], which is considered as the *de facto standard* model for generating graphs. The settings will be detailed later.

4.2 Efficiency and Scalability

Efficiency: In Fig. 1, we evaluated the clustering speed on the real-world graphs through wall clock time by varying ϵ. In this evaluation we used 14 threads for the thread-parallel algorithms, *i.e.,* ScaleSCAN and SCAN-XP. Note that SCAN did not finish its clustering for WB and TW with in 24 h, so we omitted the results from Fig. 1. Overall, ScaleSCAN outperforms SCAN-XP, pSCAN, and SCAN. On average, ScaleSCAN achieves $\times 17.3$ and $\times 90.2$ faster than the state-of-the-art methods SCAN-XP and pSCAN, respectively; also, ScaleSCAN is approximately $\times 500$ faster than the baseline method SCAN. In particular, ScaleSCAN can compute TW with 1.4 billion edges within 6.4 s. Although pSCAN slightly improves its efficiency as ϵ increases, these improvements are negligible.

In Fig. 2, we also evaluated the clustering speeds on FS by varying the parameter μ. As well as Fig. 1, we used 14 threads for ScaleSCAN and SCAN-XP. We

[1] We opened our source codes of ScaleSCAN on our website.

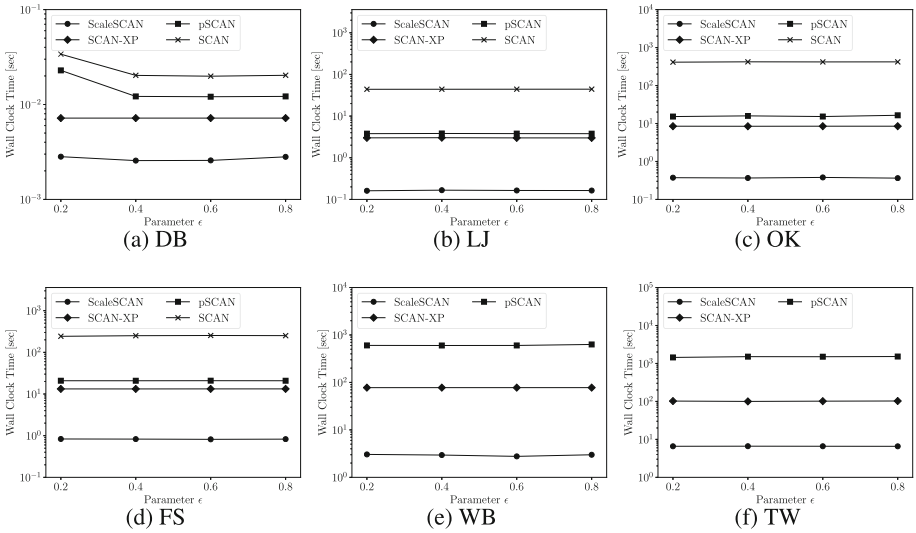


Fig. 1. Runtimes of each algorithm by varying ϵ.

omitted the results for the other datasets since they show very similar results to Fig. 2. As shown in Fig. 2, ScaleSCAN also outperforms the other algorithms that we examined even though ScaleSCAN and pSCAN slightly increase runtimes as μ increases.

Scalability: We assessed scalability tests of ScaleSCAN in Fig. 3a and b by increasing the number of threads and edges, respectively. In Fig. 3a, we used the real-world dataset TW. Meanwhile, in Fig. 3b, we generated four synthetic datasets by using LFR benchmark; we varied the number of nodes from 10^5 to 10^8 with the average degree 30. As we can see from Fig. 3, the runtimes of ScaleSCAN has near-linear in terms of the number of threads and edges. These results verify that ScaleSCAN is scalable for large-scale graphs.

4.3 Effectiveness of the Key Techniques

As mentioned in Sect. 3.3, we employed thread-parallel node pruning and data-parallel similarity computation to prune unnecessary computations. In the following experiments, we examined the effectiveness of the key techniques of ScaleSCAN.

Thread-Parallel Node Pruning. ScaleSCAN prunes nodes that have already been determined as core or non-core nodes in the thread-parallel manner. As mentioned in Sect. 3.3, ScaleSCAN specifies the nodes to be pruned by checking the two integer values sd and ed; ScaleSCAN prunes a node u from its subsequent procedure if $sd[u] \geq \mu$ or $ed[u] < \mu$ since it is determined as core or non-core, respectively.

