Parallelizing Pruning-based Graph Structural Clustering

Yulin Che, Shixuan Sun, Qiong Luo
Hong Kong University of
Science and Technology

Outline

1、Pruning-based Graph Structural Clustering

2、Performance Bottleneck & Challenge

3、Parallelization & Vectorization

4、Experimental Study

5、Conclusion

Graph Structural Clustering

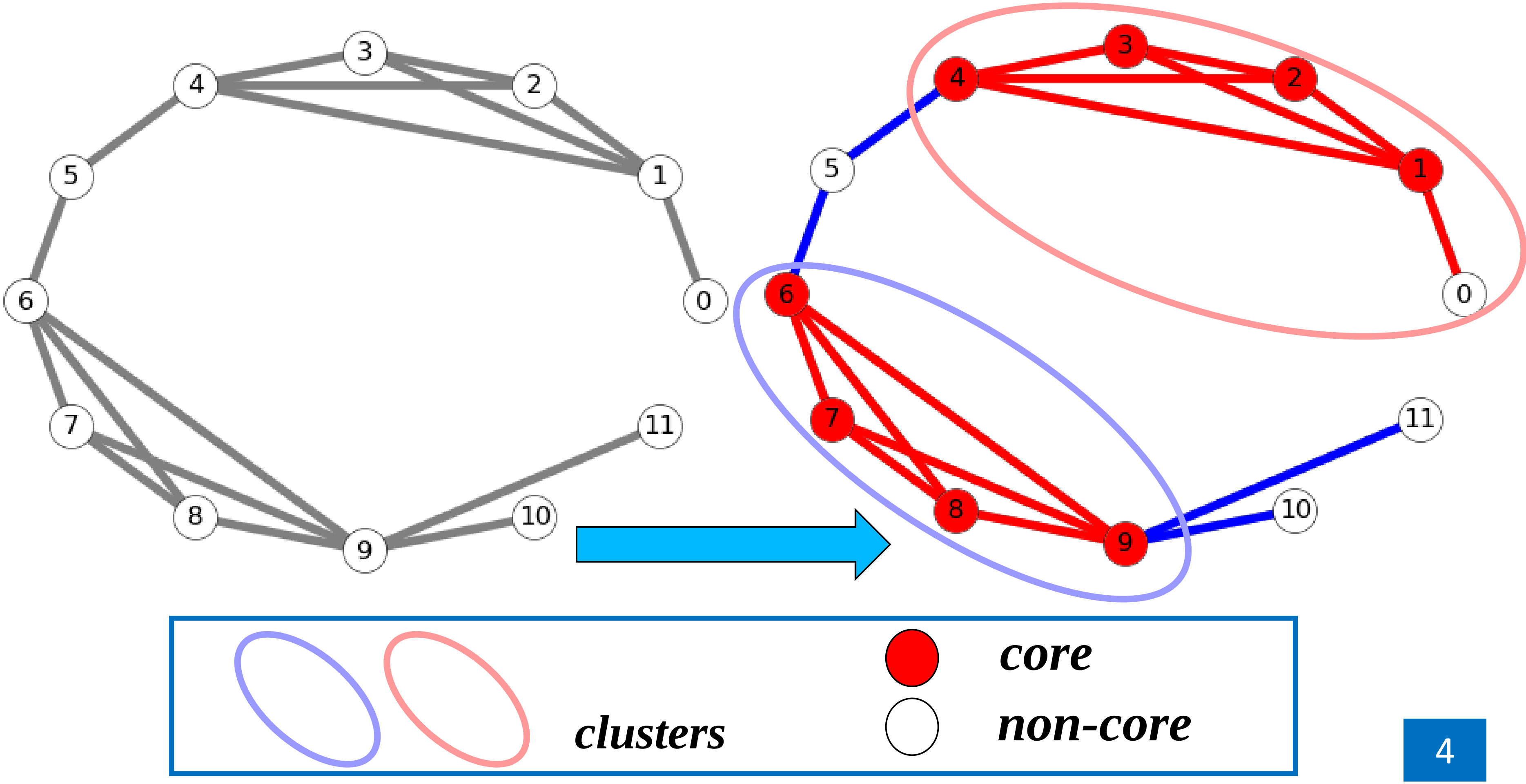
- **Graph Clustering**
  - group vertices into ***clusters***: dense intra connection and sparse inter connection

- **Application**
  - do recommendations on social networks, web graphs and co-purchasing graphs

- **Graph Structural Clustering (Our Focus)**
  - utilize ***structural similarity*** among vertices for clustering
  - identify ***clusters*** and ***vertex roles*** (cores, non-cores)

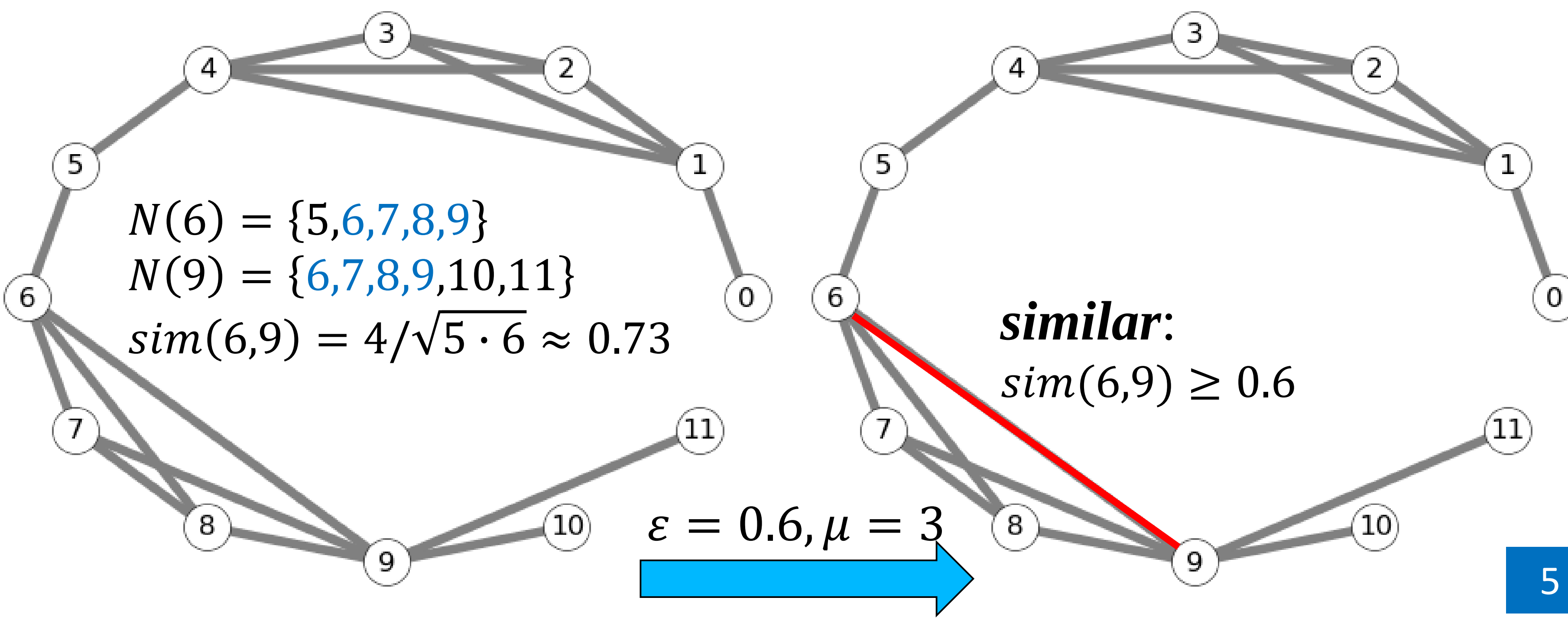
Graph Structural Clustering Example

- **Graph Structural Clustering (Our Focus)**
  - utilize ***structural similarity*** among vertices for clustering
  - identify ***clusters*** and ***vertex roles*** (cores, non-cores: hubs, outliers)



