Speeding Up Set Intersections in Graph Algorithms using SIMD Instructions

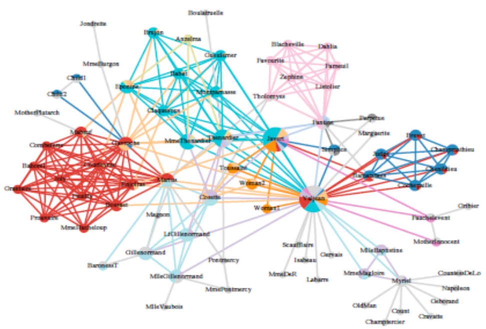
--Lei Zou

Joint work with Shuo Han and Jeffrey Xu Yu

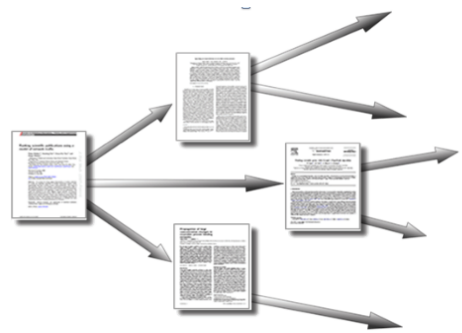
@SIGMOD 2018

Background

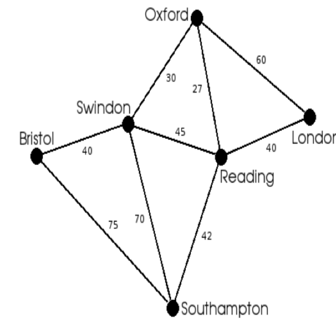
Graph is everywhere:



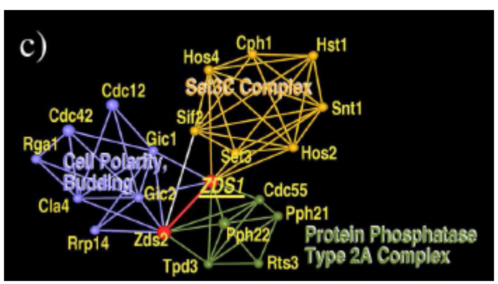
Social Network



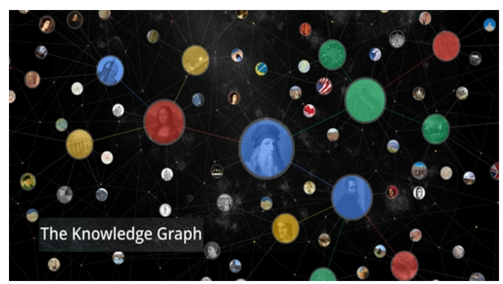
Citation Network



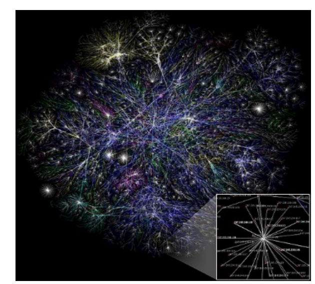
Road Network



Protein Network



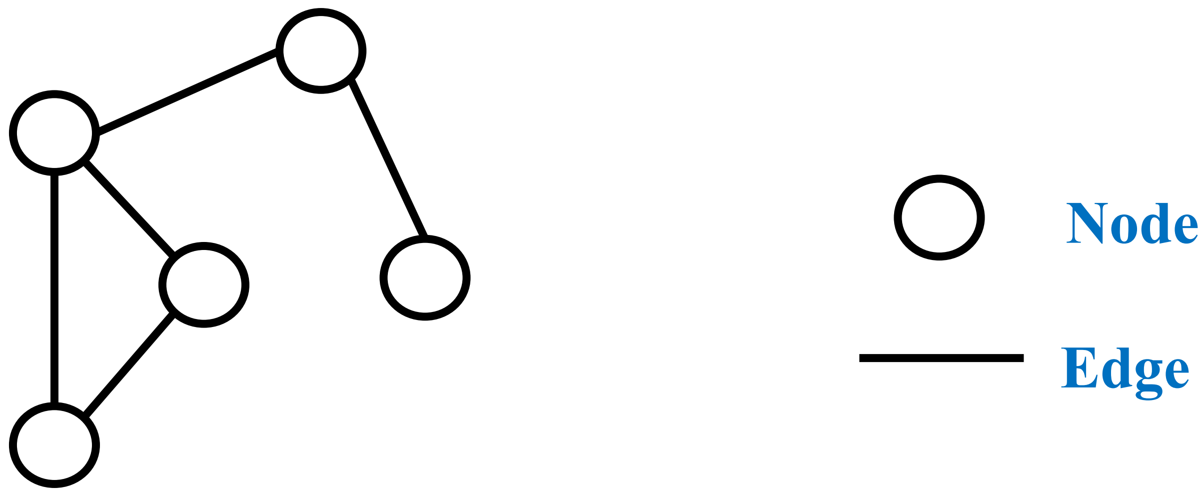
Knowledge Graph



Internet

Background

A graph is a set of nodes and edges that connect them:



Background

How to represent a large sparse graph ?

- Adjacency Matrix ✗
- Adjacency List ✓

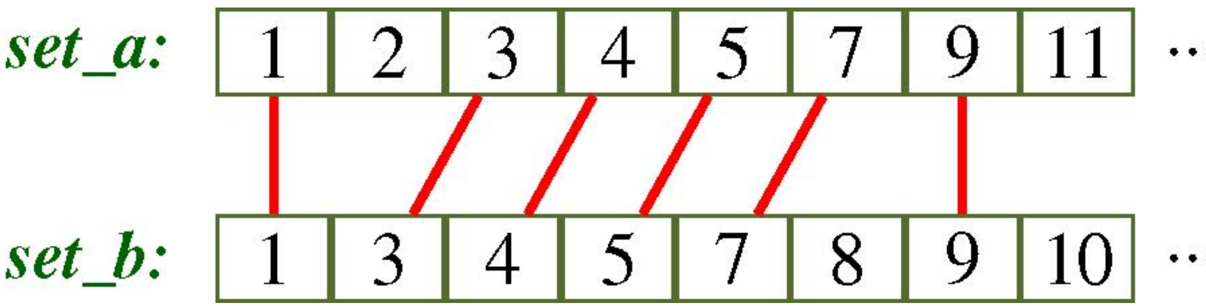
Outline

- Motivation
- Related Work
- Data Structure (Base and State Representation)
- Algorithm (QFilter, SIMD-based)
- Graph Re-ordering
- Experiments

Motivation

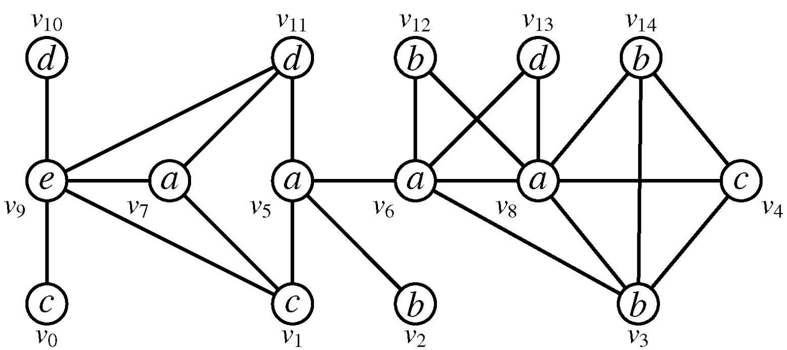
- Set Intersection

Problem Definition: Given two sets A and B, how to compute $A \cap B$ efficiently ?