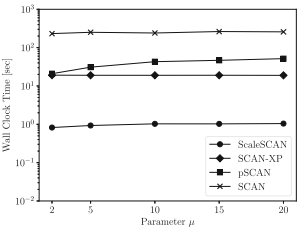
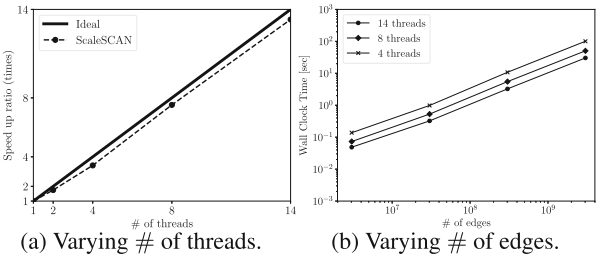


Fig. 2. Runtimes by varying μ on FS.



(a) Varying # of threads. (b) Varying # of edges.

Fig. 3. Scalability test.

To show the effectiveness, we compared the runtimes of ScaleSCAN with and without the node-pruning techniques. We set the number of threads as 14 for each algorithm. Figure 4 shows the wall clock time of each algorithm for the real-world graphs. Figure 4 shows that ScaleSCAN is faster than ScaleSCAN without the node pruning by over one order of magnitude for all datasets. These results indicate that the node pruning significantly contributes the efficiency of ScaleSCAN even though it requires several synchronization (*i.e.,* atomic operations) among threads for maintaining *sd* and *ed*.

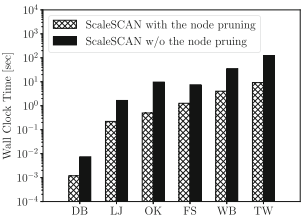


Fig. 4. Effects of the node pruning.

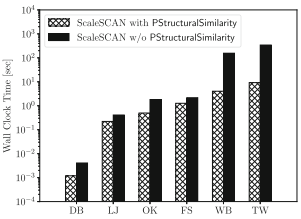


Fig. 5. Effects of PStructuralSimilarity.

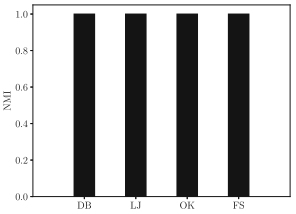
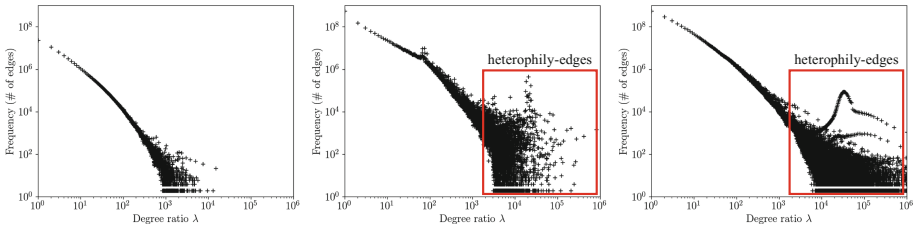


Fig. 6. Evaluate exactness of ScaleSCAN.

Data-Parallel Similarity Computation. As shown in Algorithm 4, ScaleSCAN computes the structural similarity by using the data-parallel algorithm PStructuralSimilarity. That is, ScaleSCAN compares two neighbor node sets N_u and N_v whether they share same nodes or not in the data-parallel manner. In order to confirm the impact of the data-parallel instructions, we evaluated the running time of a variant of ScaleSCAN that did not use PStructuralSimilarity for obtaining $\sigma(u, v)$.

Figure 5 shows the wall clock time comparisons between ScaleSCAN with and without using PStructuralSimilarity. As shown in Fig. 5, PStructuralSimilarity achieved significant improvements in several datasets, *e.g.* DB, OK, WB, and TW. On the other hand, the improvements seems to be moderated in LJ and FS. More specifically, ScaleSCAN is $\times 20$ faster than ScaleSCAN without PStructuralSimilarity on average for DB, OK, WB and TW. Meanwhile, ScaleSCAN is limited to approximately $\times 2$ improvements in LJ and FS.