SCAN [Xu+, KDD'07]

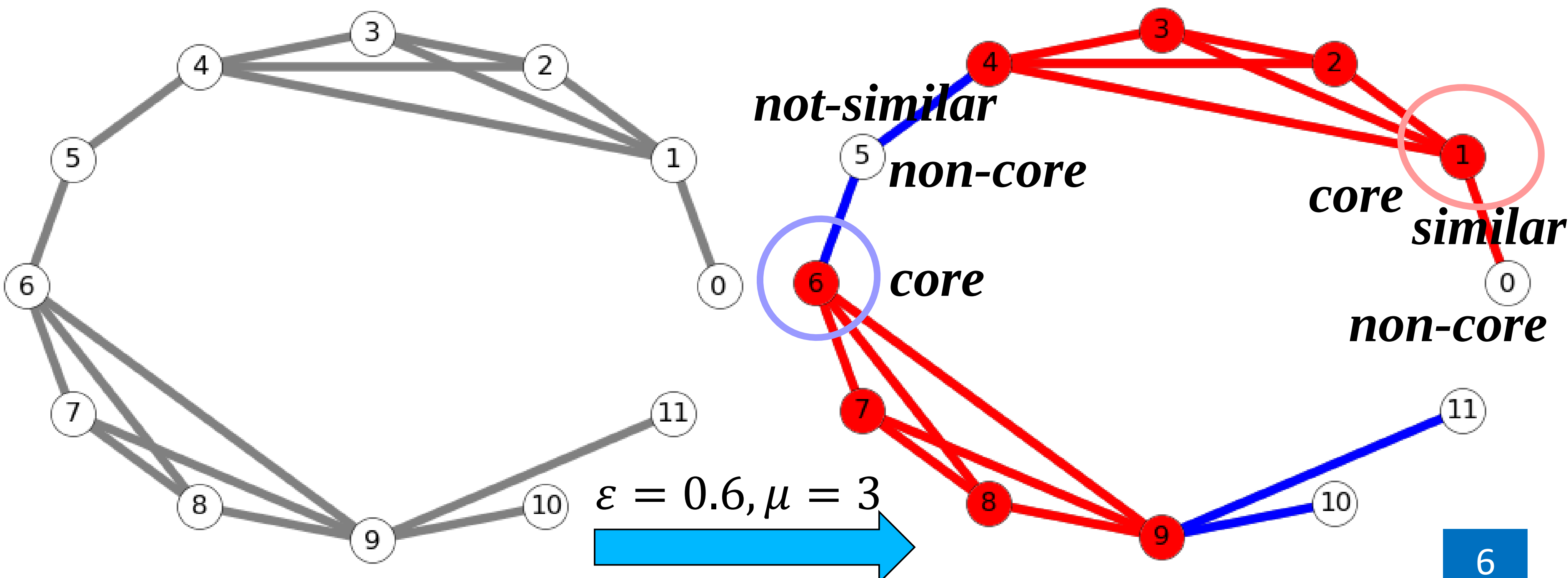
- **Structural Similarity Computation**
  - based on neighbors of two vertices u and v (cosine measure):
    - $sim(u,v) = |N(u) \cap N(v)| / \sqrt{|N(u)| \cdot |N(v)|}$
  - u and v are ***similar neighbors***, if
    - they are connected
    - their ***structural similarity*** $sim(u,v) \geq \varepsilon$



SCAN [Xu+, KDD'07]

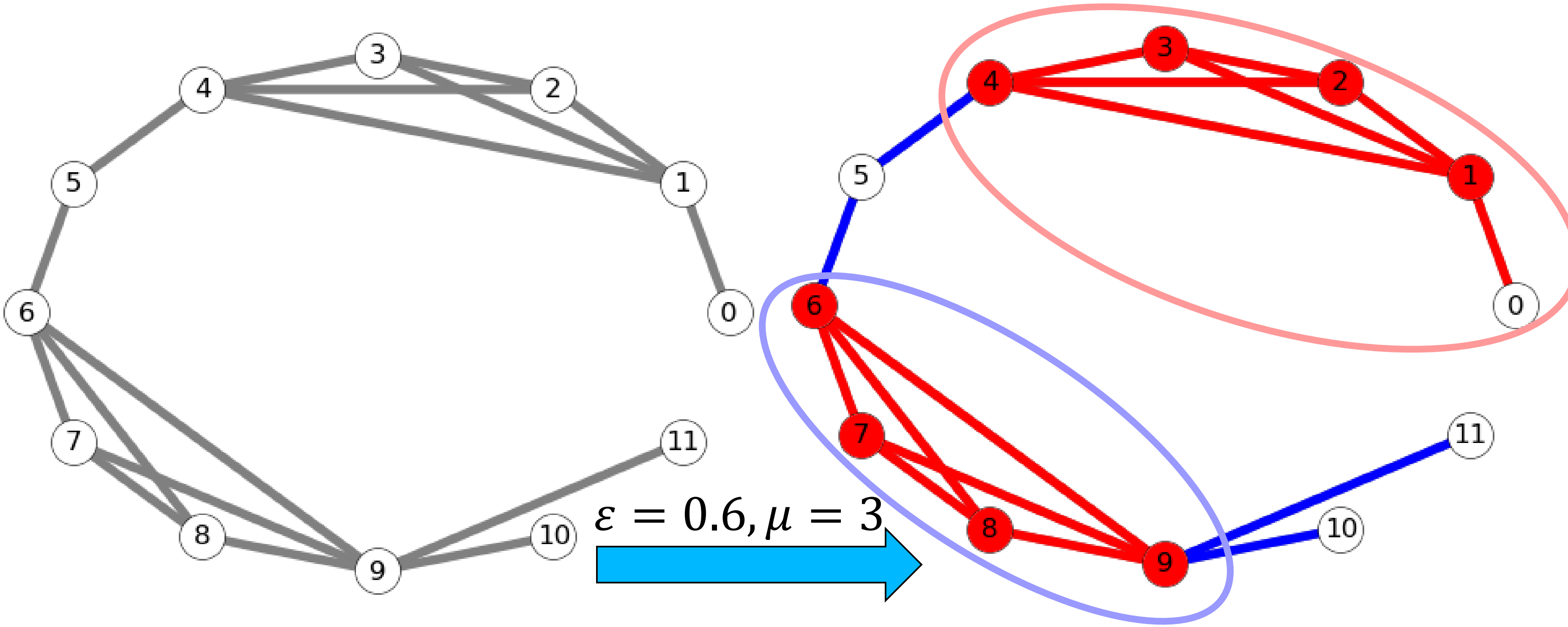
- **Core Checking**
  - a vertex u is a core, if it has $\geq \ \mu$ ***similar neighbors***