Motivation

- Why Set-Intersection is *important* in graph algorithms/systems.



Common Computing Pattern in Graph Algorithms.

- Triangle Counting [1]
- Clique Detection [2]
- Subgraph Isomorphism [3,4]
- Graph Simulation [5]

……

Important Component in Graph System

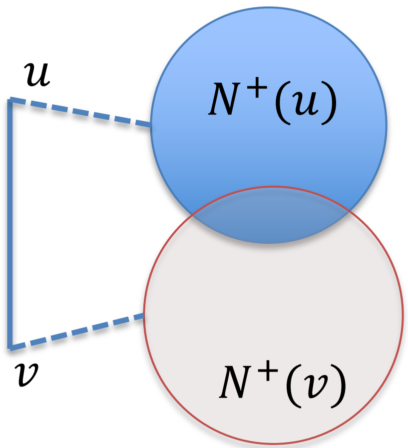
- EmptyHead [6]
- gStore [7]

……

Motivation

- Triangle Counting

Given a graph G, returns the number of triangles involved in the graph.

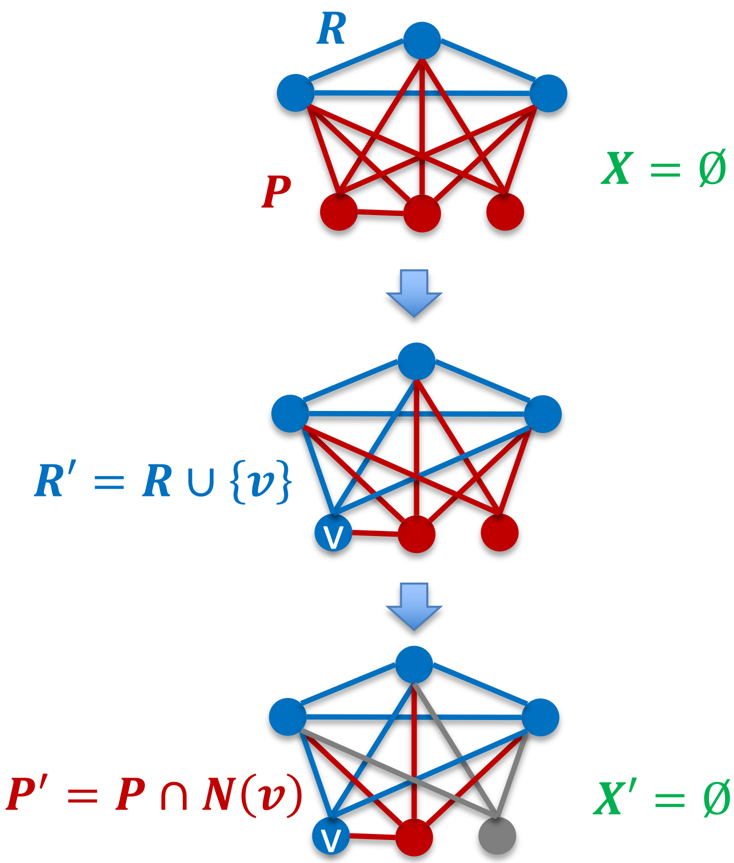


- Compute a descending order of node degree R, such that if $R(v) < R(u)$ then $Deg(v) \leq Deg(u)$;
- **For** $v \in V$ **do**:
  - $N^+(v) = \{u \in N(v) \mid R(v) < R(u)\}$
- **For** $(v, u) \in E$ and $R(v) < R(u)$ **do**:
  - $I = \boldsymbol{INTERSECT}\left(N^+(v), N^+(u)\right)$
  - $\Delta = \Delta \cup \{(v, u) \times I\}$

Motivation

- Maximal Clique Detection

Given a graph G, returns all maximal cliques in the graph.



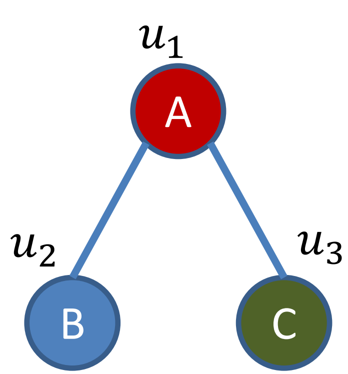
- $BroKerbosch(R, P, X)$:
  - **If** $P = \emptyset$ **and** $X = \emptyset$:
    - Report R as a maximal clique
  - **For** $v \in P$ **do**:
    - $R' = R \cup \{v\}$
    - $P' = \boldsymbol{INTERSECT}(P, N(v))$
    - $X' = \boldsymbol{INTERSECT}(X, N(v))$
    - **Call** $BroKerbosch(R', P', X')$
    - $P = P \setminus \{v\}$
    - $X = X \cup \{v\}$

Motivation

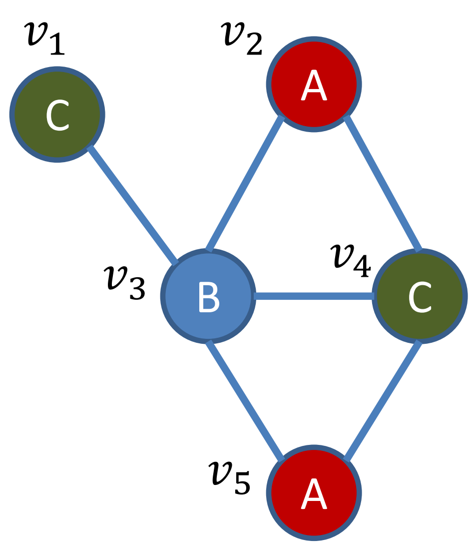
- Subgraph Isomorphism

Neighbor Connection Pruning

Used in ULLMAN [8],VF2 [3] and TurboISO [4] algorithms



Q



G

Step 1: Finding Candidate Matching nodes (only considering vertex labels)

$C(u_1) = \{v_2, v_5\}$

$C(u_2) = \{v_3\}$

$C(u_3) = \{v_1, v_4\}$

Step 2: Neighbor Connection Pruning

Considering edge (u_1, u_3)

$$\boxed{N(v_1) = \{v_3\} \cap C(u_1) = \phi}$$

=> $C(u_3) = \{v_1, v_4\}$

Motivation

Let us see some experiment results

Profiling of 3 Representative Graph Algorithms

	# Set Inter. Calls	Set Inter. Time	Total Time	Prop.
Triangle Counting	21274216	9.9s	10.5s	94.3%
Maximal Clique	254503699	120.7s	164.1s	73.6%
Subgraph Matching	120928579	31.5s	54.1s	58.2%

Set Intersection plays an important role!

Motivation

- Why Set-Intersection Important ?

