# Parallelizing Pruning-based Graph Structural Clustering

Yulin Che
HKUST
Hong Kong, China
yche@cse.ust.hk

Shixuan Sun
HKUST
Hong Kong, China
ssunah@cse.ust.hk

Qiong Luo
HKUST
Hong Kong, China
luo@cse.ust.hk

## ABSTRACT

A common class of graph structural clustering algorithms, pioneered by SCAN (Structural Clustering Algorithm for Networks), not only find clusters among vertices but also classify vertices as cores, hubs and outliers. However, these algorithms suffer from efficiency issues due to the great amount of computation required on structural similarity among vertices. Pruning-based SCAN algorithms improve efficiency by reducing the amount of computation. Nevertheless, this structural similarity computation is still the performance bottleneck, especially on big graphs of billions of edges. In this paper, we propose to parallelize pruning-based SCAN algorithms on multi-core CPUs and Intel Xeon Phi Processors (KNL) with multiple threads and vectorized instructions. Specifically, we design ppSCAN, a multi-phase vertex computation based parallel algorithm, to avoid redundant computation and achieve scalability. Moreover, we propose a pivot-based vectorized set intersection algorithm for structural similarity computation. Experimental results show that ppSCAN is scalable on both CPU and KNL with respect to the number of threads. On the 1.8 billion-edge graph friendster, our ppSCAN completes within 65 seconds on KNL (64 physical cores with hyper-threading). This performance is 100x-130x faster than our single-threaded version, and up to 250x faster than pSCAN, the state-of-the-art sequential algorithm, on the same platform.

## CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**;

## KEYWORDS

Pruning-based Graph Structural Clustering, Multi-Phase Parallel Vertex Computation, Vectorized Set Intersection

**ACM Reference Format:**
Yulin Che, Shixuan Sun, and Qiong Luo. 2018. Parallelizing Pruning-based Graph Structural Clustering. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225063

## 1 INTRODUCTION

Given an undirected unweighted graph, a structural clustering algorithm, e.g., the SCAN algorithm [22], deterministically finds cores, hubs and their cluster labels based on adjacency and similarity between vertices. There are two input parameters in SCAN [22], namely the similarity threshold $\epsilon$ and the core threshold $\mu$. Two adjacent vertices are similar if their similarity value exceeds $\epsilon$, and a vertex with at least $\mu$ similar neighbors is a *Core*. Clusters are grown from cores to their similar neighbors, whereas vertices not in any cluster are further classified into hubs and outliers.

Since SCAN and its variant algorithms are able to identify exact clusters and classify vertices as cores, hubs and outliers, they have a number of applications, such as advertising and epidemiology. However, these algorithms suffer from efficiency issues on massive graphs, because they require exhaustive similarity computations among all adjacent vertices with each computation involving a set-intersection of two vertex arrays [22].

A number of existing algorithms, including sequential algorithms SCAN++ [18] and pSCAN [6], and parallel algorithms anySCAN [16] and SCAN-XP [19], have been proposed to accelerate SCAN [22]. Nevertheless, they either take an excessively long time or run out of memory on big graphs. For instance, in our experimental environment, the sequential pSCAN [6] took hours to analyze a twitter dataset and the sequential SCAN++ [18] could not finish within 24 hours. With parallelization, anySCAN [16] took 10 minutes to analyze the twitter dataset and ran out of memory (over 64GB) on the friendster dataset with 1.8 billion edges. Due to the lack of pruning, SCAN-XP [19] took 30 minutes to analyze the twitter dataset, even though it exploited both thread-level and instruction-level parallelism. As such, none of the existing algorithms can support online structural clustering on big graphs.

To address the performance problem, we propose a multi-phase vertex computation based parallel algorithm. Due to the data and order dependencies in the sequential pruning-based SCAN algorithm [6], we cannot directly parallelize it. Instead, we decompose the SCAN computation into two steps, namely role computing, and core and non-core clustering. To apply pruning techniques, we further separate each step into multiple phases and parallelize each phase in a lock-free manner. We bundle vertex computation into tasks and dynamically schedule tasks with a vertex degree-based task scheduler. The scheduler estimates workloads based on the vertex roles and the sum of vertex degrees. When the accumulated sum exceeds a threshold, a task is submitted, which helps to achieve load balance at a negligible cost.

In order to speed up the set-intersections for similarity computation, we propose a pivot-based vectorized set intersection algorithm. This algorithm reduces condition comparisons and utilizes vectorized instructions. Additionally, our set-intersection algorithm keeps the early-termination optimization introduced by pSCAN [6].

We have implemented our algorithm on both multi-core CPUs and Intel Xeon Phi Processors Knights Landing (KNL), and evaluated it on various datasets in comparison with a number of existing


*ICPP 2018, August 13–16, 2018, Eugene, OR, USA*

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225063

algorithms. Our optimized set intersection can achieve speedups of up to 4x over the original set-intersection algorithm on the twitter dataset. As a result, our ppSCAN achieved a speedup of over two orders of magnitude over the sequential pSCAN [6] on KNL, and is able to support interactive result exploration (with a response time of under a minute), on billion-edge graphs with a wide range of parameter values.

## 2 PRELIMINARY

We consider an unweighted and undirected graph $G = (V, E)$, define cores, clusters, hubs and outliers of SCAN [22], and give the problem statement.

*Definition 2.1.* The **neighborhood** of $u$, denoted as $N(u)$, is defined as: $N(u) = \{v | v \in V \wedge (u, v) \in E\}$. The **closed neighborhood** of $u$, denoted by $\Gamma(u)$, is defined as: $\Gamma(u) = N(u) \cup \{u\}$.

*Definition 2.2.* The **structural similarity predicate** between $u$ and $v$, denoted by $\sigma_\epsilon(u, v)$, is defined as follows:
$\sigma_\epsilon(u, v) = (\frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)||\Gamma(v)|}} \geq \epsilon)$.
This cosine similarity definition is widely adopted by SCAN and its variants [6, 16, 18, 19, 21, 22, 25, 26]. Denote the **degree** of $u$ as $d[u]$, then $\sigma_\epsilon(u, v)$ can be written as: $\sigma_\epsilon(u, v) = (|\Gamma(u) \cap \Gamma(v)| \geq \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$.

*Definition 2.3.* The **$\epsilon$-neighborhood** of $u$, denoted by $N_\epsilon(u)$, is defined as: $N_\epsilon(u) = \{v | v \in \Gamma(u) \wedge \sigma_\epsilon(u, v)\}$.

*Definition 2.4.* The **core predicate** of $u$, denoted by $Core_{\epsilon,\mu}(u)$, is defined as: $Core_{\epsilon,\mu}(u) = (|N_\epsilon(u)| \geq \mu + 1)$.

*Definition 2.5.* The **role** of $u$, denoted by $role[u]$, is labeled as: ***Core*** if $Core_{\epsilon,\mu}(u)$ is true; otherwise it is ***NonCore***.

*Definition 2.6.* The **direct structural reachability** between $u$ and $v$, denoted by $DSR_{\epsilon,\mu}(u, v)$, is defined as follows:
$DSR_{\epsilon,\mu}(u, v) = (Core_{\epsilon,\mu}(u) \wedge (v \in N_\epsilon(u)))$.

*Definition 2.7.* Given a length $l \geq 1$, the **vertex sequence**, denoted by $V_s$, is defined as: $V_s = [v_0, v_1, ..., v_i, ...v_{l-1}]$ where $|V_s| = l$, $V_s[i] = v_i$ and $v_i \in V$. The **structural reachability** between $u$ and $v$, denoted by $SR_{\epsilon,\mu}(u, v)$, is defined as follows:
$SR_{\epsilon,\mu}(u, v) = (\exists_{V_s} : (|V_s| \geq 2) \wedge (V_s[0] == u) \wedge (V_s[|V_s| - 1] == v) \wedge (\forall_{i \in [0, |V_s|-1)} : DSR(V_s[i], V_s[i + 1])))$.

*Definition 2.8.* The **structural connectivity** between $u$ and $v$, denoted by $SC_{\epsilon,\mu}(u, v)$, is defined as follows:
$SC_{\epsilon,\mu}(u, v) = (\exists_{w \in V} : SR_{\epsilon,\mu}(w, u) \wedge SR_{\epsilon,\mu}(w, v))$.