- **Structural Clustering**
  - clustering by ***similar neighbors*** from cores
  - adopting Breadth-First-Search (***BFS***) for cluster expansion



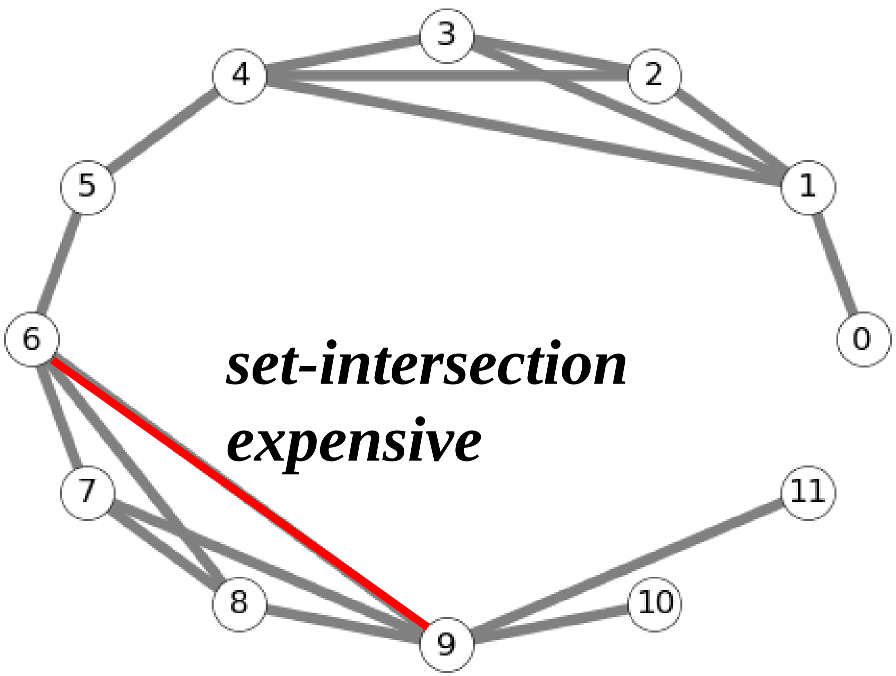
SCAN [Xu+, KDD'07]

- **Structural Clustering**
  - clustering by ***similar neighbors*** from cores
  - adopting Breadth-First-Search (***BFS***) for cluster expansion



pSCAN [Chang+, ICDE'16]

- **Pruning Similarity Computations**
  - adopt ***union-find*** data-structure, change algorithmic design
  - avoid ***redundant similarity computation***, apply pruning techniques
  - apply ***early termination*** in similarity computation

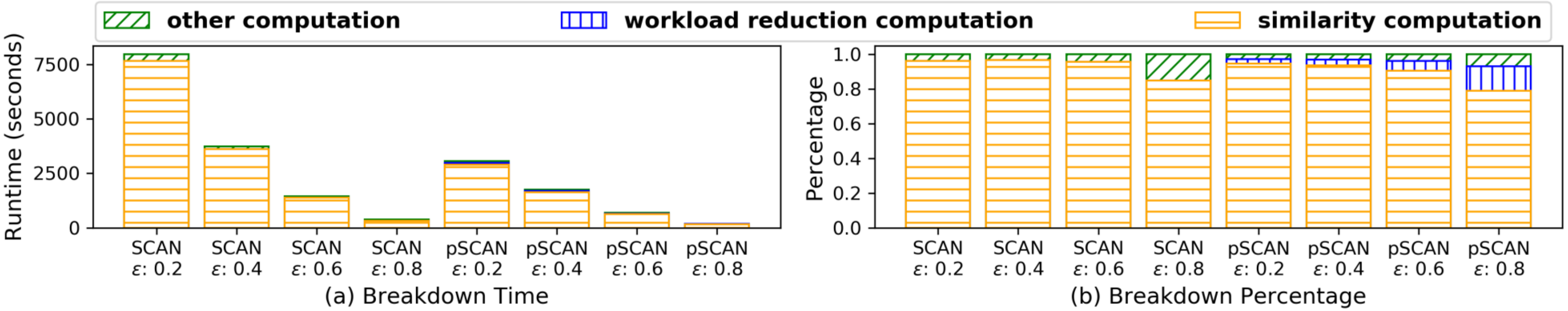


set-intersection expensive

$N(6) = \{5,6,7,8,9\}$
$N(9) = \{6,7,8,9,10,11\}$
$sim(6,9) = 4/\sqrt{5 \cdot 6} \approx 0.73$

Motivation of Parallelization

- **Motivation of Parallelizing pSCAN**
  - cost ***too much time*** for interactive exploration of clustering results
  - cost most from the ***set-intersection based similarity computation***



On Twitter (0.7-billion-edge)

Outline

1、Pruning-based Graph Structural Clustering

2、Performance Bottleneck & Challenge

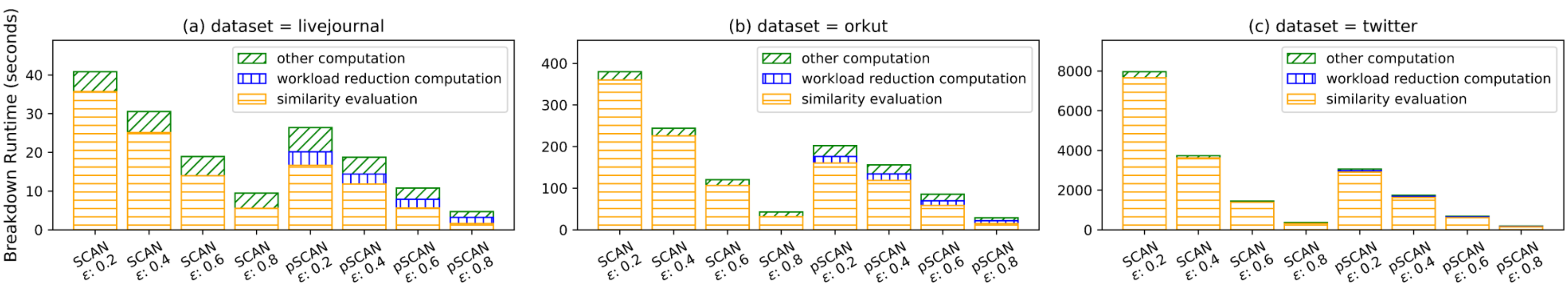
3、Parallelization & Vectorization

4、Experimental Study

5、Conclusion

Time BreakDown (SCAN and pSCAN)