Primitive Operations

set-intersection

Triangle Counting/Listing
- Clustering Coefficient
- Triangle−based Community Detection $(K - truss)$
- … …

Clique Detection
- Maximal Commong Subgraph
- Dense Subgraph Detection $(K - clique)$
- … …

Subgraph Matching
- SPARQL Query
- Social Group Finding
- … …

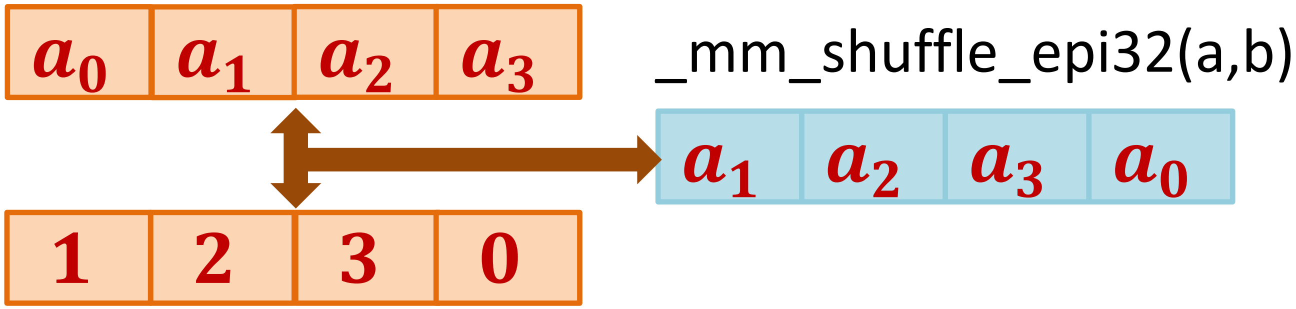
Speeding up set-intersection will result in accelerating a bunch of graph computing tasks.

Related Work

- SIMD Instructions

SIMD: Single instruction multiple data.

C-intrinsics	Meanings
_mm_load_si128()	Load consecutive 128 bit piece of data from memory that aligned on a 16-byte boundary to a SIMD register.
_mm_store_si128()	Write the content of a register to aligned memory.
_mm_shuffle_epi32(a,b)	Shuffle 32-bit integers in a according to the control mask in b.
_mm_and_si128(a,b)	Compute bitwise AND of 128 bits data in a and b.

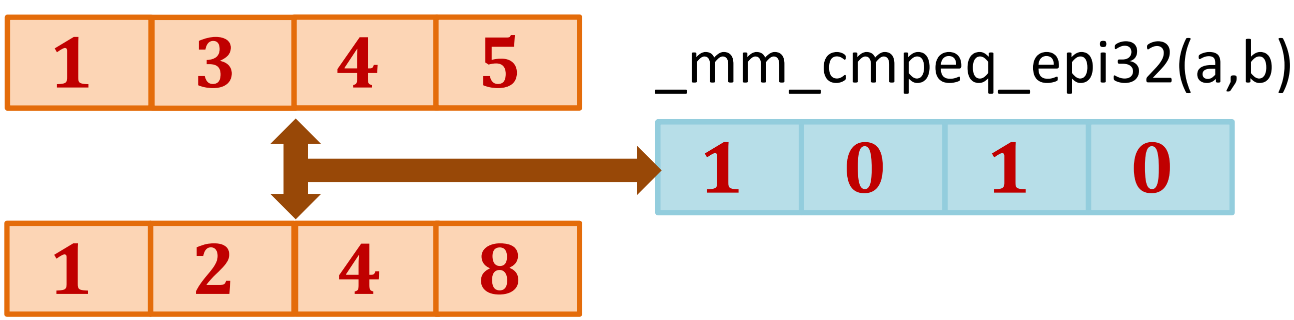


Related Work

- SIMD Instructions (continued)

SIMD: Single instruction multiple data.

C-intrinsics	Meanings
_mm_andnot_si128(a,b)	Compute the bitwise NOT of 128 bits data in a and then AND with b.
_mm_cmpeq_epi32(a,b)	Compares the four 32-bit integers in a and b for equality.
_mm_movemask_ps()	Create masks for the most significant bit of each 32-bit integer
_mm_movemask_epi8()	Creat masks for the most significant bit of each 8-bit integer



Related Work

- Pairwise Set-Intersection

Merge-based Solution

Algorithm 1: Merge-based Intersection (non-SIMD)

```
int i = 0, j = 0, size_c = 0 ;
while i < size_a && j < size_b do
    if set_a[i] == set_b[j] then
        set_c[size_c ++] = set_a[i] ;
        i++; j++ ;
    else if set_a[i] < set_b[j] then  i++ ;
    else  j++ ;
return set_c, size_c ;
```

# of comparisons:

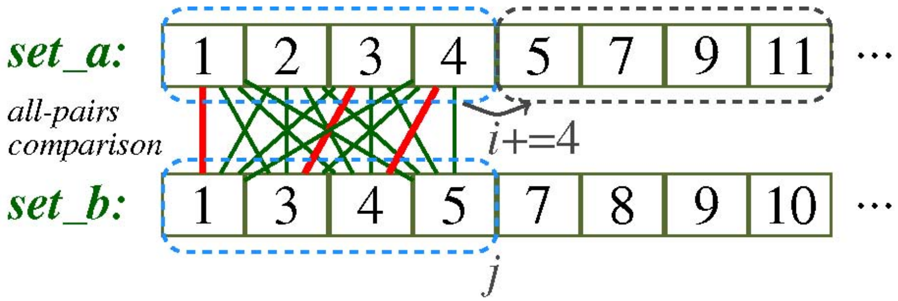
Best case: $\mathrm{Min}(|S_a|, |S_b|)$

Worst case: $|S_a| + |S_b|$

Related Work

- Pairwise Set-Intersection

SIMD Merge-based Solution [10, 11, 12]



Step 1: (LOAD)
Load two blocks of elements from two arrays into SIMD registers (using *_mm_load_si128()*).

Step 2: (COMPARE)
Make all-pairs comparison between two blocks in parallel.

- Employing SIMD compare instructions (*_mm_compeq_epi32()*)
- Pack the common values together by shuffle instructions (*_mm_shuffle_epi32()*)
- Store them in the result array (*_mm_store_si128()*)

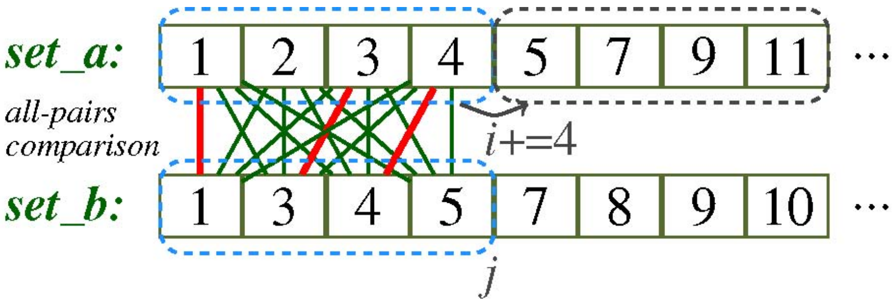
Step 3: (FORWARD)
Compare the last elements of the two blocks. If equal, move forward both pointers; otherwise, only advance the pointer of the smaller one to the next block.

Related Work

- Pairwise Set-Intersection

SIMD Merge-based Solution (Shuffling [11])

Step 2: (COMPARE)
Make all-pairs comparison between two blocks in parallel.



set_c

1	3	4	

Loop 1:

1	2	3	4
1	3	4	5

1	0	0	0

Loop 2:

1	2	3	4
5	1	3	4

0	0	1	1

Loop 3:

Loop 4:

1	0	1	1

Final mask.