(a) LJ (b) WB (c) TW

Fig. 7. Distribution of degree ratio $\lambda_{(u,v)}$

For further discussing about this point, we measured the degree ratio $\lambda_{(u,v)} = \max\{\frac{d_u}{d_v}, \frac{d_v}{d_u}\}$ of each edge $(u, v) \in E$ for LJ, WB, and TW. Figure 7 shows the distributions of the degree ratio for each dataset; horizontal and vertical axis show the degree ratio $\lambda_{(u,v)}$ and the number of edges with the corresponding ratio. In Fig. 7, we can observe that WB has large number of edges with large $\lambda_{(u,v)}$ values while LJ does not have such edges. This indicates that, differ from LJ, edges in WB prefer to connect nodes with different size of degree. Here, let us say an edge with large $\lambda_{(u,v)}$ value as *heterophily-edge*, PStructuralSimilarity can perform efficiently if a graph has many heterophily-edges. This is because that, as shown in Algorithm 4 (Lines 2–3), we can load a lot of nodes from N_u (or N_v) to the 256-bit wise registers at the same time since we set $\alpha = 8$ and $beta = 1$ for the heterophily-edges. In addition, by setting such imbalanced α and $beta$, PStructuralSimilarity is expected to terminate earlier since the while loop in Algorithm 4 (Lines 6–11) stops when $p_u \geq d_u$ or $p_v \geq d_v$. As a result, PStructuralSimilarity thus performs efficiently for the heterophily-edges.

We observed that large-scale graphs tend to have a lot of heterophily-edges because their structure grows more complicated when the graphs become more larger. For example, TW shown in Fig. 7c has a peak around $\lambda_{(u,v)} = 10^5$ values (heterophily-edges), and ScaleSCAN gains large improvements on this dataset (Fig. 5). Thus, these results imply that our approach is effective for large-scale graphs.

4.4 Exactness of the Clustering Results

Finally, we experimentally confirm the exactness of clustering results produced by ScaleSCAN. In order to measure the exactness, we employed the information-theoretic metric, *NMI (normalized mutual information)* [11], that returns 1 if two clustering results are completely same, otherwise 0. In Fig. 6, we compared the clustering results produced by the original method SCAN and our proposed method ScaleSCAN. Since SCAN did not finish in WB and TW within 24 h, we omitted the results from Fig. 6. As we can see from Fig. 6, ScaleSCAN shows 1 for all conditions we examined. Thus, we experimentally confirmed that ScaleSCAN produces exactly same clustering results as those of SCAN.

5 Related Work

The original density-based graph clustering method SCAN requires $O(m^{1.5})$ times and it is known as worst-case optimal [3]. To address the expensive time-complexity, many efforts have been made for the recent few years, especially from sequential and parallel computing perspectives. Here, we briefly review the most successful algorithms.

Sequential Algorithms. One of the major approaches for improving clustering speed is the node/edge pruning techniques: SCAN++ [16] and pSCAN [3] are the representative algorithms. SCAN++ is designed to handle the property of real-world graphs; a node and its two-hop-away nodes tend to have lots of common neighbor nodes since real-world graphs have high clustering coefficients [16]. Based on this property, SCAN++ effectively reduces the number of structural similarity computations. Chang *et al.* proposed pSCAN that employs a new paradigm based on the observations in real-world graphs [3]. By following the observations, pSCAN employs several the nodes pruning techniques and their optimizations for reducing the number of structural similarity computations. To the best of our knowledge pSCAN is the state-of-the-art sequential algorithm that achieves high performance and exact clustering results at the same time. However, SCAN++ and pSCAN ignore the thread-parallel and the data-parallel computation schemes, and thus their performance improvements are still limited. Our work is different from these algorithms in that provides not only the node pruning techniques but also both thread-parallel and data-parallel algorithms. Our experimental analysis in Sect. 4 show that ScaleSCAN is approximately $\times 90$ faster clustering than pSCAN.

Parallel Algorithms. In a recent few years, several thread-parallel algorithms have been proposed for improving the clustering speed of SCAN. To the best of our knowledge, AnySCAN [10], proposed by Son *et al.* in 2017, is the first solution that performs SCAN algorithm on the multicore CPUs. Similar to SCAN++ [16], they applied randomized algorithm in order to avoid unnecessary structural similarity computations. By performing the randomized algorithm in the thread-parallel manner, AnySCAN achieved almost similar efficiency on the multicore CPU compared with pSCAN [3]. Although AnySCAN is scalable on large-scale graphs, it basically produces approximated clustering results due to its randomized algorithm nature.