*Definition 2.9.* A **cluster** is a set of vertices $C$ that satisfies connectivity and maximality as follows:
- (Connectivity) $\forall_{u,v \in C} : SC_{\epsilon,\mu}(u, v)$;
- (Maximality) $\forall_{u \in C, v \in V} : (((u \in C) \wedge SC_{\epsilon,\mu}(u, v)) \rightarrow (v \in C))$.

*Definition 2.10.* (Hub and Outlier) A vertex $u$ not in any cluster can be classified into hub or outlier. $u$ is a **hub** if it satisfies $\exists_{v,w}$ : $(v \in N(u)) \wedge (w \in N(u)) \wedge$ (v and w are in different clusters))); otherwise, it is an **outlier**.

In this paper, we do not distinguish hubs and outliers, since they can be found by exploring all the neighbors of vertices not in any cluster with a time complexity $O(|E| + |V|)$, as stated in pSCAN [6].

**Algorithm 1:** SCAN [22]

```
Input: a graph G = (V, E), parameters 0 < ϵ ≤ 1 and μ ≥ 1
Output: roles role, clusters ℂ
1  foreach u ∈ V and role[u] == Unknown do
2      if (role[u] ← CheckCore(u)) == Core then
3          ℂ ← ℂ ∪ ExpandCluster(u)
4  Procedure ExpandCluster(u)
5      C = {u}, Q.Push(u)
6      while not Q.IsEmpty() do
7          v ← Q.Pop()
8          foreach w ∈ N(v) and sim[e(v, w)] == Sim do
9              C ← C ∪ {w}
10             if role[w] == Unknown then
11                 if (role[w] ← CheckCore(w)) == Core then
12                     Q.Push(w)
13     return C
14 Procedure CheckCore(u)
15     return |N_ϵ(u)| − 1 ≥ μ ? Core : NonCore
```

**Problem Statement**. Given a graph $G = (V, E)$, parameters $0 < \epsilon \leq 1$ and $\mu \geq 1$, compute the **roles** of all the vertices and the set $\mathbb{C}$ of all the **clusters** in $G$.

*Definition 2.11.* The **compressed spare row representation (CSR)** of a graph consists of edge and offset arrays, denoted as $dst$ and $off$, where $dst[off[u] : off[u + 1])$ represents u's neighbors. Let $e(u, v)$ denote the **offset of edge** $(u, v)$, then we have $e(u, v) \in [off[u], off[u + 1])$ and $dst[e(u, v)] = v$.

In this paper, we use CSR to represent the input graph where each vertex $u$'s neighbors $dst[off[u] : off[u + 1])$ are sorted, the same as in pSCAN [6].

*Definition 2.12.* The **similarity value** of an edge $(u, v)$, denoted by $sim[e(u, v)]$, is labeled as: ***Sim*** if $\sigma_\epsilon(u, v)$ is true; otherwise it is ***NSim***.

## 3 RELATED WORK

In this section, we briefly introduce SCAN [22], pSCAN [6] and other relevant clustering algorithms. In our work, we focus on parallelizing the pruning based SCAN, in specific, the pSCAN algorithm.

### 3.1 SCAN

*Definition 3.1.* The **structural similarity computation**, denoted by $CompSim(u, v)$, is to compute $sim[e(u, v)]$. According to Definition 2.2, $CompSim(u, v)$ is mainly to compute the intersection count $|\Gamma(u) \cap \Gamma(v)|$, since $\lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil$ is easy to compute.

*Definition 3.2.* The **core checking computation**, denoted by $CheckCore(u)$, is to determine the role of $u$, through computing $N_\epsilon(u)$. It involves $d[u]$ invocations of $CompSim(u, v)$, after which $sim[e(u, v)]$ is cached for the later cluster expansion.

Lemma 3.3. *The set of all vertices structurally reachable from a core vertex is a cluster [22].*

According to lemma 3.3, SCAN [22] (Algorithm 1), finds a *Core* $u$ that is not clustered yet and expands a cluster from $\{u\}$ in a breadth first search (BFS). When all the vertices are visited, their roles are determined and all the clusters are complete.

Theorem 3.4. *If the similarity computation $CompSim(u, v)$ adopts a merge-based set intersection method, the total workload of SCAN's structural similarity computations is* $2 \sum_{v \in V} d[v]^2$.

**Algorithm 2:** pSCAN [6]

```
   Input: a graph G = (V, E), parameters 0 < ϵ ≤ 1 and μ ≥ 1
   Output: roles role, clusters
1  InitDisjointSets()
2  foreach u ∈ V do
3  |  sd[u] ← 0, ed[u] ← d[u]
4  foreach u ∈ V in a non-increasing ed[u] order do
5  |  CheckCore(u)
6  |  if role[u] == Core then
7  |  |  ClusterCore(u)
8  InitClusterId(), ClusterNonCores()
9  Procedure CheckCore(u)
10 |  if sd[u] < μ and ed[u] ≥ μ then
11 |  |  foreach v ∈ N(u) and sim[e(u, v)] == Unknown do
12 |  |  |  sim[e(v, u)] ← sim[e(u, v)] ← CompSim(u, v)
13 |  |  |  Update sd[u], ed[u], sd[v], ed[v]
14 |  |  |  if sd[u] ≥ μ or ed[u] < μ then
15 |  |  |  |  break
16 |  role[u] ← sd[u] ≥ μ ? Core : NonCore
17 Procedure ClusterCore(u)
18 |  foreach v ∈ N(u) and sd[v] ≥ μ and not IsSameSet(u, v) do
19 |  |  if sim[e(u, v)] == Unknown then
20 |  |  |  sim[e(v, u)] ← sim[e(u, v)] ← CompSim(u, v)
21 |  |  |  Update sd[v], ed[v]
22 |  |  if sim[e(u, v)] == Sim then
23 |  |  |  Union(u, v)
```

Proof. Each $CompSim(u, v)$ requires $d[u]+d[v]$ comparisons in computing the set intersection $|\Gamma(u) \cap \Gamma(v)|$. A similarity computation between $u$ and $v$ is executed twice: one is for $CheckCore(u)$, and the other is for its neighbor $v$'s $CheckCore(v)$. Therefore, the total workload for exhaustive similarity checking is $2\sum_{v \in V} d[v]^2$. □

## 3.2 pSCAN

To reduce the amount of similarity computation, pSCAN [6] introduces the following two techniques: 1) min-max pruning and similarity reuse for the core checking, 2) union-find-set operations for the core clustering instead of a BFS based cluster expansion as in SCAN [22]. There are two steps of pSCAN (Algorithm 2): 1) the core checking and clustering step finalizes roles and clusters of cores (Lines 4-7); 2) the cluster id initialization and non-core clustering step produces the final clusters (Line 8).

Lemma 3.5. *Each core vertex belongs to only one cluster [6], i.e, clusters of cores are disjoint.*

Based on this lemma, pSCAN proposed to use union-find-set operations for the core clustering: union-find sets represent clusters of cores, and union-find operations the clustering of core $u$.

*Definition 3.6.* The **union-find-set** operation consists of the following: 1) finding the root of $u$'s set, denoted by ***FindRoot(u)***; 2) merging the sets that $u$ and $v$ are in, denoted by ***Union(u,v)***. We define ***IsSameSet(u,v)***, to check if $u$ and $v$ are currently in the same set: $IsSameSet(u, v) = (FindRoot(u) == FindRoot(v))$.

*Definition 3.7.* The **cluster id** of each union-find-set, denoted by $cluster\_id[FindRoot(u)]$, is the minimum core vertex id in the union-find-set.

*3.2.1 Pruning Techniques.* Min-max and similarity reuse techniques were adopted for the core checking; and union-find pruning was applied for the core clustering.

**1) Min-Max Pruning (for the Core Checking).** In order to terminate early in the core checking, pSCAN introduces similar and effective degrees as follows.

*Definition 3.8.* The **similar and effective** degrees of $u$, denoted by $sd[u]$ and $ed[u]$ are lower and upper bounds of $(|N_\epsilon(u)| - 1)$ ($sd[u] \le |N_\epsilon(u)| - 1 \le ed[u]$). Initially $sd[u] = 0$, $ed[u] = d[u]$, updated in the core checking.

pSCAN explores vertices in a non-increasing $ed[u]$ order. There are two early termination conditions of $CheckCore(u)$:

- $(sd[u] \ge \mu)$: return $Core$;
- $(ed[u] < \mu)$: return $NonCore$.