Related Work

- Pairwise Set-Intersection

Merge-based solution does not work well when two set sizes are significantly different (e.g., $\frac{|S_a|}{|S_b|} \geq 32 \; or \; \frac{|S_b|}{|S_a|} \geq 32$) .

Binary Search-based method works, e.g. Galloping [9]

Algorithm 2: Galloping Intersection (non-SIMD)

// suppose $size\_a \ll size\_b$

$i = 0$; $j = 0$; $size\_c = 0$;

while $i < size\_a$ && $j < size\_b$ **do**

    sequential search the smallest r ($r = 2^0, 2^1, 2^2, \ldots$), such that $set\_b[j + r] \geq set\_a[i]$;

    binary search the smallest r' in range $[r/2, r]$ such that $set\_b[j + r'] \geq set\_a[i]$;

    if $set\_a[i] == set\_b[j + r']$ **then**

        $set\_c[size\_c{+}{+}] = set\_a[i]$;

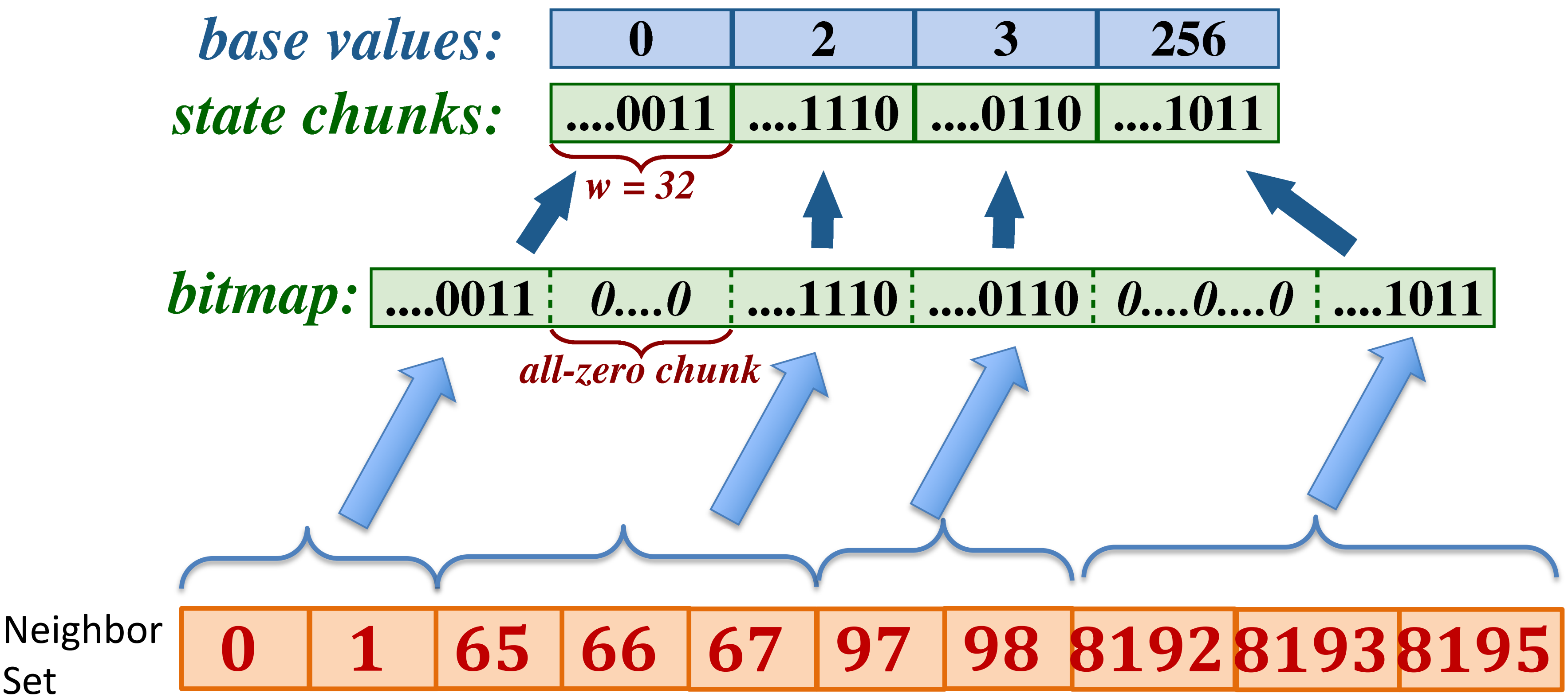
    $i{+}{+}$; $j{+}{=}\, r'$;

return set_c, $size\_c$;

Outline

- Motivation
- Related Work
- Our work
  - Data Structure (Base and State Representation)
  - Algorithm (QFilter, SIMD-based)
  - Graph Re-ordering
- Experiments

Base and State Representation



Our Algorithm-QFilter

- INPUTS: two sets in BSR format $(bv_a, sv_a);(bv_\mathrm{b}, sv_b)$

bv_a	0	1	2	259
sv_a	...0100	1001	0111	...1101

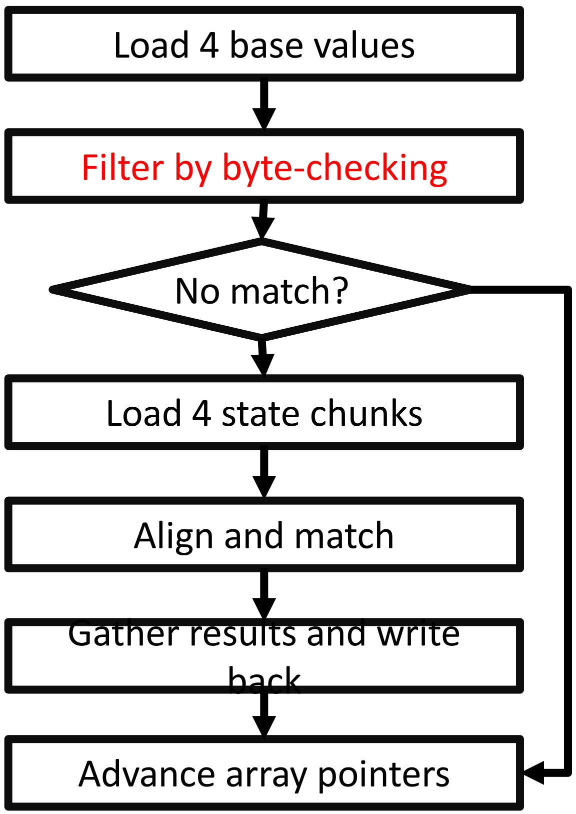
bv_b	0	2	3	256
sv_b	...0011	...1110	...0110	...1011