- **Observations**
  - ***similarity computation*** is the performance bottleneck
  - ***workload reduction*** of pSCAN is light-weight but useful



On LiveJournal/Orkut/Twitter

pSCAN Parallelization Challenge

- **Data Dependency**
  - there exists concurrent access of ***lower and upper bounds*** of ***number of similar neighbors***
  - a priority queue for selecting vertex with max upper bound of ***similar neighbors*** requires heavy synchronization
  - owing to the ***similarity reuse*** technique, similarity values $\boldsymbol{sim(u, v)}$ and $\boldsymbol{sim(v, u)}$ are dependent

- **Clustering Concurrency Issues**
  - ***union-find*** operations should be thread-safe
  - ***cluster id initialization and assignment*** should be thread-safe

- **Workload Skew and Irregularity**
  - the workload for each vertex depends on its ***degree*** and ***role***
  - ***pruning techniques*** make the workload irregular

Outline

1、Pruning-based Graph Structural Clustering

2、Performance Bottleneck & Challenge

3、Parallelization & Vectorization

4、Experimental Study

5、Conclusion

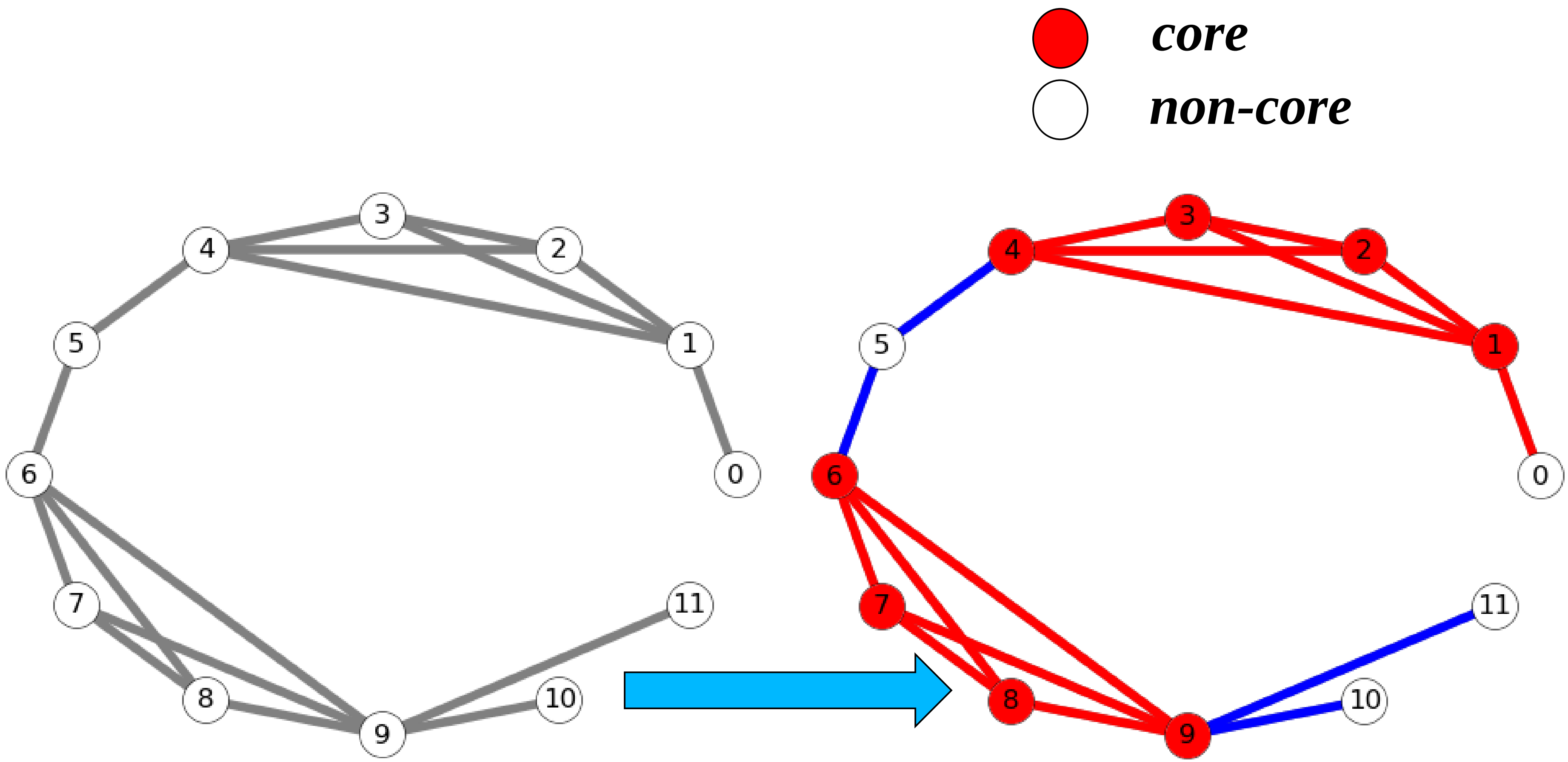
Two-Step Multi-Phase Design Overview

- **Step 1: Role Computing (Determining Core or Non-Core)**
  - similarity pruning phase (without similarity computation)
  - core checking phase
  - core consolidating phase (finalizing roles of all the vertices)

- **Step 2: Core and Non-Core Clustering**
  - core clustering without similarity computation phase
  - finalizing core clustering with similarity computation phase
  - cluster id initialization phase
  - non-core clustering (cluster id assignment) phase

- **Computation Optimizations**
  - ***degree-based task scheduling*****:** dealing with the workload skewness
  - ***pivot-based set-intersection vectorization***: improving the efficiency of similarity computation