**2) Similarity Value Reuse (for the Core Checking).** For an undirected graph, the edge between $u$ and $v$ is stored twice (as $(u, v)$ and $(v, u)$). However, the similarity predicate values for $(u, v)$ and $(v, u)$ are the same. Thus, assigning $sim[e(u, v)]$ to the reverse edge $sim[e(v, u)]$ helps to avoid any redundant computation. The reverse edge offset computation for the $e(v, u)$ is a binary search of $u$ in $v$'s sorted neighbors. After the computation of $e(v, u)$, the assignment $sim[e(v, u)] \leftarrow sim[e(u, v)]$ can be made.

**3) Union-Find Pruning (for the Core Clustering).** When core $u$ and its neighboring core $v$ are already in the same set, further similarity computations among them can be avoided.

*3.2.2 Similarity Computation Optimization.* $CompSim(u, v)$ is to compute the intersection count $|\Gamma(u) \cap \Gamma(v)|$ and check whether it is greater than or equal to $\ge \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil$). Based on the definition of structural similarity predicate, early termination can be made with the intersection count bounds as follows.

*Definition 3.9.* The **intersection count bounds**, denoted as $du$, $dv$, $cn$ satisfy $cn \le |\Gamma(u) \cap \Gamma(v)| \le min(du, dv)$. Initially $du = d[u] + 2$, $dv = d[v] + 2$ and $cn = 2$, where 2 counts for $\{u, v\}$, since similarity computations of pSCAN only happen for adjacent vertices. There are three early termination conditions of $CompSim(u, v)$:

- $(dv < \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$: return $NSim$;
- $(du < \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$: return $NSim$;
- $(cn \ge \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$: return $Sim$.

The optimization of $CompSim(u, v)$ takes the same principle as min-max pruning for $CheckCore(u)$.

**Similarity Predicate Pruning.** The initial values of intersection bounds can be used for determining $sim[e(u, v)]$ without any intersection as follows:

- $sim[e(u, v)] \leftarrow NSim$, if $(d[u]+2 < \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$;
- $sim[e(u, v)] \leftarrow NSim$, if $(d[v]+2 < \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$;
- $sim[e(u, v)] \leftarrow Sim$, if $(2 \ge \lceil \epsilon \cdot \sqrt{(d[u] + 1)(d[v] + 1)} \rceil)$.

There are galloping-search based set-intersections [2, 13] and branch mis-prediction reduction approaches [12]. However, due to the irregularity of memory access, galloping-search based set intersections are unsuitable for pSCAN. Branch mis-prediction reduction approaches can not handle early terminations.

## 3.3 Other Structural Clustering Algorithms

SCAN++ [18] introduces a data structure called Directly Two-hop Away Reachable vertices (DTAR) and shares intermediate similarities within DTAR to reduce the workload. However, maintaining DTAR comes at a high cost. anySCAN [16] grows clusters from

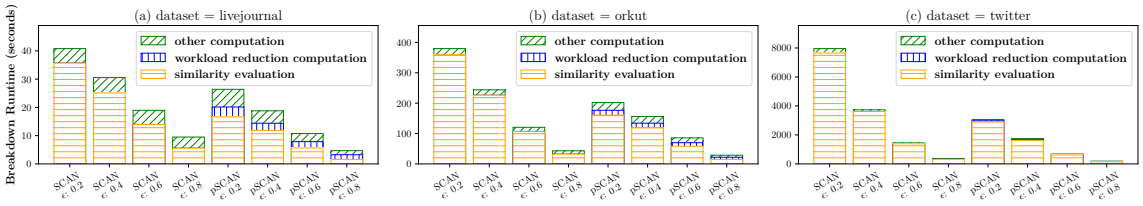



**Figure 1: Time Breakdown of SCAN and pSCAN on the twitter dataset, with $\mu = 5$**

super nodes iteratively in parallel and introduces complex vertex transitions in the expansion phase to reduce the workload. However, the transitions incur significant dynamic memory allocation overheads. SCAN-XP [19] conducts exhaustive similarity computations and exploits both thread-level and instruction-level parallelism. GS*-Index [21] constructs an index to support querying SCAN results given different parameters. However, the indexing phase involves exhaustive similarity computations, which are prohibitively expensive for massive graphs with large degree vertices. SparkSCAN [26] and PSCAN [25] are two distributed algorithms, incurring communication overheads.

Both gSkeleton [11] and SHRINK [10] are parameter-free extensions of SCAN [22]. LinkSCAN* is an extension of SCAN into link-space clustering [15]. HintClus [5], DENGRAPH [7], DHSCAN [23], AHSCAN [24] are other similarity-based hierarchical algorithms but define clusters differently from SCAN. Work on spatial DB-SCAN [8, 17, 20] is different from SCAN, since spatial data requires some indexing methods whereas in graphs, the neighborhood already provides filtering power.

**Difference.** Our work is different from previous work in that we parallelize the *pruning-based* SCAN algorithms and design the vectorized set intersection algorithm with *early termination*.

## 3.4 Analysis and Discussions

*3.4.1 Performance Bottleneck.* In Figure 1, we give the time breakdown of SCAN [22] and pSCAN [6]. We use the twitter dataset ($|V| = 41.6M$ and $|E| = 684.5M$) and set the parameter $\mu = 5$ as in pSCAN [6]. We make two observations.

• The **similarity computation** is the performance bottleneck. In SCAN [22], the total workload of similarity computations is $2\sum_{v \in V} d[v]^2$. In pSCAN, even though the number of similarity computation $CompSim(u, v)$ is reduced, in a representative case ($\epsilon = 0.2$ and $\mu = 5$), similarity computations are still time-consuming.

• The **workload reduction computation** of pSCAN is lightweight and useful, which motivates our parallelization approach.

*3.4.2 Challenges in Parallelization.* Challenges of exploring task-parallelism of pSCAN are incurred by the dependencies, and concurrency issues. Further, skews in node degrees in real-world networks pose challenges to the task scheduling. In addition, the early termination conditions of similarity computations make it difficult to utilize data-parallelism.

• **Order and Data Dependency.** 1) During the core checking and clustering, the non-increasing $ed[u]$ vertex iteration order requires synchronization. 2) Core checking and clustering of neighboring vertices involve concurrent access of $sd[u]$, $sd[v]$, $ed[u]$ and $ed[v]$. Specifically, both $sd[u]$ and $ed[u]$ can be modified by $u$ and its neighbors, which incurs write-write conflict. 3) Owing to the similarity value reuse technique, similarity values $sim[e(u, v)]$ and $sim[e(v, u)]$ are dependent. The dependencies lead to possible *redundant computation*: i) concurrent computation of $sim[e(u, v)]$ and $sim[e(v, u)]$; ii) concurrent union operations of $e(u, v)$ and $e(v, u)$, namely $Union(u, v)$ and $Union(v, u)$.

• **Clustering Concurrency Issues.** Three operations for the clustering are required to be thread safe in the concurrent execution. They are: 1) union-find operations $IsSameSet(u, v)$ and $Union(u, v)$ for the core clustering, 2) initializing a cluster id from the union-find-set data structure, and 3) assigning cluster id to non-cores.

• **Workload Skew and Irregularity.** The workload of core checking and clustering of a vertex $u$'s depends on $d[u]$, and skews in node degrees occur in real-world networks. Thus, vertex computations on core checking and clustering are skewed. Also, pruning techniques of pSCAN make the workload irregular, which poses challenges on the task scheduling.

• **Similarity Computation Vectorization.** The early termination technique for the $CompSim(u, v)$ requires to record number of matches and mismatches during the set intersection, which makes it difficult to parallelize via vectorized instructions.

# 4 PARALLELIZATION

To decouple the dependencies of pSCAN [6], we separate the core checking and clustering into two big steps as follows: 1) **role computing** (core checking and consolidating) finalizes the roles of all vertices; and 2) **core and non-core clustering** produces final clusters. In addition, inspired by the similarity predicate pruning technique, we add a pre-processing phase, during which some similarity values and roles are determined without set intersections.