- OUTPUT: the intersection set (bv_c, sv_c)

bv_c	2			
sv_c	...0110			

Our Algorithm - QFilter

Overview

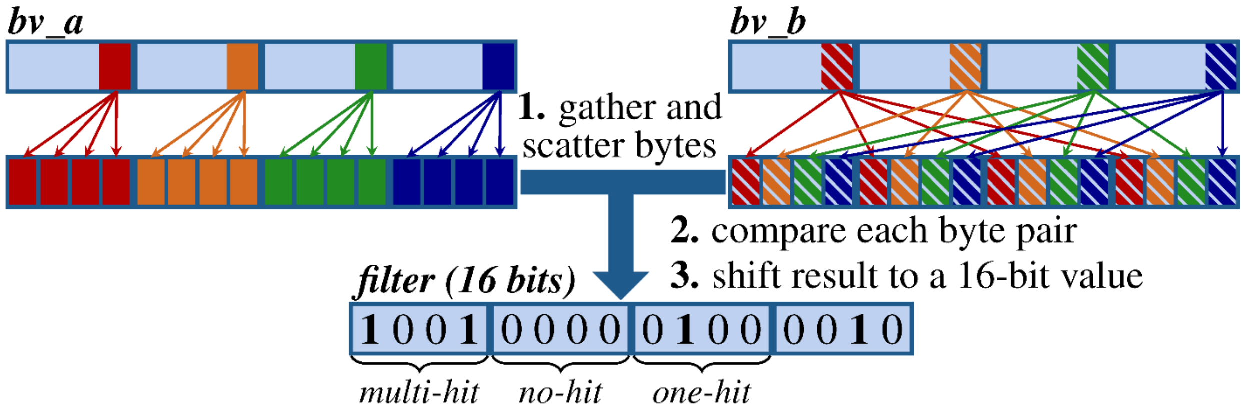


Main Stage 1:
Compare the base values from the two sets. We quickly filter out the redundant comparisons by byte-checking using SIMD instructions.

Main Stage 2:
For the matched base values, we execute the bitwise AND operation on the corresponding state chunks.

Filter Step

Considering the least significant byte of each base value.



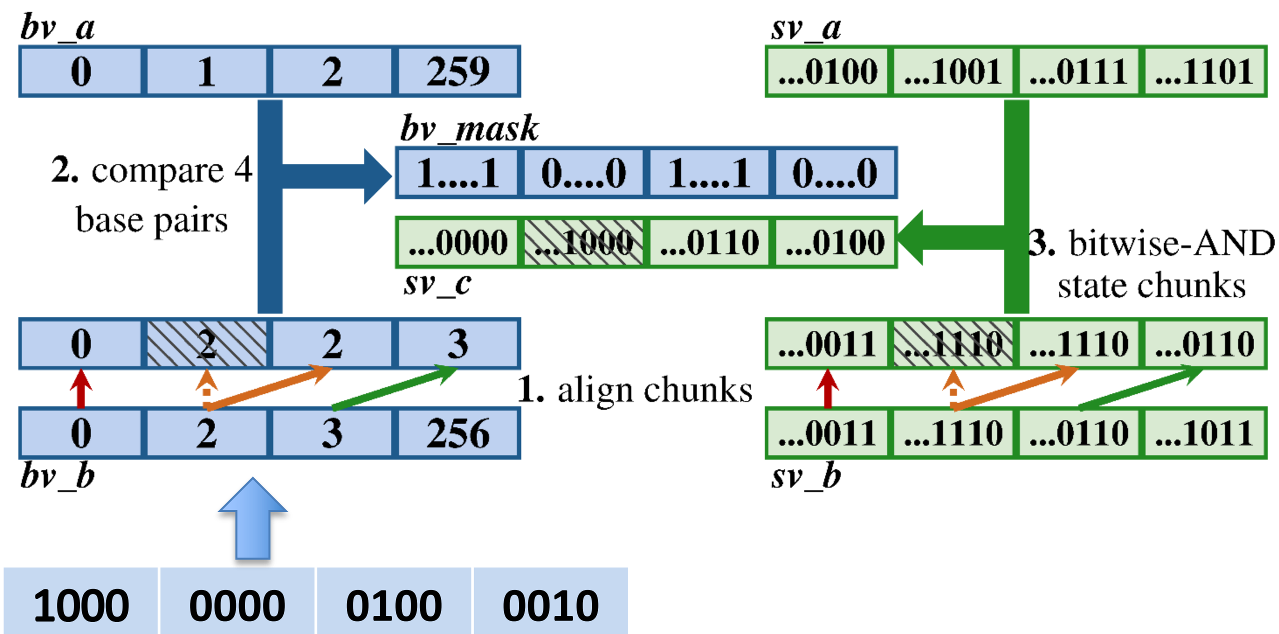
If there exists multi-hit cases, it must be false positive . We need to check the next byte.

!!! BUT we claim "multi-hit" case rarely happens (both theoretical analysis and experiment results) ***(less than 1.9%)***

The filter step stops when there are only no-hit or one-hit.

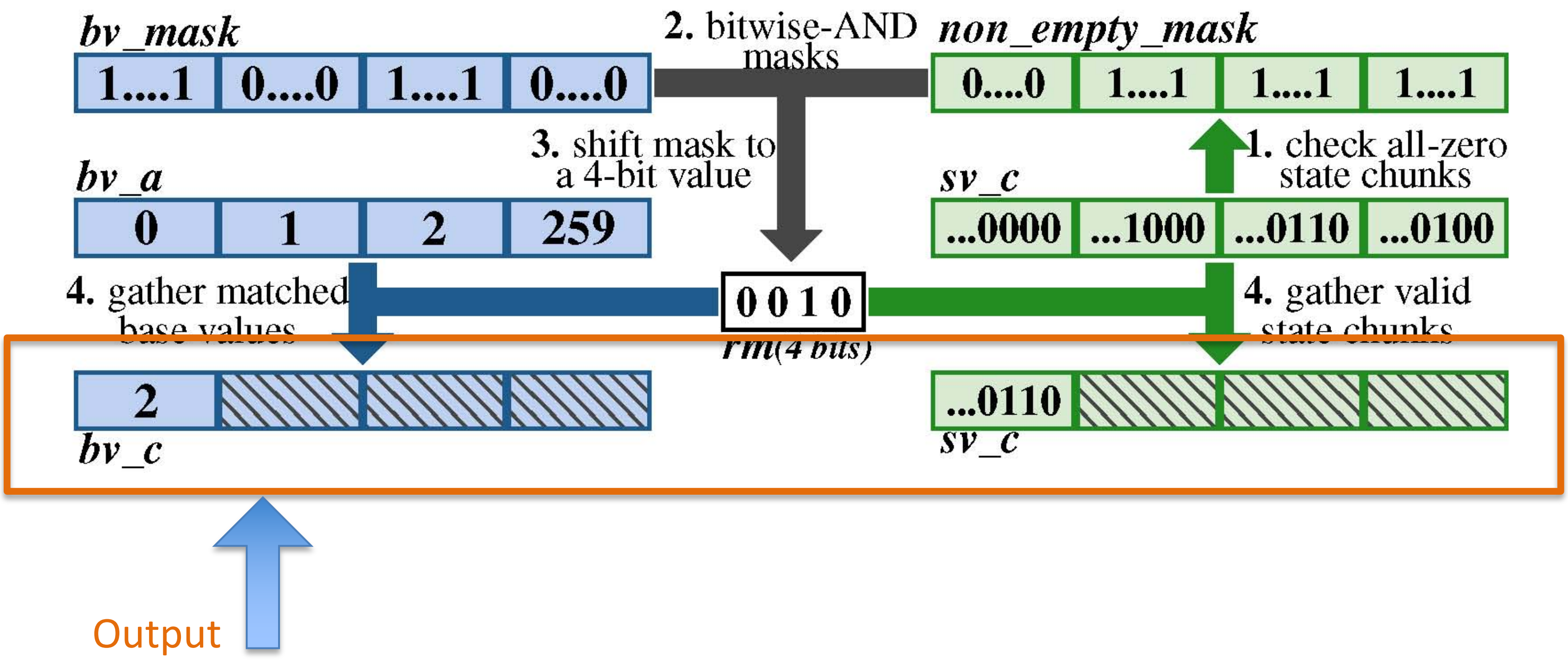
Filter vector	**1000**	**0000**	**0100**	**0010**