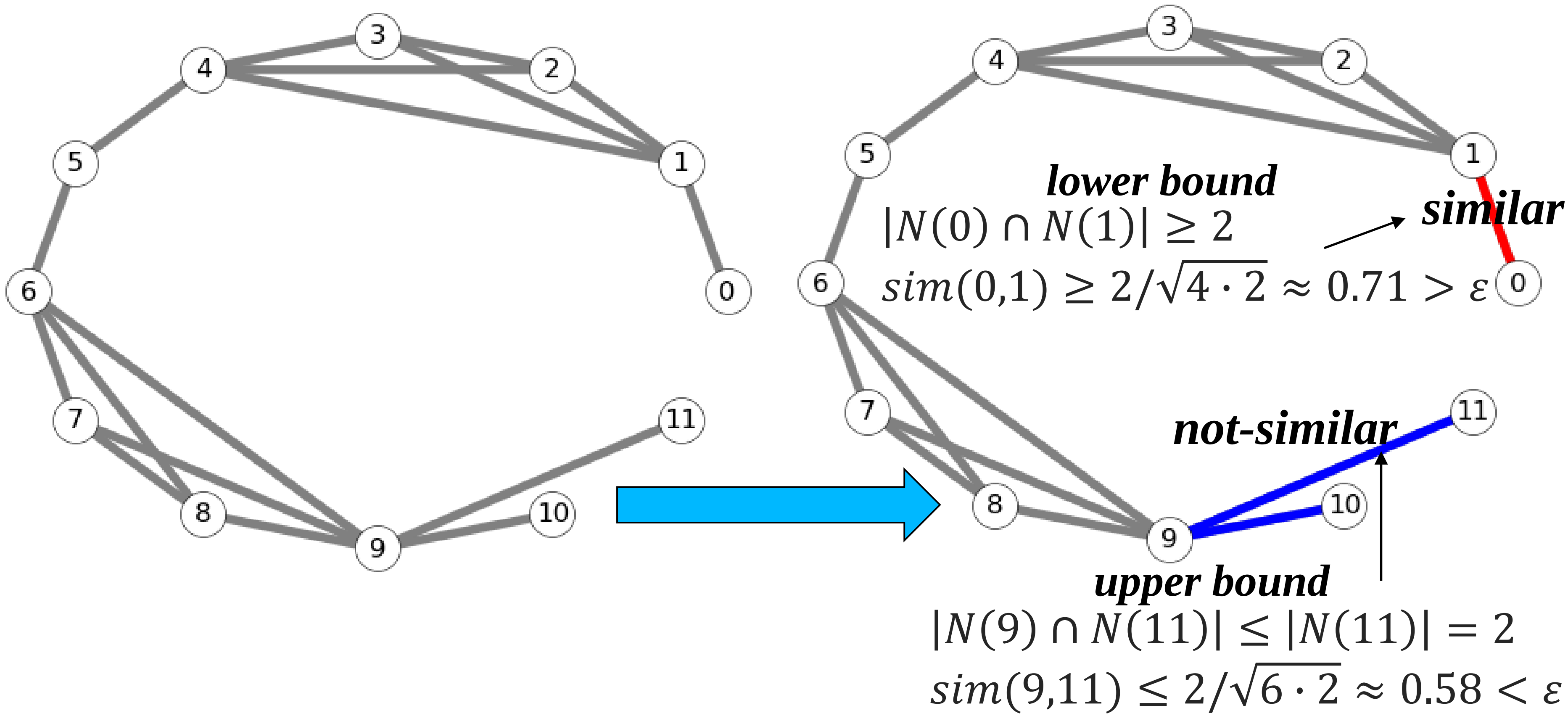
Step 1 : Role Computing



finalizing vertex roles (either ***core*** or ***non-core***),
stashing some similarity values (***unknow***/***similar***/***not-similar***)

Similarity Pruning

- **Utilize Similarity Definition & Min-Max Pruning**
  - do not incur set intersections (similarity computations)
  - utilize lower and upper bounds of number of similar neighbors



Similarity Pruning

- **Utilize Similarity Definition & Min-Max Pruning**
  - do not incur set intersections (similarity computations)
  - utilize lower and upper bounds of number of similar neighbors

```
Procedure PruneSim(u)
    foreach v ∈ N(u) do
        sim[e(u, v)] ← Unkown
        Update sim[e(u, v)] using the similarity predicate pruning
        if sim[e(u, v)] == Sim then
            sd ← sd + 1
        else if sim[e(u, v)] == NSim then
            ed ← ed − 1
    if sd ≥ μ then
        role[u] ← Core
    else if ed < μ then
        role[u] ← NonCore
    else
        role[u] ← Unknown
```

lower bound

upper bound

```
foreach u ∈ V in parallel do
    PruneSim(u)
```

this phase determines some vertex roles ***without similarity computations***

local variables ***sd*** (similar degree) and ***ed*** (effective degree): ***lower and upper bounds*** of u's similar neighborhood size

Core Checking and Consolidating

- **Vertex Exploration Order Constraint ($u < v$):**
  - to guarantee no redundant computation: each undirected edge is computed at most once for the similarity value

- **Core Checking and Consolidating Two-Phase**
  - to apply ***similarity reuse*** technique given vertex order constraint, while finalizing all vertex roles

foreach $u \in V$ ***and*** $role[u] == Unknown$ ***in parallel*** **do**
    $CheckCore(u)$
foreach $u \in V$ ***and*** $role[u] == Unknown$ ***in parallel*** **do**
    $ConsolidateCore(u)$

two phases

after the two parallel phases,
all vertex roles are known

Core Checking

```
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)
```

two phases

```
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
```

1) initialize ***pruning related lower and upper bounds***, and see if we can benefit from ***parallel core checking*** from u's neighbors with the ***min-max pruning*** technique

Core Checking

3 **foreach** $u \in V$ ***and*** $role[u] == Unknown$ ***in parallel*** **do**
4 $\quad CheckCore(u)$
5 **foreach** $u \in V$ ***and*** $role[u] == Unknown$ ***in parallel*** **do**
6 $\quad ConsolidateCore(u)$

two phases

21 **Procedure** $CheckCore(u)$
22 $\quad$ **foreach** $v \in N(u)$ **do**
23 $\quad\quad$ **if** $sim[e(u, v)] == Sim$ **then**
24 $\quad\quad\quad sd \leftarrow sd + 1$
25 $\quad\quad\quad$ **if** $sd \geq \mu$ **then**
26 $\quad\quad\quad\quad role[u] \leftarrow Core$, **return**
27 $\quad\quad$ **else if** $sim[e(u, v)] == NSim$ **then**
28 $\quad\quad\quad ed \leftarrow ed - 1$
29 $\quad\quad\quad$ **if** $ed < \mu$ **then**
30 $\quad\quad\quad\quad role[u] \leftarrow NonCore$, **return**
31 $\quad$ **foreach** $v \in N(u)$ ***and*** $u < v$ ***and*** $sim[e(u, v)] == Unknown$ **do**
32 $\quad\quad sim[e(v, u)] \leftarrow sim[e(u, v)] \leftarrow CompSim(u, v)$
33 $\quad\quad$ Update sd, ed and $role$ **in the same logic as Lines 23-30**

2) determine some vertex roles, and apply the ***similarity reuse*** and ***min-max pruning*** techniques

Core Consolidating

```
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)
```