This section is organized as follows. First, we give an overview of how we tackle parallelization challenges. Subsequently, we show the two big steps, namely the role computing (core checking and consolidating), and the core and non-core clustering. At the end, we discuss the degree-based dynamic task scheduling.

## 4.1 Overview

We tackle parallelization challenges of pSCAN [6] as follows.

• **sd, ed Dependency Decoupling**. 1) We remove the $ed[u]$ based max priority queue, since it incurs heavy synchronization. 2) We replace $sd$, $ed$ arrays with local variables for each vertex to eliminate data races. Local $sd$, $ed$ for a vertex $u$ can be initialized from similarity values related to $u$ at a small cost. For removing the $ed[u]$ based priority queue, we show its effect experimentally on the workload reduction is negligible .

• **Vertex Order Constraints.** We add a constraint $u < v$ in the core checking and clustering to guarantee that each undirected edge $(u, v)$ is computed at most once for the similarity value and used at most once for the core clustering.

**Algorithm 3:** ppSCAN Role Computing

```
   Input: a graph G = (V, E), parameters 0 < ϵ ≤ 1 and μ ≥ 1
   Output: roles role
1  foreach u ∈ V in parallel do
2  |  PruneSim(u)
3  foreach u ∈ V and role[u] == Unknown in parallel do
4  |  CheckCore(u)
5  foreach u ∈ V and role[u] == Unknown in parallel do
6  |  ConsolidateCore(u)
7  Procedure PruneSim(u)
8  |  foreach v ∈ N(u) do
9  |  |  sim[e(u, v)] ← Unkown
10 |  |  Update sim[e(u, v)] using the similarity predicate pruning
11 |  |  if sim[e(u, v)] == Sim then
12 |  |  |  sd ← sd + 1
13 |  |  else if sim[e(u, v)] == NSim then
14 |  |  |  ed ← ed − 1
15 |  if sd ≥ μ then
16 |  |  role[u] ← Core
17 |  else if ed < μ then
18 |  |  role[u] ← NonCore
19 |  else
20 |  |  role[u] ← Unknown
21 Procedure CheckCore(u)
22 |  foreach v ∈ N(u) do
23 |  |  if sim[e(u, v)] == Sim then
24 |  |  |  sd ← sd + 1
25 |  |  |  if sd ≥ μ then
26 |  |  |  |  role[u] ← Core, return
27 |  |  else if sim[e(u, v)] == NSim then
28 |  |  |  ed ← ed − 1
29 |  |  |  if ed < μ then
30 |  |  |  |  role[u] ← NonCore, return
31 |  foreach v ∈ N(u) and u < v and sim[e(u, v)] == Unknown do
32 |  |  sim[e(v, u)] ← sim[e(u, v)] ← CompSim(u, v)
33 |  |  Update sd, ed and role in the same logic as Lines 23-30
34 Procedure ConsolidateCore(u)
35 |  Do the same as CheckCore(u), except for removing the constraint
   |  u<v in Line 31
```

• **Thread-Safe Clustering.** 1) We adopt wait-free union-find implementations [1] for the core clustering operations. 2) We adopt compare-and-swap operations for the cluster id initialization. 3) We adopt a pipelined design in the non-core clustering by overlapping the computation of local non-core id and cluster id pairs and the copying back to a global pair array.

• **Multi-Phase Computations.** We further decompose big steps into phases for applying pruning techniques and avoiding workload redundancy coming from the concurrency. Barriers are introduced between phases. 1) To apply similarity value reuse and min-max pruning techniques, we separate the core checking into two phases: the first does similarity computations only when $u < v$ and the second consolidates the roles of all vertices. 2) To fully make use of the union-find pruning technique, we separate the core clustering into two phases: the first clusters cores without set intersections whereas the second produces final clusters of cores with set intersections.

• **Task Scheduling.** We bundle a set of vertex computations into a task and dynamically submit them into a thread pool, according to the degrees and current roles of vertices in the task.

## 4.2 Role Computing

The role computing step (Algorithm 3), which consolidates the roles of all vertices, is separated into three phases: similarity pruning, core checking and consolidating as follows. 1) The **similarity pruning** (Lines 7-20) applies the similarity predicate pruning technique to determine some similarities without set intersections, at the end of which the roles are initialized according to the known similarities. 2) The **core checking** (Lines 21-33) applies the min-max pruning technique to check the $role[u]$. In order to achieve the similarity reuse and avoid concurrent execution redundancy, the vertex order constraint $u < v$ is introduced. However, due to the $u < v$ constraint in the core checking, some roles may not be known. 3) The **core consolidating** (Lines 34-35) is for consolidating these unknown roles, using the same logic as the core checking except it removes the constraint $u < v$.

*4.2.1 Similarity Pruning.* Recall, for each vertex $u$, we introduce local variables $sd$, $ed$ to work as lower and upper bounds of $|N_\epsilon(u)| - 1$. The motivation behind introducing the similarity pruning phase is to help initialize some similarities that can be known without set intersections. At the core checking of $u$, $sd$ and $ed$ can be initialized from these known similarities. Thus, we do as few similarity computations as possible to meet the min-max pruning termination condition ($sd \geq \mu$ or $ed < \mu$), which yields better pruning. In the end, some $role[u]$ is updated as *Core* or *NonCore*, which helps avoid entering into the $CheckCore(u)$ to save iterations through similarity values (Lines 22-30).

*4.2.2 Core Checking and Consolidating.* In the core checking, $u$'s neighbor $v$ satisfying $v < u$ may update the similarity value $sim[e(u, v)]$ which is read by $u$ for the $sd$, $ed$ initialization. Even though it incurs a read-write conflict, the logic of $CheckCore(u)$ is still correct. This is because $u$ does not compute similarities $sim[e(u, v)]$ satisfying $v < u$, which guarantees $sd$ and $ed$ are updated correctly. Recall, the $u < v$ constraint in the core checking helps to remove the similarity reuse technique incurs concurrent redundancy without the order. However, the $u < v$ constraints also means some roles may be unknown if the termination condition is not met after the incomplete iteration (Lines 31-33). Thus, the core consolidating phase is introduced for correctness.

Theorem 4.1. *The similarity computation is at most invoked once for the similarity values $sim[e(u, v)]$ and $sim[e(v, u)]$.*

Proof. The similarity computation only occurs in the core checking and consolidating. 1) We show computations do not repeatedly occur in both phases. If $CompSim(u, v)$ is invoked in the core checking, then because of the assignment $sim[e(v, u)] \leftarrow sim[e(u, v)]$ and the barrier between phases, core consolidating will not involve any similarity computation. 2) We show the execution does not occur concurrently in each phase. During the core checking, this holds because of the constraint $u < v$. For the core consolidating, we show the argument holds by contradiction. The concurrent similarity computations imply roles of $u$ and $v$ (without the loss of generality, we assume $u < v$) are both *Unkown* before the consolidating, then $sim[e(u, v)]$ must be known for the core checking of $u$ (see Line 31), which contradicts the need to compute it. □

Theorem 4.2. *Roles of all the vertices are correct and complete after the core checking and consolidating.*

Proof. Access of $sim[e(u, v)]$ does not incur duplicated $sd$ or $ed$ updates, because of the three different conditions in Lines 23, 27 and 31. Thus, $sd$ and $ed$ are correctly updated, which means