Takahashi *et al.* recently proposed SCAN-XP [18] that exploits massively parallel processing hardware for the density-based graph clustering. As far as we know, SCAN-XP is the state-of-the-art parallel algorithm that achieves the fastest clustering without sacrificing clustering quality for graphs with millions or even billions of edges. However, different from our proposed method ScaleSCAN, SCAN-XP does not have any node pruning techniques; it need to compute all nodes and edges included in a graph. As shown in Sect. 4, our ScaleSCAN is much faster than SCAN-XP; ScaleSCAN outperforms SCAN-XP by over one order of magnitude for the large datasets.

6 Conclusion

We developed a novel parallel algorithm ScaleSCAN for density-based graph clustering using the multicore CPU. We proposed thread-parallel and data-parallel approaches that combines parallel computation capabilities and efficient node pruning techniques. Our experimental evaluations showed that ScaleSCAN outperforms the state-of-the-art algorithms over one order of magnitude even though ScaleSCAN does not sacrifice its clustering qualities. The density-based graph clustering is now a fundamental graph mining tool to current and prospective applications in various disciplines. By providing our scalable algorithm, it will help to improve the effectiveness of future applications.

Acknowledgement. This work was supported by JSPS KAKENHI Early-Career Scientists Grant Number JP18K18057, JST ACT-I, and Interdisciplinary Computational Science Program in CCS, University of Tsukuba.

References

1. Arai, J., Shiokawa, H., Yamamuro, T., Onizuka, M., Iwamura, S.: Rabbit order: just-in-time parallel reordering for fast graph analysis. In: Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, pp. 22–31 (2016)
2. Boldi, P., Vigna, S.: The webgraph framework I: compression techniques. In: Proceedings of the 13th International Conference on World Wide Web, pp. 595–601 (2004)
3. Chang, L., Li, W., Qin, L., Zhang, W., Yang, S.: pSCAN: fast and exact structural graph clustering. IEEE Trans. Knowl. Data Eng. **29**(2), 387–401 (2017)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2009)
5. Ding, Y., et al.: atBioNet–an integrated network analysis tool for genomics and biomarker discovery. BMC Genom. **13**(1), 1–12 (2012)
6. Fortunato, S., Lancichinetti, A.: Community detection algorithms: a comparative analysis. In: Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools, pp. 27:1–27:2 (2009)
7. Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Ida, Y., Toyoda, M.: Adaptive message update for fast affinity propagation. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 309–318 (2015)
8. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
9. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection, June 2014. http://snap.stanford.edu/data
10. Mai, S.T., Dieu, M.S., Assent, I., Jacobsen, J., Kristensen, J., Birk, M.: Scalable and interactive graph clustering algorithm on multicore CPUs. In: Proceedings of the 33rd IEEE International Conference on Data Engineering, pp. 349–360 (2017)
11. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)

12. Naik, A., Maeda, H., Kanojia, V., Fujita, S.: Scalable Twitter user clustering approach boosted by personalized PageRank. In: Kim, J., Shim, K., Cao, L., Lee, J.-G., Lin, X., Moon, Y.-S. (eds.) PAKDD 2017. LNCS, vol. 10234, pp. 472–485. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57454-7_37
13. Sato, T., Shiokawa, H., Yamaguchi, Y., Kitagawa, H.: FORank: fast ObjectRank for large heterogeneous graphs. In: Companion Proceedings of the the Web Conference, pp. 103–104 (2018)
14. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Trans. Pattern Anal. Mach. Intell. **22**(8), 888–905 (2000)
15. Shiokawa, H., Fujiwara, Y., Onizuka, M.: Fast algorithm for modularity-based graph clustering. In: Proceedings of the 27th AAAI Conference on Artificial Intelligence, pp. 1170–1176 (2013)
16. Shiokawa, H., Fujiwara, Y., Onizuka, M.: SCAN++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. Proc. Very Large Data Bases **8**(11), 1178–1189 (2015)
17. Solihin, Y.: Fundamentals of Parallel Multicore Architecture, 1st edn. Chapman & Hall/CRC, Boca Raton (2015)
18. Takahashi, T., Shiokawa, H., Kitagawa, H.: SCAN-XP: parallel structural graph clustering algorithm on Intel Xeon Phi coprocessors. In: Proceedings of the 2nd International Workshop on Network Data Analytics, pp. 6:1–6:7 (2017)
19. Wang, L., Xiao, Y., Shao, B., Wang, H.: How to partition a billion-node graph. In: Proceedings of the IEEE 30th International Conference on Data Engineering, pp. 568–579 (2014)
20. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: SCAN: a structural clustering algorithm for networks. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 824–833 (2007)