two phases

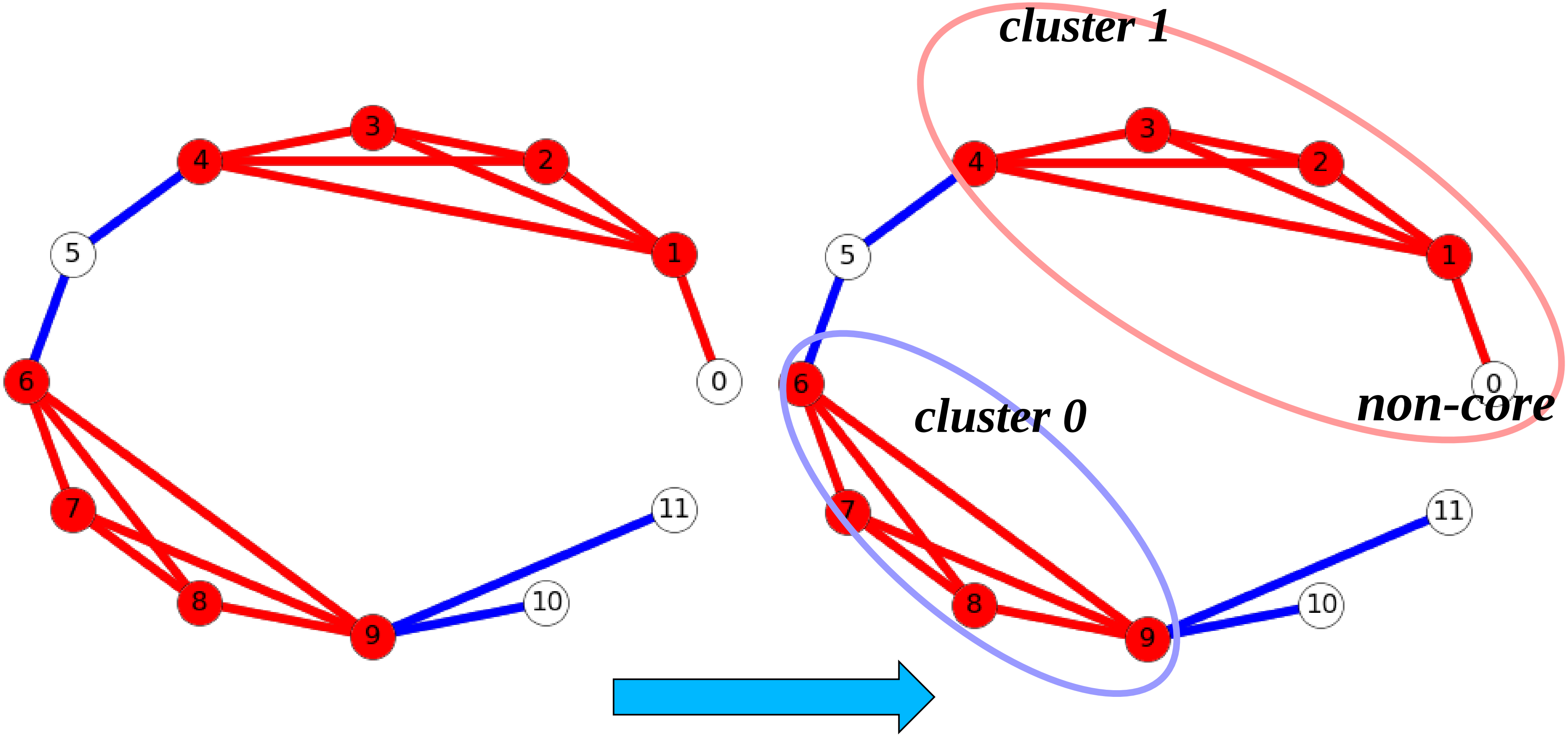
```
21 Procedure CheckCore(u)
22 |   foreach v ∈ N(u) do
23 |   |   if sim[e(u, v)] == Sim then
24 |   |   |   sd ← sd + 1
25 |   |   |   if sd ≥ μ then
26 |   |   |   |   role[u] ← Core, return
27 |   |   else if sim[e(u, v)] == NSim then
28 |   |   |   ed ← ed − 1
29 |   |   |   if ed < μ then
30 |   |   |   |   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
```

finalizing all vertex roles

```
34 Procedure ConsolidateCore(u)
35 |   Do the same as CheckCore(u), except for removing the constraint
   |     u<v in Line 31
```

work-efficiency-proof: the similarity computation **is at most invoked once** for the similarity values $\boldsymbol{sim[e(u,v)]}$ and $\boldsymbol{sim[e(v,u)]}$

Step 2 : Core and Non-Core Clustering



core and non-core clustering

Core Clustering

- **Two Phase Separation**
  - core clustering using already known similarity values ***without set intersections***
  - finalizing core clustering ***with set intersections***

foreach $u \in V$ ***and*** $role[u] == Core$ ***in parallel*** **do**
  $ClusterCoreWithoutCompSim(u)$
foreach $u \in V$ ***and*** $role[u] == Core$ ***in parallel*** **do**
  $ClusterCoreWithCompSim(u)$

two phases

- **Avoiding Redundant Computation**
  - adding $u < v$ constraint during the clustering for both phases
  - applying union-find pruning in the second phase

Procedure $ClusterCoreWithoutCompSim(u)$
  foreach $v \in N(u)$ ***and*** $role[v] == Core$ ***and*** $u < v$ ***and***
   not $IsSameSet(u, v)$ ***and*** $sim[e(u, v)] == Sim$ **do**
    $Union(u, v)$
Procedure $ClusterCoreWithCompSim(u)$
  foreach $v \in N(u)$ ***and*** $role[v] == Core$ ***and*** $u < v$ ***and***
   not $IsSameSet(u, v)$ ***and*** $sim[e(u, v)] == Unknown$ **do**
    $sim[e(u, v)] \leftarrow CompSim(u, v)$
    if $sim[e(u, v)] == Sim$ **then**
     $Union(u, v)$

vertex order constraint

vertex order constraint
union-find pruning

Non-Core Clustering

- **Two Phase Separation**
  - ***cluster id initialization*** using atomic operations
  - ***non-core cluster id assignment*** from all the cores to similar neighbors (to form final results)

```
Procedure InitClusterId(u)
    ru ← FindRoot(u)
    do
        min_core_id ← cluster_id[ru]
        if u ≥ min_core_id then
            break
    while not CAS(&cluster_id[ru], min_core_id, u)
Procedure ClusterNonCore(u)
    foreach v ∈ N(u) and role[v] == NonCore do
        if sim[e(u, v)] == Unknown then
            sim[e(u, v)] ← CompSim(u, v)
        if sim[e(u, v)] == Sim then
            Assign cluster_id[FindRoot(u)] to the NonCore v
```

CAS atomic operations

cluster id assignment

Degree-based Task Scheduling

- **Dynamic Scheduling**
  - vertex computations relate to the degree and role of the vertex in all the phases, for the ***exploration of neighbors*** and ***computations***
  - a task can be represented with $[v_{beg}, v_{end})$, and the parameter of range size is tunable (in our experimental setting: 32768)

```
InitThreadPool(), deg_sum ← 0, beg ← 0
for u ← 0; u < |V|; u ← u + 1 do
    if role[u] == Unknown then
        deg_sum ← deg_sum + d[u]
        if deg_sum > 32768 then
            SubmitTaskToPool(Task(beg, u + 1))
            deg_sum ← 0, beg ← u + 1
SubmitTaskToPool(Task(next_beg, |V|)), JoinThreadPool()
Procedure Task(beg, end)
    for u ← beg; u < end; u ← u + 1 do
        if role[u] == Unknown then
            CheckCore(u)
```