**Algorithm 4:** ppSCAN Core and Non-Core Clustering

```
   Input: a graph G = (V, E), parameters 0 < ϵ ≤ 1 and μ ≥ 1, roles role
   Output: clusters
1  foreach u ∈ V and role[u] == Core in parallel do
2  |  ClusterCoreWithoutCompSim(u)
3  foreach u ∈ V and role[u] == Core in parallel do
4  |  ClusterCoreWithCompSim(u)
5  foreach u ∈ V and role[u] == Core in parallel do
6  |  InitClusterId(u)
7  foreach u ∈ V and role[u] == Core in parallel do
8  |  ClusterNonCore(u)
9  Procedure ClusterCoreWithoutCompSim(u)
10 |  foreach v ∈ N(u) and role[v] == Core and u < v and
   |    not IsSameSet(u, v) and sim[e(u, v)] == Sim do
11 |  |  Union(u, v)
12 Procedure ClusterCoreWithCompSim(u)
13 |  foreach v ∈ N(u) and role[v] == Core and u < v and
   |    not IsSameSet(u, v) and sim[e(u, v)] == Unknown do
14 |  |  sim[e(u, v)] ← CompSim(u, v)
15 |  |  if sim[e(u, v)] == Sim then
16 |  |  |  Union(u, v)
17 Procedure InitClusterId(u)
18 |  ru ← FindRoot(u)
19 |  do
20 |  |  min_core_id ← cluster_id[ru]
21 |  |  if u ≥ min_core_id then
22 |  |  |  break
23 |  while not CAS(&cluster_id[ru], min_core_id, u)
24 Procedure ClusterNonCore(u)
25 |  foreach v ∈ N(u) and role[v] == NonCore do
26 |  |  if sim[e(u, v)] == Unknown then
27 |  |  |  sim[e(u, v)] ← CompSim(u, v)
28 |  |  if sim[e(u, v)] == Sim then
29 |  |  |  Assign cluster_id[FindRoot(u)] to the NonCore v
```

$role[u]$ is correct. Due to the core consolidating phase, roles of all the vertices will be known. Thus, the roles are complete. □

## 4.3 Core and Non-Core Clustering

The core and non-core clustering step (Algorithm 4), to produce the final clusters, is separated into four phases: core clustering with and without similarity computations, cluster id initialization and non-core clustering as follows. 1) The two-phase **core clustering** with and without similarity computation (Lines 9-16) produces clusters of cores. 2) The **cluster id initialization** (Lines 17-23) initializes cluster id for each union-find-set, which uses atomic operations. 3) The **non-core clustering** (Lines 24-29) assigns cores' cluster id to the similar non-core neighbors to produce the final clusters.

*4.3.1 Core Clustering.* The core clustering is separated into two phases for better union-find pruning. In each phase, we add a $u < v$ constraint to avoid redundant clustering. In the first phase, similarity computations are not involved. After that, some small clusters of cores are formed among cores, which can be used for the union-find pruning in the next phase. In the second phase, similarity computations are conducted to get the complete clusters of cores. During the core clustering, many cores may occur in the same cluster, thus the condition *not* $IsSameSet(u, v)$ can help reduce similarity computations for unknown edges among these cores.

*4.3.2 Non-Core Clustering.* Before the non-core clustering stage, we initialize the cluster id of union-find-sets, which involves atomic operations over all the cores. After this phase, for each core vertex $u$, $cluster\_id[FindRoot(u)]$ is $u$'s cluster id. In the non-core clustering phase, we assign the cluster id from cores to its similar neighbors.

**Algorithm 5:** Dynamic Task Scheduling for $CheckCore(u)$

```
1  InitThreadPool(), deg_sum ← 0, beg ← 0
2  for u ← 0; u < |V|; u ← u + 1 do
3  |  if role[u] == Unknown then
4  |  |  deg_sum ← deg_sum + d[u]
5  |  |  if deg_sum > 32768 then
6  |  |  |  SubmitTaskToPool(Task(beg, u + 1))
7  |  |  |  deg_sum ← 0, beg ← u + 1
8  SubmitTaskToPool(Task(next_beg, |V|)), JoinThreadPool()
9  Procedure Task(beg, end)
10 |  for u ← beg; u < end; u ← u + 1 do
11 |  |  if role[u] == Unknown then
12 |  |  |  CheckCore(u)
```

Some similarity computations among cores and non-cores occur in the non-core clustering. We adopt a pipelined design in the non-core clustering by overlapping the local non-core id and cluster id pairs' computing and the copying back to a global pair array.

*Definition 4.3.* The **similar edge** is defined as an edge $(u, v) \in E$ that satisfies $sim[e(u, v)] == Sim$.

Theorem 4.4. *Each similar edge is either used for the core clustering at most once, or for the non-core clustering exactly once.*

Proof. Due to the constraint $u < v$ (Lines 10, 13), the cluster union of $u$ and $v$ occurs once when they are not yet in the same set. Owning to the logic of non-core clustering being that cores assign a cluster id to its similar neighboring non-cores, the clustering occurs exactly once from the core $u$ to the non-core $v$. □

Theorem 4.5. *Clusters are correct and complete after the core and non-core clustering.*

Proof. After the role computing, all the roles are known (Theorem 4.2). Besides, the core and non-core clustering strictly follows the definition of direct structural reachability, which means the clustering of $u$ and $v$ occurs only when $u$ is a core and $sim[e(u, v)] == Sim$. Thus, the clustering is correct. In the core clustering, all similar edges where it is possible to force cluster union are explored. In the non-core clustering, all similar edges among cores and non-cores are explored. Thus, the final clusters are complete. □

## 4.4 Degree-Based Dynamic Task Scheduling

To achieve both load balance and small overhead, we use dynamic estimation of a workload for a single task. We use accumulated degree sum of vertices requiring computations to estimate workloads, because each vertex computation depends on its role and involves computations on its neighbors.

We illustrate our task scheduling for the core checking phase (Algorithm 5) as an example. The scheduling logic also applies to other vertex computations. A task is represented by a vertex range pair $v\_beg$ and $v\_end$. We wrap the entire task execution flow as a procedure $Task(v\_beg, v\_end)$. A worker thread iterates through vertices in the range $[v\_beg, v\_end)$ and checks whether the corresponding vertex computation is required. If it is required ($role[u] == Unknown$), then $CheckCore(u)$ is invoked.

The master thread is responsible for constructing and submitting tasks to the worker thread pool. Initially, the beginning vertex id for the next task $v\_next\_beg$ and degree sum $deg\_sum$ are both initialized to 0. The master thread iterates through all the vertices and

accumulates the degree sum when a vertex $v$ requires computations ($role[v] == Unknown$). When the degree sum is above the threshold 32768 (tuned for our experimental setting), a task is submitted. We tune the parameter by multiplying the threshold (originally 1) by 2 until the workload is not balanced or the task queue maintaining cost is negligible. Correspondingly, *deg_sum* is reset to 0 and *v_next_beg* becomes $v + 1$. At last, the master thread submits the remaining task for the completeness and uses *JoinThreadPool*() to provide a barrier for synchronization purposes.

The degree-based scheduling has a couple of advantages. Firstly, worker threads will access adjacent memory locations of the edge array *dst* or the edge property arrays *sim*. Secondly, degree-based workload estimation introduces little overhead, since we only access degree array *d* and conduct addition operations on *deg_sum*.

## 5 SET-INTERSECTION VECTORIZATION

We propose the vectorized pivot-based set intersection (Algorithm 6). We explore the data parallelism in finding the next element in an array satisfying ≥ the pivot and keep the early termination optimization via the intersection count bounds (Definition 3.9). Recall, $du$ and $dv$ are upper bounds of $|\Gamma(u) \cap \Gamma(v)|$ and $cn$ is the lower bound (initially, $du = d[u] + 2$, $dv = d[v] + 2$, $cn = 2$).

**Pivot Idea.** There are three steps to be repeated until the termination condition satisfies. 1) We use $dst[off\_v]$ as the pivot, and find the first element in $dst[off\_u : off[u+1])$ satisfying ≥ the current pivot $dst[off\_v]$. 2) We use $dst[off\_u]$ as the pivot, and find the first element in $dst[off\_v : off[v+1])$ satisfying ≥ the pivot $dst[off\_u]$. 3) We update $cn$, $off\_u$, $off\_v$ when we find a match ($dst[off\_u] == dst[off\_v]$). In addition, to handle the case that the number of remaining elements in an array is smaller than 16 (the size of a vector register), we fall back to a pivot-based *CompSim* without vectorization to get the correct results in all cases. The logic is similar to the vectorized one, so we skip it.

**Early Termination Idea.** Intersection bounds $du$, $dv$ and $cn$ are updated respectively at the first, second and third steps. The early termination condition is checked right after the intersection bound is updated. There are three cases as follows. 1) When the upper bound $du$ gets decremented, we check whether $du < c$ satisfactory to see if we can return *NSim*. 2) The first case logic also applies to $dv$. 3) When the lower bound $cn$ gets incremented, $cn \geq c$ is checked to see if we can return *Sim*. The reasons we can apply the early termination idea to our pivot-based set intersection are as follows. 1) $cn$ is only updated in checking the match (step 3). 2) While finding the next pivot via vectorized comparisons, we can know exactly the number of elements satisfying < *pivot*, which helps correctly update $du$ or $dv$.