Align and match



Filter vector

Align and match



Our Algorithm - QFilter

Intra-chunk and Inter-chunk Parallelism

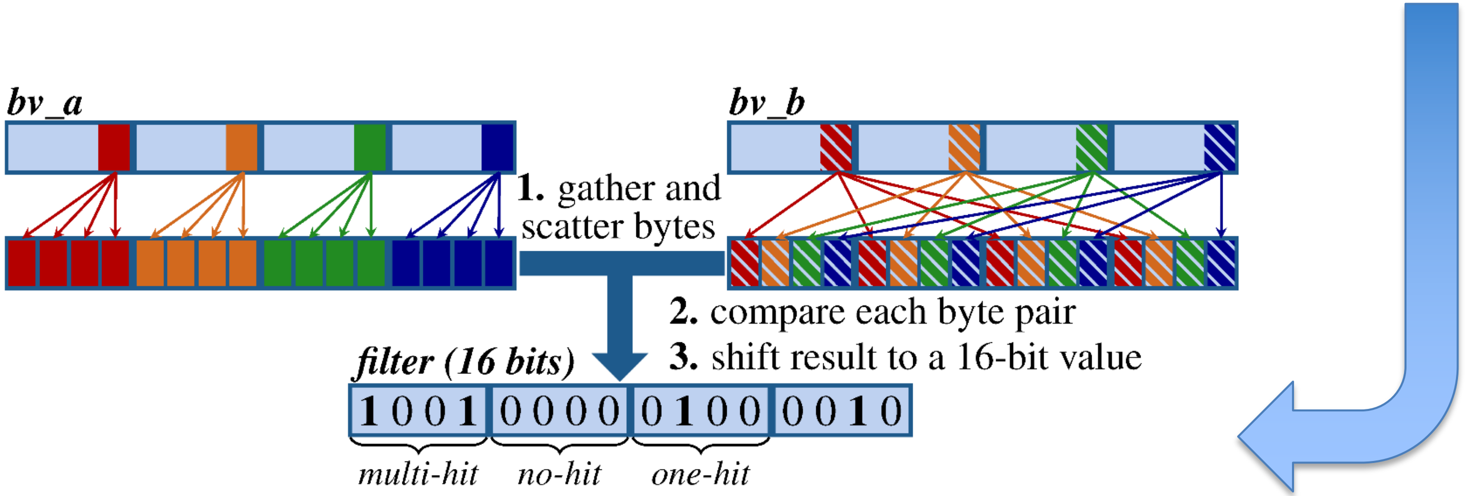
- **Intra-chunk Parallelism**:
  - Each chunk in BSR represents several elements by an integer. In this way, we can process multiple elements within a chunk.
- **Inter-chunk Parallelism**:
  - We can process multiple chunks simultaneously by SIMD instructions.

Intra-chunk Parallelism + Inter-chunk Parallelism → More than ***10x*** *speedup!*

Quantitative Analysis

Why "multi-match" rarely happens ?

A match that includes at least one "multi-hit" is called "**multi-match**".



Quantitative Analysis

Why "multi-match" rarely happens ?

Definition. (Selectivity) : Given two sets S_a and S_b, the selectivity is defined as follows:

$$selectivity = \frac{|S_c|}{MIN(|S_a|, |S_b|)}$$

Let p to be the probability of successful matching for one comparison as a random variable.

If the intersection algorithm takes C comparisons in total, the probability p is

$$p = \frac{|S_c|}{C}$$

Since $16 \cdot MIN\left(\frac{|S_a|}{4}, \frac{|S_b|}{4}\right) \leq C \leq 16 \cdot \left(\frac{|S_a|}{4} + \frac{|S_b|}{4}\right)$

Thus, $p \leq 0.25 \cdot selectivity \leq 0.25$

Quantitative Analysis

Why “multi-match” rarely happens ?

In the byte-checking filter step, suppose that the range of base values is up to w bits; each turn we take b bits to check.

Note that we have no false negatives.

After checking the least significant byte, we have the following :

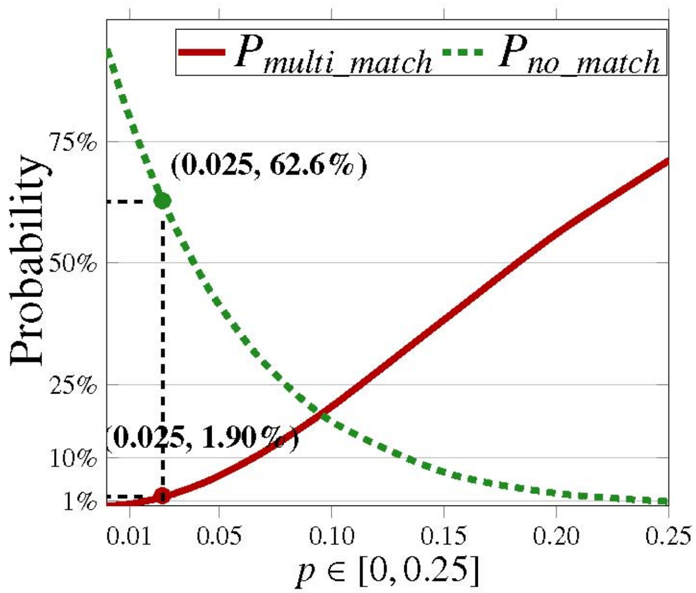
	Positive	Negative
True	p	$\frac{2^w - 2^{w-b}}{2^w - 1}(1-p)$
False	$\frac{2^{w-b} - 1}{2^w - 1}(1-p)$	0

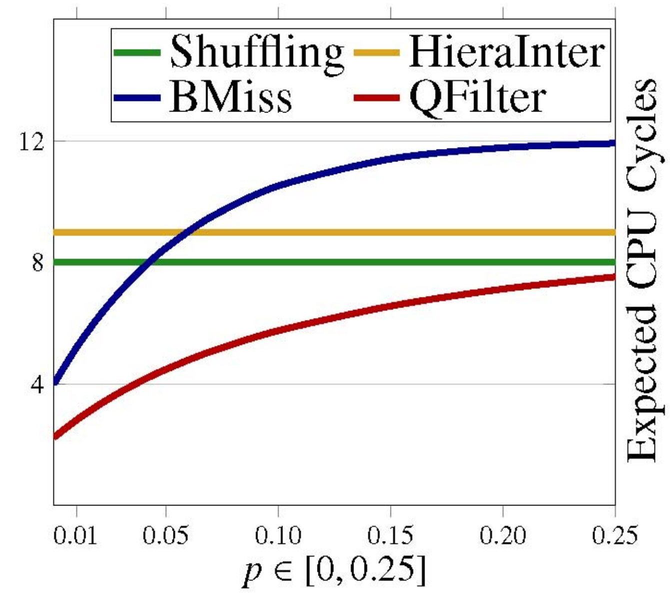
$$P_{\{no-hit\}} = P_{TN}^4;$$
$$P_{\{one-hit\}} = 4 \times P_{TN}^3 \times \left(P_{\{TP\}} + P_{\{FP\}}\right);$$
$$P_{\{multi-hit\}} = 1 - P_{\{no-hit\}} - P_{\{one-hit\}};$$

$$\boldsymbol{P_{\{multi-match\}} = 1 - \left(P_{\{no-hit\}} + P_{\{one-hit\}}\right)^4}$$

Quantitative Analysis

Why "multi-match" rarely happens ?