degree-accumulation-based-task-submission

handling tail task

Dynamic Scheduling Example for Core Checking

Set-Intersection Vectorization

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

upper and lower bounds

early termination

1. Find the first element in u's sorted neighbors ≥ pivot_v

2. Find the first element in v's sorted neighbors ≥ pivot_u

3. Count the match

Outline

1、 Pruning-based Graph Structural Clustering

2、 Performance Bottleneck & Challenge

3、 Parallelization & Vectorization

4、 Experimental Study

5、 Conclusion

Experimental Setup

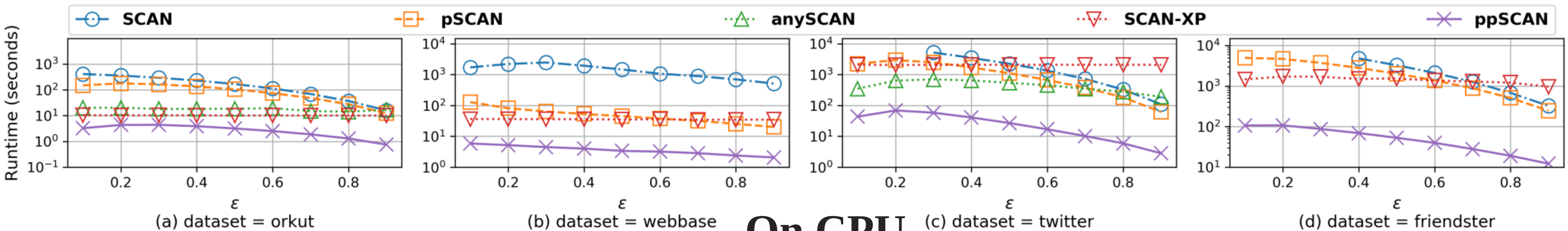
- **Environments**
  - ***Xeon Phi Processor (KNL):*** 64 cores (2 VPUs / core), AVX512, 64KB/1024KB L1/L2 caches, 16GB MCDRAM (cache mode), 96GB RAM
  - ***Xeon CPU E5-2650***: 20 cores, AVX2, 64KB/256KB/25MB L1/L2/L3 cache, 64GB RAM

- **Algorithms**
  - Sequential: ***SCAN*** [Xu+, KDD'07], ***pSCAN*** [Chang+, ICDE'16]
  - Parallel: ***anySCAN*** [Mai+, ICDE'17], ***SCAN-XP*** [Takahashi, NDA'17], our ***ppSCAN***, ***ppSCAN-NO*** (without vectorization)

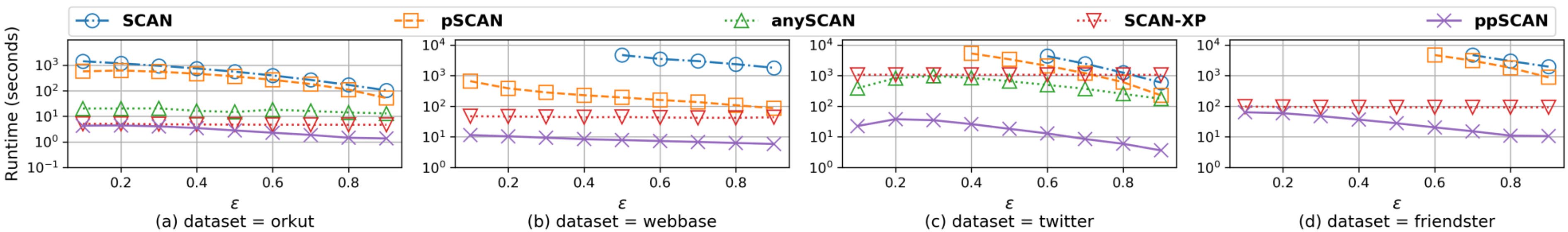
- **Graphs (Billion-Edge)**
  - Real-World: Orkut/Twitter/Friendster (***Social***), Webbase (***Web***)
  - Synthetic Power-Law: ***ROLL*** Graphs [Hadian+, SIGMOD'16]

Overall Performance (on CPU and KNL)

- **Algorithm Comparison**
  - ***SCAN*** and ***pSCAN*** lack parallelization
  - ***SCAN-XP*** lacks usage of pruning techniques
  - ***anySCAN*** suffers from heavy synchronization and poor memory locality, and runs out of memory on webbase and friendster datasets
  - ***ppSCAN*** has good memory locality and negligible synchronization overheads, and utilizes vectorization AVX2 on CPU and AVX512 on KNL



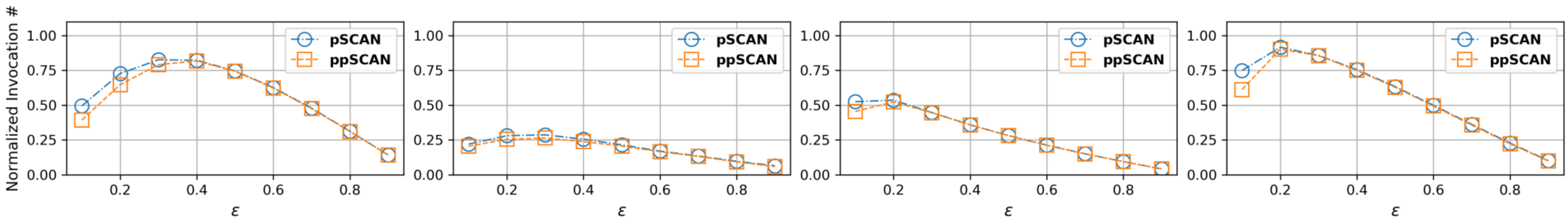
On CPU



On KNL

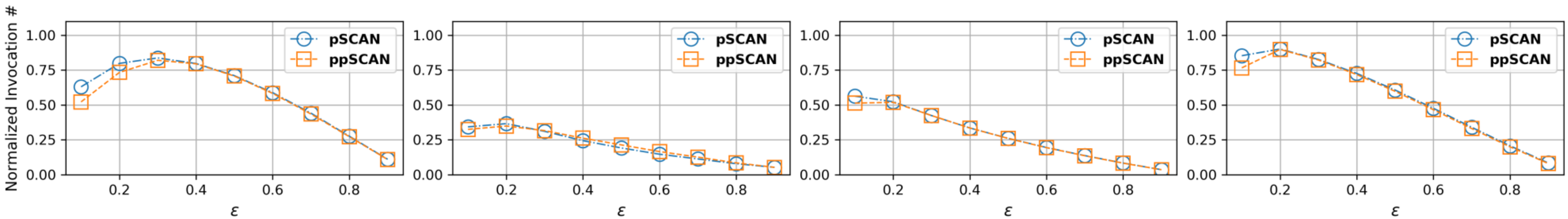
Set-Intersection Invocation Reduction

- **Work Efficiency**
  - multi-phase computation does not introduce more workload
  - ppSCAN even compute less because of the ***similarity pruning*** and ***parallel core checking benefits***