We use the step 1 (Lines 4-15) to illustrate the logic of finding the next pivot's offset $off\_u$, based on the current pivot $dst[off\_v]$. There is a while-loop, with the condition of whether there are ≥ 16 $u$'s neighbors yet to check. The condition is to handle cases where the remaining element size is less than 16. In the while loop, we try to find the first $off\_u$ satisfying $dst[off\_u] \geq$ the pivot $dst[off\_v]$. Once we find the offset of the next pivot, we break.

The details of step 1 loop body are as follows. Firstly, we load 16 identical integers $dst[off\_v]$ into a 512-bit vector register *pivot_v* and load 16 integers $dst[off\_u : off\_u + 16)$ into another vector register *u_eles*. Secondly, we conduct a bit-wise operation (>) to compare each element in *pivot_v* to the corresponding one in *u_eles* and store the comparison results into a 16-bit variable *mask*. If the element in *pivot_v* is greater than that in *u_eles*, the corresponding mask bit is set to 1, otherwise 0. After we count the number of 1-bits in *mask*, we know how many elements in *u_els* are less than $dst[off\_v]$ and store this value into *bit_cnt*. Thirdly, when we know *bit_cnt*, we advance $off\_u$ and $du$, and check if $du < c$ satisfies for the early termination. Fourthly, we check if we have already found the first element in $u$'s neighbors greater than or equal to the pivot $dst[off\_v]$. If we find it, we can break.

There are a couple of advantages using pivot-based vectorized instructions to find the next element in an array satisfying ≥ the pivot. Firstly, $du$ and $dv$ are updated less frequently than that in a sequential implementation, since they decrease by *bit_cnt* rather than 1. Secondly, condition comparisons ($dst[off\_u] < dst[off\_v]$ and $dst[off\_u] > dst[off\_v]$) are replaced with vectorized comparisons, alleviating the problem of branch mis-predictions.

**Algorithm 6:** Vectorized Pivot-based *CompSim*$(u, v)$

**Input:** $u$, $v$, $u$'s and $v$'s neighbors (sorted arrays)
**Output:** similarity value of $e(u, v)$

1 $c \leftarrow \sqrt{(d[u]+1)(d[v]+1)}$, $du \leftarrow d[u]+2$, $dv \leftarrow d[u]+2$, $cn \leftarrow 2$
2 $off\_u \leftarrow off[u]$, $off\_v \leftarrow off[v]$
3 **while** ***true*** **do**
  /* Step1: find the next pivot offset $off\_u$ */
4  **while** $off\_u + 16 < off[u+1]$ **do**
    /* Load 16 identical integers */
5    $pivot\_v \leftarrow \_mm512\_set1\_epi32(dst[off\_v])$
    /* Load 16 integers */
6    $u\_eles \leftarrow \_mm512\_loadu\_si512(\&dst[off\_u])$
    /* Mask bit is 1 if $pivot\_v > u\_ele$, 0 otherwise */
7    $mask \leftarrow \_mm512\_cmpgt\_epi32\_mask(pivot\_v, u\_eles)$
    /* Number of elements < $pivot\_v$ */
8    $bit\_cnt \leftarrow \_mm\_popcnt\_u32(mask)$
9    $off\_u \leftarrow off\_u + bit\_cnt$, $du \leftarrow du - bit\_cnt$
10    **if** $du < c$ **then**
11      **return** $NSim$
    /* If not all $u\_ele < pivot\_v$, we find the $off\_u$ */
12    **if** $bit\_cnt < 16$ **then**
13      **break**
14  **if** $off\_u + 16 \geq off[u+1]$ **then**
15    **break**
  /* Step2: find the next pivot offset $off\_v$ */
16  Find the next $off\_v$, satisfying $dst[off\_v] \geq pivot\_u$ **using the same logic as Lines 4-13**
17  **if** $off\_v + 16 \geq off[v+1]$ **then**
18    **break**
  /* Step3: if find a match, we update $cn$, $off\_u$, $off\_v$ */
19  **if** $dst[off\_u] == dst[off\_v]$ **then**
20    $cn \leftarrow cn + 1$, $off\_u \leftarrow off\_u + 1$, $off\_v \leftarrow off\_v + 1$
21    **if** $cn \geq c$ **then**
22      **return** $Sim$
23 Fall back to the non-vectorized logic to finish the remaining work

## 6 EVALUATION

We compare **ppSCAN** with the sequential **SCAN** and **pSCAN** [6, 22], and the parallel **anySCAN** [16], **SCAN-XP** [19] and our **ppSCAN-NO** (without the set-intersection optimization). All the algorithms are implemented in C++, compiled with -O3 option.

We use two Linux servers (CPU and KNL servers, both with hyper-threading, supporting AVX2 and AVX512 respectively). The CPU server has two 10-core 2.3GHz Intel Xeon E5-2650 CPUs (in

**Table 1: Real-world Graph Statistics**

| Name | $\|V\|$ | $\|E\|$ | $\overline{d}$ | max $d$ |
|---|---|---|---|---|
| orkut | 3, 072, 627 | 117, 185, 083 | 76.3 | 33, 312 |
| webbase | 118, 142, 143 | 525, 013, 368 | 8.9 | 803, 138 |
| twitter | 41, 652, 230 | 684, 500, 375 | 32.9 | 1, 405, 985 |
| friendster | 124, 836, 180 | 1, 806, 067, 135 | 28.9 | 5, 214 |

**Table 2: Synthetic Graph Statistics**

| Name | $\|V\|$ | $\|E\|$ | $\overline{d}$ | max d |
|---|---|---|---|---|
| ROLL-d40 | 50, 000, 000 | 999, 999, 600 | 40 | 54, 953 |
| ROLL-d80 | 25, 000, 000 | 999, 998, 400 | 80 | 52, 074 |
| ROLL-d120 | 16, 700, 000 | 1, 001, 996, 400 | 120 | 52, 472 |
| ROLL-d160 | 12, 500, 000 | 999, 993, 600 | 160 | 49, 296 |

total 40 threads). The L1, L2, L3 cache and DRAM of the CPU server are 64KB, 256KB, 25MB, 64GB respectively. The KNL server has a 64-core 1.3GHz Intel Xeon Phi 7210 CPU (configured in quadrant mode, in total 256 threads). The L1, L2 cache, MCDRAM (configured in cache mode) and RAM of the KNL server are 64KB, 1024KB, 16GB, 96GB respectively.

We use four representative real-world graphs (Table 1) from SNAP [14] and WebGraph [3, 4]. We use synthetic ROLL graphs [9] with 1 billion edges and various average degrees (Table 2). By default, we fix $\mu = 5$ and vary $\epsilon \in [0.1, 0.9]$, since the results under different $\mu$ values show similar trends. To support this observation, we include a performance study with $\mu$ varied in $\{2, 5, 10, 15\}$. For parallel algorithms, by default, we use 64 and 256 threads for the CPU and KNL servers respectively. In the scalability experiment, we vary the number of threads in $\{1, 2, 4, 8, 16, 32, 64, 128, 256\}$.

## 6.1 Overall Performance

We compare ppSCAN with SCAN [22], pSCAN [6], anySCAN [16] and SCAN-XP [19]. Both SCAN and pSCAN are sequential algorithms, whereas ppSCAN, anySCAN and SCAN-XP are parallel. We fix $\mu = 5$ and vary the datasets and $\epsilon$. We measure the in-memory processing time for each algorithm. We repeat each execution three times and report the best run for each algorithm since the time variance among runs is small. If an algorithm incurs a runtime error (RE) or time-limit-exceeded (TLE, with time limit 90 minutes), we stop the execution. We show results on both CPU and KNL servers (Figures 2 and 3). anySCAN incurs runtime errors in the webbase and friendster datasets due to running out of memory. Both SCAN and pSCAN incur TLE on the KNL server. In all cases, SCAN is slower than pSCAN.