Fewer CPU cycles than other methods

Typically, in practice, $selectivity < 0.1$,
$i.e. p < 0.025$;
$P_{\{multi-match\}} < 1.90\%$
$P_{\{no-match\}} > 62.6\ \%$

High pruning power of our Qfilter byte-checking approach.

Let us see some experiments

Why "multi-match" rarely happens ?

	TC	MC	SM
$skew\_ratio \leq 1/32$	5.04%	47.72%	25.37%
$skew\_ratio > 1/32$	94.96%	52.28%	74.63%
$selectivity \leq 0.3$	91.75%	95.60%	96.68%
"No-Match" Cases	36.54%	26.07%	43.53%
"One-Match" Cases	58.06%	26.10%	30.41%
"Multi-Match" Cases	0.35%	0.12%	0.69%

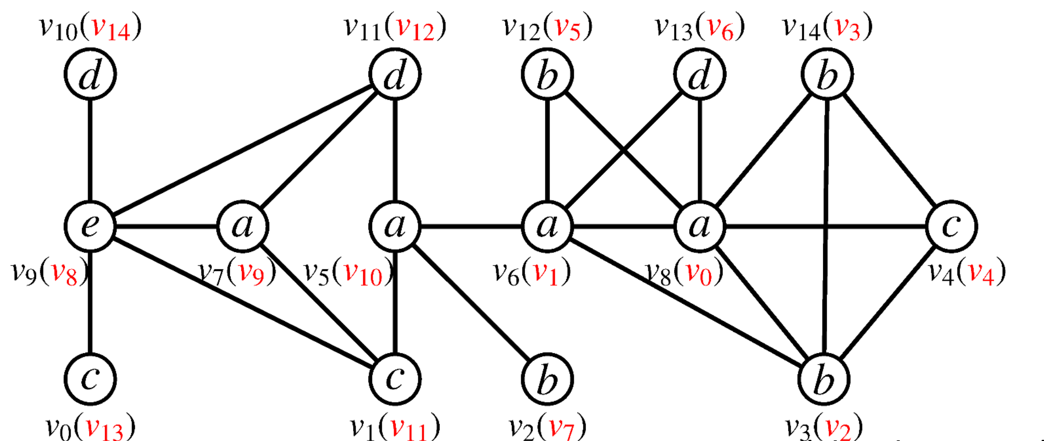
Table 4: Proportions of different cases

Outline

- Motivation
- Related Work
- Our work
  - Data Structure (Base and State Representation)
  - Algorithm (QFilter, SIMD-based)
  - Graph Re-ordering
- Experiments

Graph Reordering

The Node Ordering Matters



(chunk size = 8)

V	$N(V)$	V'	$N(V')$
...	...	...	...
v_3	v_4, v_6, v_8, v_{14}	v_2	v_0, v_1, v_3, v_4
v_4	v_3, v_8, v_{14}	v_4	v_0, v_2, v_3
...	...	...	...
27 state chunks in total		17 state chunks in total	

Graph Reordering

BSR Compactness Score

$$S(G, w, f, \alpha) = \sum_{v_i \in V} \alpha_O(v_i) \cdot \left|\widetilde{N_O}(v_i)\right| + \alpha_I(v_i) \cdot \left|\widetilde{N_I}(v_i)\right|$$

- $w:$ the state chunk size of BSR;
- f: the node ID assignment function, $f.V \rightarrow \{0, 1, \ \dots\ , \ |V| - 1\}$;
- $\left|\widetilde{N_O}(v_i)\right|$(or $\left|\widetilde{N_I}(v_i)\right|$): the number of state chunks of v_i's out-neighbors (or in-neighbors);
- $\alpha_O(v_i)$ (or $\alpha_I(v_i)$): the biased weight to estimate the accessing frequency of v_i's out-neighbors (or in-neighbors).

Graph Reordering

Definition of the Graph Reordering Problem

- Given a graph $G(V, E)$, where each node $v_i \in V$ is assigned with the ID i in advance, a state chunk size w. The ***graph reordering problem*** is to find the node ID assignment function $f. V \rightarrow \{0, 1, \ \dots , \ |V| - 1\}$, which minimizes the compactness score $S(G, w, f, \alpha)$.

Graph Reordering

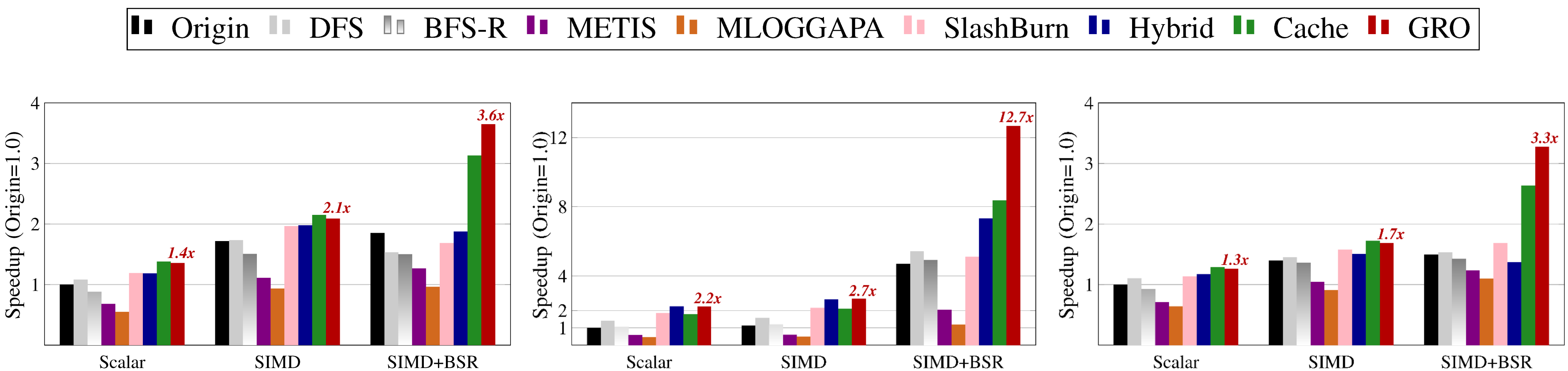
Hardness

- The graph reordering problem is ***NP-complete***.

- we propose an approximate algorithm that can find a better ordering to enhance the intra-chunk parallelism.