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

$\boldsymbol{\mu = 5}$

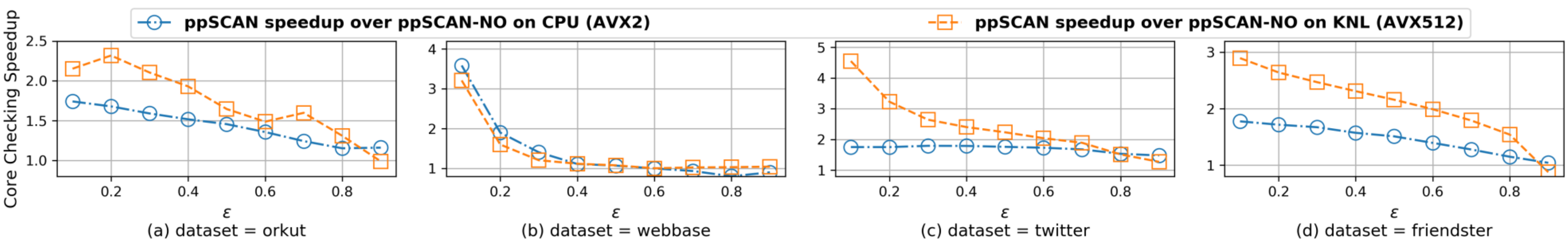


(a) dataset = orkut (b) dataset = webbase (c) dataset = twitter (d) datset = friendster

$\boldsymbol{\mu = 10}$

Set-Intersection Vectorization Improvement

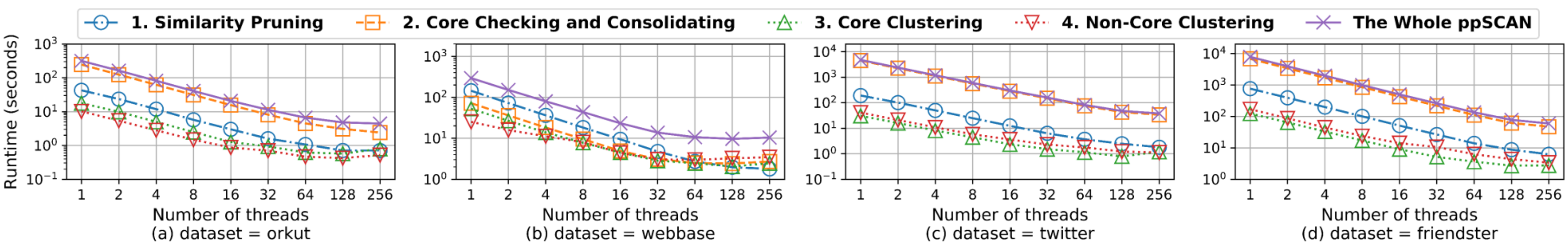
- **Core Checking Speedup from Vectorization**
  - on CPU: at most ***3.5x***, on KNL: at most ***4.5x***
  - vectorization have better performance with more workloads (when memory access is not a bottleneck, e.g., when $\varepsilon = 0.1$, ***intensive set intersections*** hide the ***memory access latency***



$\mu = 5$ (On Both CPU and KNL)

Scalability to Number of threads

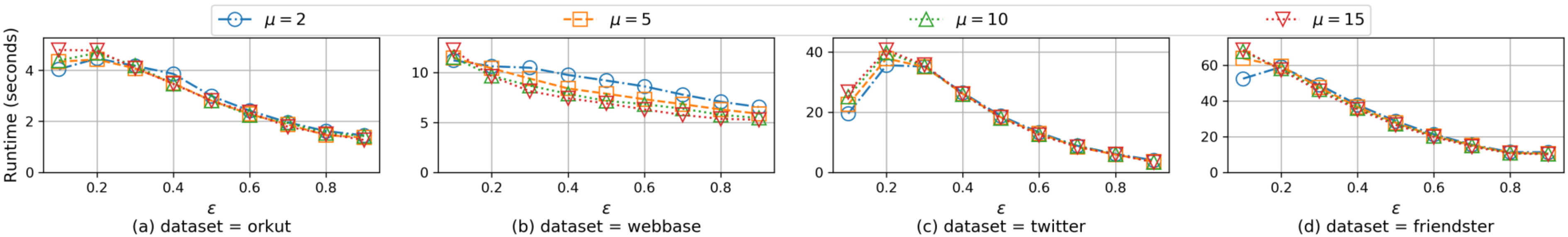
- **Scalability and Time Breakdown**
  - all the four phases scale well to number of threads
  - speedup of ***core checking*** is better than other phases, because the ***intensive set intersection computations*** hide the ***memory access latency***
  - ***time breakdown ratio***:
    - core checking and consolidating > similarity pruning > non-core clustering > core clustering



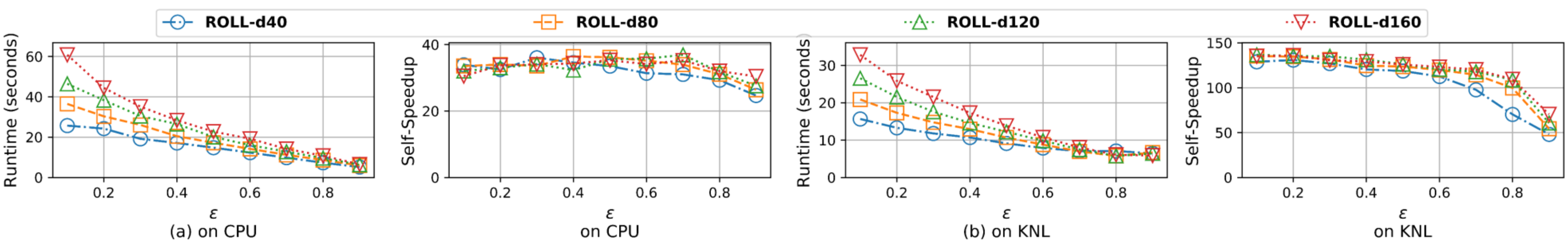
$$\boldsymbol{\varepsilon = 0.2, \mu = 5}$$ **(On KNL)**

Robustness

- **Given Different Parameters and Graphs**
  - robust on ***varying input parameters***
    - finish computations within ***70*** seconds



  - robust on ***synthetic graphs (1-billion edge)***
    - finish computations within ***60*** seconds
    - achieve up to ***135x*** self-speedup on KNL



Outline

1、Pruning-based Graph Structural Clustering

2、Performance Bottleneck & Challenge

3、Parallelization & Vectorization

4、Experimental Study

5、Conclusion

Conclusion

- **Parallelization & Vectorization Design**
  - multi-phase lock-free parallel vertex computations
  - dynamic degree-based vertex computation task scheduling
  - pivot-based set intersection vectorization

- **Experimental Study**
  - ***ppSCAN*** is about ***2x*** faster on KNL (64 cores, ***AVX512***) than on Xeon CPU (20 cores, AVX2) because of wider SIMD width
  - on KNL, ***two orders of magnitude*** faster than the sequential ***pSCAN***
  - on KNL, ***one order of magnitude*** faster than the parallel ***anySCAN*** and ***SCAN-XP***
  - on KNL, up to ***135x*** self-speedup (over single-thread ***ppSCAN***)

End - Q & A

- **Source Codes / Figures / Related Projects :**
  https://github.com/GraphProcessor/ppSCAN

- **More Experimental Studies (Scripts / Figures):**
  https://github.com/GraphProcessor/ppSCAN/tree/master/python_experiments

- **This PPT:**
  https://www.dropbox.com/sh/i1r45o2ceraey8j/AAD8V3WwPElQjwJ0-QtaKAzYa?dl=0&preview=ppSCAN.pdf