In most cases, ppSCAN is 26x-51x faster than pSCAN on the CPU and 98x-442x faster on the KNL. In cases such as the twitter dataset with $\epsilon = 0.8$, the memory access bounds the speedup of ppSCAN to tens. However, such cases are acceptable, since the in-memory processing of ppSCAN can be done within a few seconds.

On the orkut dataset, ppSCAN is 6x-8x faster than anySCAN on the CPU and 5x-9x faster on the KNL. On the twitter dataset, ppSCAN is 8x-34x faster on the CPU and 17x-43x faster on the KNL. The speedups are a result of ppSCAN better exploring memory access patterns and incurring negligible overhead for task scheduling. The scalability of ppSCAN is also better than anySCAN.

ppSCAN does less work than SCAN-XP: SCAN-XP conducts exhaustive computations regardless of the $\epsilon$ value, whereas the workload of ppSCAN decreases when $\epsilon$ increases. Thus, ppSCAN is faster than SCAN-XP in all cases. On the twitter dataset, due to the lack of pruning, SCAN-XP is 47x-204x slower on the CPU and 47x-125x slower on the KNL than our ppSCAN.

## 6.2 Set-Intersection Performance

*6.2.1 Invocation Reduction.* We compare the normalized number of set-intersection invocations (number of invocations divided by $|E|$) between pSCAN and ppSCAN. We fix $\mu = 5$ and vary datasets and $\epsilon$. Experimental results (Figure 4) show that ppSCAN and pSCAN conduct a similar amount of work. When we vary $\mu$, results show similar trends. Due to the limited space, we omit the results with $\mu$ varied.

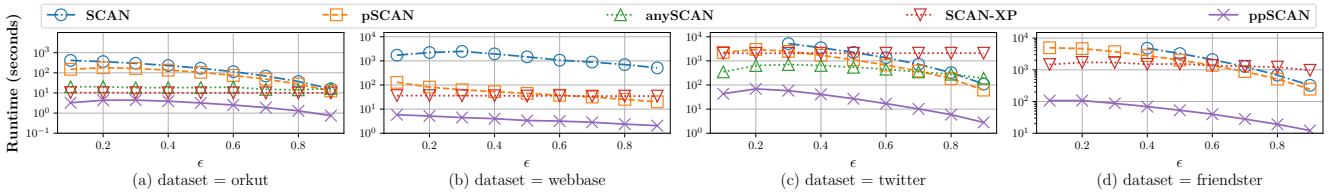
*6.2.2 Vectorization Improvement.* We compare the core checking time between ppSCAN and ppSCAN-NO (without vectorization), since core checking involves the majority of set intersections. We fix $\mu = 5$ and vary the datasets and $\epsilon$. We show the speedups of core checking with our pivot-based vectorized set-intersection on both the CPU and the KNL servers (Figure 5). We have a couple of observations. Firstly, with $\epsilon$ increasing, the benefits of set-intersection optimization decreases. This phenomenon is because at a large $\epsilon$, we make fewer advances of offsets in an array to find the first element greater than or equal to the pivot. However, in this case, a lot of workload is pruned. Secondly, the speedup on the KNL is better because the AVX512 instructions on the KNL support operating twice the number of bits per instruction than that of the AVX2 instructions on the CPU. In summary, this optimization works well with large datasets given a small $\epsilon$, and achieves up to 4.5x and 3.5x speedup on the KNL and the CPU servers.

## 6.3 Scalability to Number of Threads

We fix $\mu = 5$ , $\epsilon = 0.2$ , vary the number of threads and show the time breakdown of ppSCAN's four stages (Figure 6). On the orkut, twitter and friendster datasets, core checking takes the most time in ppSCAN, which is about one order of magnitude more than pruning and two orders of magnitude more than core and non-core clustering. On the webbase dataset, due to the powerful pruning (Figure 4(b)), core checking is not dominant. All stages scale well on the 64-core KNL server (256 threads, hyper-threading). In most cases, the speedup of core checking is better than the other stages because the computations from set intersections hide the memory access latency. Also, since core and non-core clustering involves concurrent lock-free operations on union-find-sets, the overhead increases with the number of threads. Specifically, given 256 threads, the speedups on core checking for orkut, webbase, twitter, friendster datasets are respectively 102x, 26x (memory bound), 132x and 143x respectively, and speedups of all four stages for these four datasets are 72x, 28x, 125x and 131x respectively.

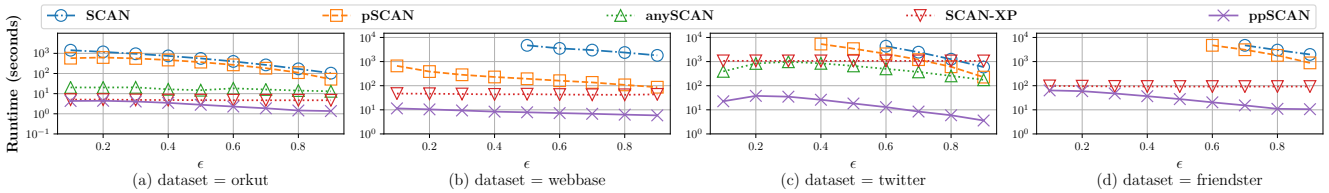
## 6.4 Robustness

*6.4.1 Varying Parameters $\mu$ and $\epsilon$ on Real-World Graphs.* We vary datasets, $\mu$ and $\epsilon$ and show ppSCAN runtime for all the cases (Figure 7). On the orkut, twitter and friendster datasets, with different $\mu$ values, the runtime shows similar trends. In the case of $\epsilon = 0.1$, runtime with $\mu = 15$ becomes a little bit more than with $\mu = 2$ due to less pruning. On the webbase dataset, the trends in the runtime with $\epsilon$ varied become slightly different with $\mu$. When $\mu$ is 2, it takes longer, because there are many cores, which increase



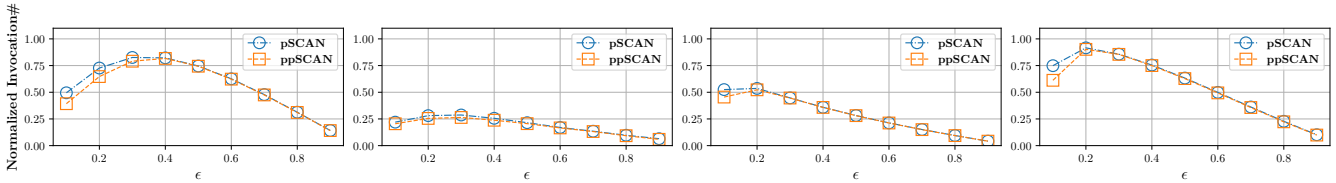

(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) dataset = friendster

**Figure 2: Comparison with existing algorithms (on CPU), $\mu = 5$**



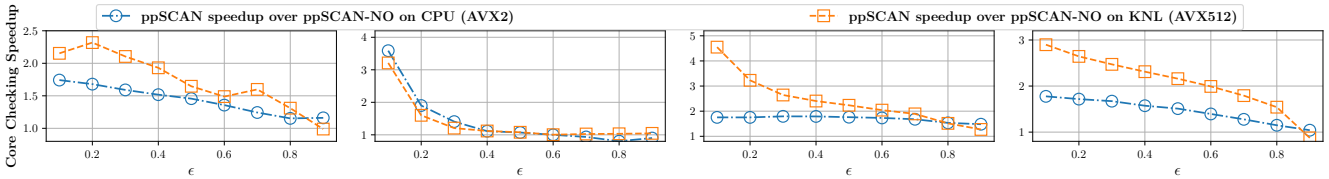

(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) dataset = friendster

**Figure 3: Comparison with existing algorithms (on KNL), $\mu = 5$**



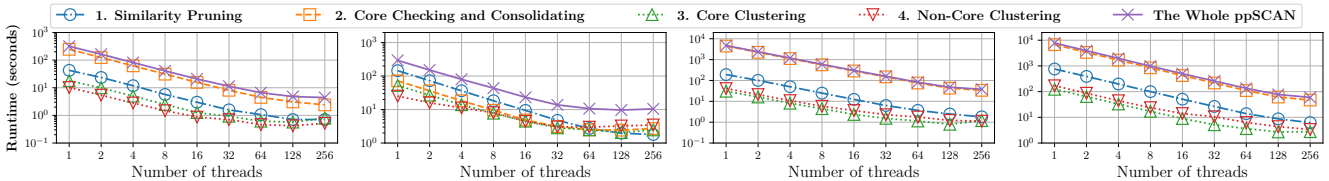

(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) datset = friendster

**Figure 4: Set-intersection invocation reduction experiment, $\mu = 5$**



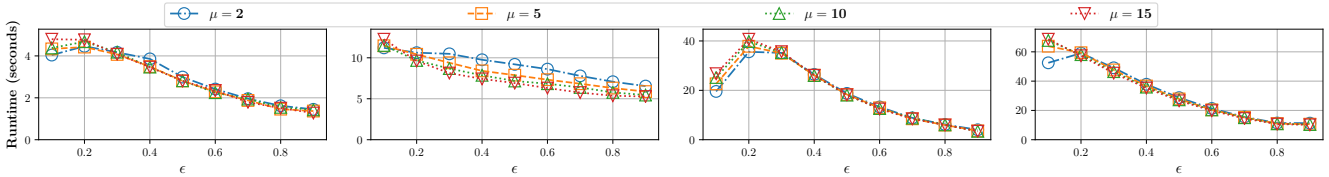

(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) dataset = friendster

**Figure 5: Set-intersection optimization experiment (on CPU and KNL), $\mu = 5$**



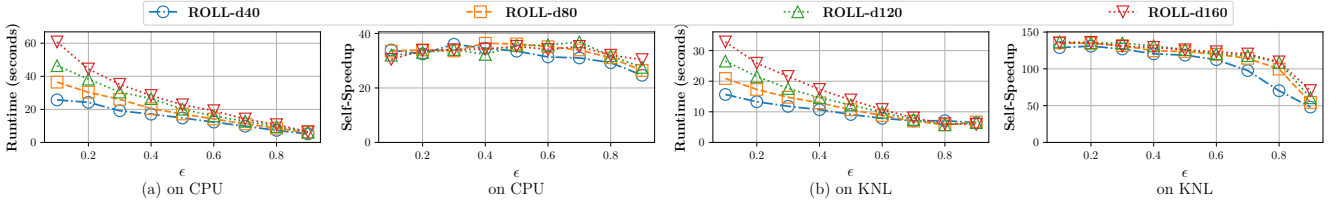

(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) dataset = friendster

**Figure 6: Scalability experiment (on KNL), $\epsilon = 0.2$ and $\mu = 5$**




(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) dataset = friendster

**Figure 7: Robustness experiment 1 (on KNL)**




(a) on CPU on CPU (b) on KNL on KNL

**Figure 8: Robustness experiment 2 on ROLL graphs (on CPU and KNL), $|E| = 1$ *billion* and $\mu = 5$**

the clustering time. In summary, ppSCAN is robust given different combinations of datasets, $\mu$ and $\epsilon$. The vertex degree based task scheduling is important for the performance robustness under different parameters since workloads and parameters influence both load balance and scheduling overheads. Our lock-free design makes it easy to achieve robust scheduling.

*6.4.2 Varying $\bar{d}$ on Synthetic ROLL Graphs.* We also evaluate the performance of ppSCAN on synthetic ROLL graphs [9]. We generate four 1 billion-edge ROLL graphs with average degrees respectively 40, 80, 120 and 160. We fix $\mu = 5$ and vary $\epsilon$ and datasets. We report both runtime and self-speedup (over ppSCAN with 1 thread) on both CPU and KNL servers (Figure 8). The runtime for graphs of larger degrees is greater than that of smaller degree graphs. However, we can finish the in-memory processing for all cases within 60 seconds on the CPU server and 35 seconds on the KNL server. With $\epsilon$ increasing, the runtime for graphs of different degrees gets close to each other. This is because the core checking takes less in the total computation. On the CPU server, we can achieve 25x-35x speedups in all cases. However, the speedups on the KNL server decrease sharply at $\epsilon = 0.8$. This is because the computations of core checking takes too little time to hide the memory access time. However, the performance is acceptable, since the runtime is already under 5 seconds.

## 7 CONCLUSION

We parallelize the state-of-the-art pruning-based graph clustering algorithm pSCAN. Our parallel algorithm ppSCAN consists of multi-phase lock-free vertex computations, which are dynamically bundled into a task and scheduled based on vertex degrees and roles. Moreover, we propose a pivot-based vectorized set intersection algorithm to optimize the performance bottleneck. Experimental results show that ppSCAN computes similar workloads to pSCAN, scales well to the number of threads and is robust given different datasets and parameters. In most cases on the KNL server, ppSCAN is two orders of magnitude faster than pSCAN and one order of magnitude faster than the parallel anySCAN and SCAN-XP.

## 8 ACKNOWLEDGEMENT

This work was partly supported by grants 16206414 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft.

## REFERENCES

[1] Richard J Anderson and Heather Woll. 1991. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM, 370–380.

[2] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. 2006. Faster adaptive set intersections for text searching. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 146–157.

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[5] Dustin Bortner and Jiawei Han. 2010. Progressive clustering of networks using structure-connected order of traversal. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 653–656.

[6] Lijun Chang, Wei Li, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. pSCAN: Fast and exact structural graph clustering. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 253–264.

[7] Tanja Falkowski, Anja Barth, and Myra Spiliopoulou. 2007. Dengraph: A density-based community detection algorithm. In *Web Intelligence, IEEE/WIC/ACM International Conference on*. IEEE, 112–115.

[8] Markus Götz, Christian Bodenstein, and Morris Riedel. 2015. HPDBSCAN: highly parallel DBSCAN. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2.

[9] Ali Hadian, Sadegh Nobari, Behrooz Minaei-Bidgoli, and Qiang Qu. 2016. ROLL: Fast in-memory generation of gigantic scale-free networks. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1829–1842.

[10] Jianbin Huang, Heli Sun, Jiawei Han, Hongbo Deng, Yizhou Sun, and Yaguang Liu. 2010. SHRINK: a structural clustering algorithm for detecting hierarchical communities in networks. In *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM, 219–228.

[11] Jianbin Huang, Heli Sun, Qinbao Song, Hongbo Deng, and Jiawei Han. 2013. Revealing density-based clustering structure from the core-connected tree of a network. *IEEE Transactions on Knowledge and Data Engineering* 25, 8 (2013), 1876–1889.

[12] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment* 8, 3 (2014), 293–304.

[13] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 46, 6 (2016), 723–749.

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

[15] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping community detection using the link-space transformation. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 292–303.

[16] Son T Mai, Martin Storgaard Dieu, Ira Assent, Jon Jacobsen, Jesper Kristensen, and Mathias Birk. 2017. Scalable and Interactive Graph Clustering Algorithm on Multicore CPUs. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 349–360.

[17] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 62.

[18] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. SCAN++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1178–1189.

[19] Tomokatsu Takahashi, Hiroaki Shiokawa, and Hiroyuki Kitagawa. 2017. SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors. In *Proceedings of the 2nd International Workshop on Network Data Analytics*. ACM, 6.

[20] Benjamin Welton, Evan Samanas, and Barton P Miller. 2013. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 84.

[21] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient Structural Graph Clustering: An Index-Based Approach. *Proceedings of the VLDB Endowment* 11, 3 (2017).

[22] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 824–833.

[23] Nurcan Yuruk, Mutlu Mete, Xiaowei Xu, and Thomas AJ Schweiger. 2007. A divisive hierarchical structural clustering algorithm for networks. In *Data Mining Workshops, 2007. ICDM Workshops 2007. Seventh IEEE International Conference on*. IEEE, 441–448.

[24] Nurcan Yuruk, Mutlu Mete, Xiaowei Xu, and Thomas AJ Schweiger. 2009. AHSCAN: Agglomerative hierarchical structural clustering algorithm for networks. In *Social Network Analysis and Mining, 2009. ASONAM'09. International Conference on Advances in*. IEEE, 72–77.

[25] Weizhong Zhao, Venkataswamy Martha, and Xiaowei Xu. 2013. PSCAN: a parallel Structural clustering algorithm for big networks in MapReduce. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 862–869.

[26] Qijun Zhou and Jingbin Wang. 2015. SparkSCAN: A Structure Similarity Clustering Algorithm on Spark. In *National Conference on Big Data Technology and Applications*. Springer, 163–177.