Evaluation Results

Average Speedups on 9 Graph Orderings and 3 Graph Algorithms under Different Settings



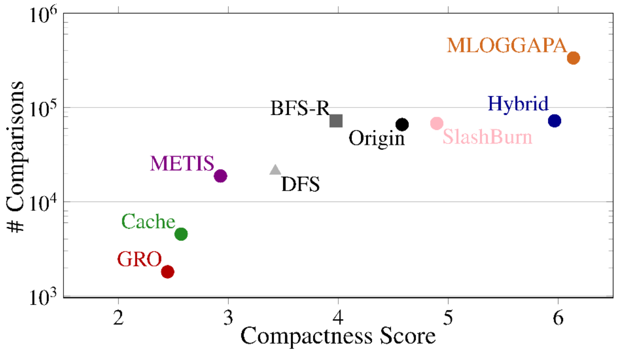
(a) Triangle Counting

(b) Maximal Clique

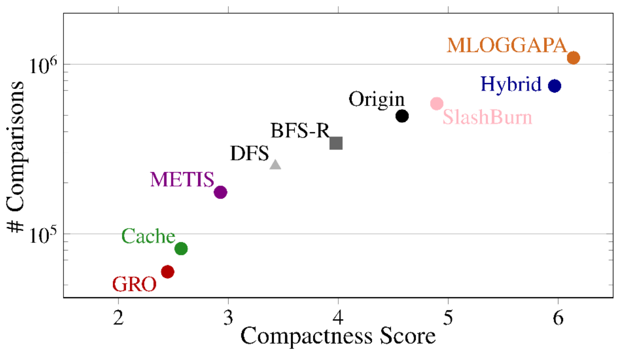
(c) Subgraph Matching

Evaluation Results

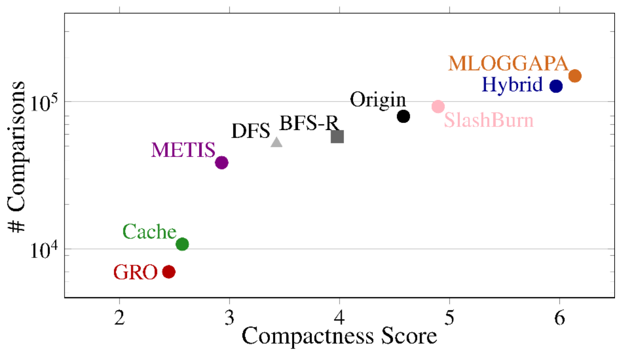
Number of All-pairs Comparisons vs. Compactness Score



(a) Triangle Counting



(b) Maximal Clique

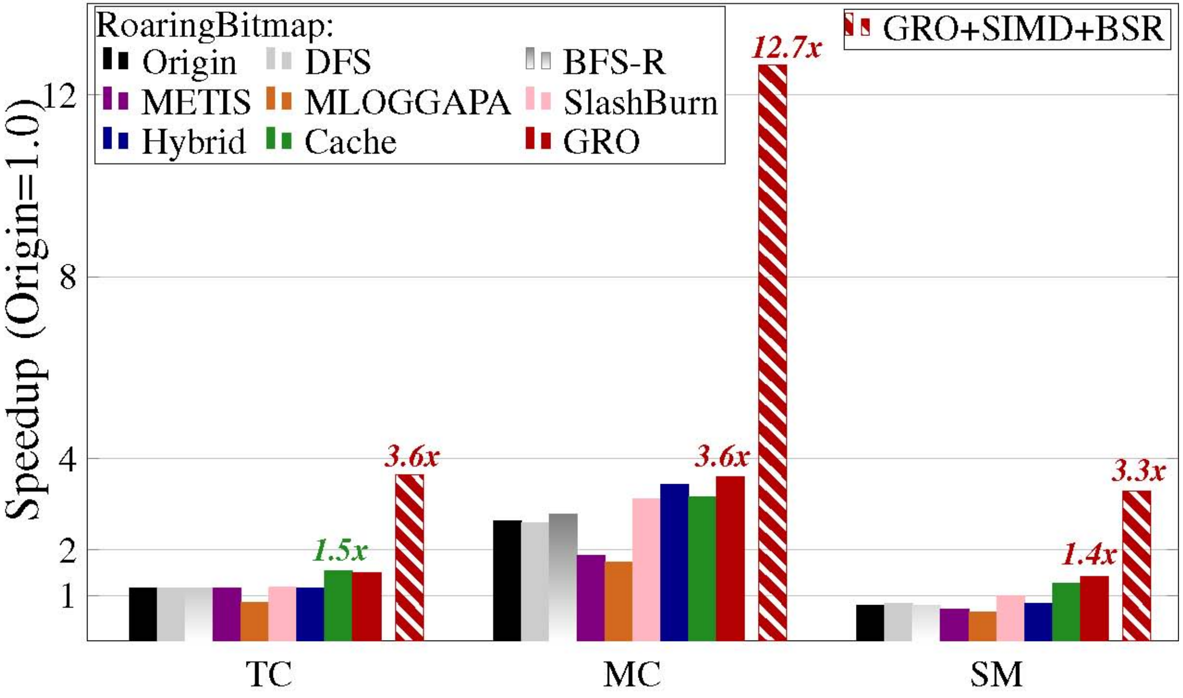


(c) Subgraph Matching

Evaluation Results

Comparing with state-of-the-arts

VS. **Roaring** [13], that is reported as the fastest set intersection against other compression techniques (reported in SIGMOD 2017 experimental study paper [14])



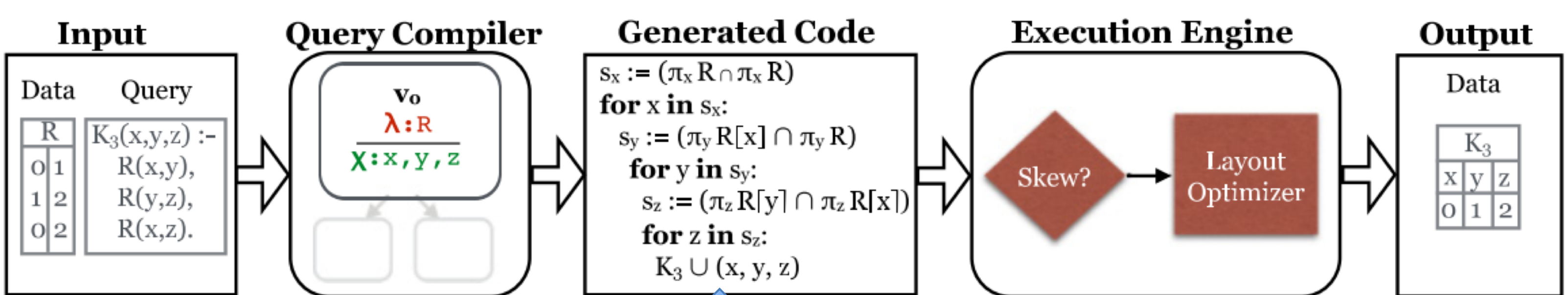
Evaluation Results

Comparing with state-of-the-arts

VS. **EmptyHeaded**, in TODS 2017 [6]

EmptyHeaded:

- A **high-level relational engine** for **graph processing** achieves performance comparable to that of low-level engines.



Using worst-case optimal join algorithm [15, 16]

set-intersection

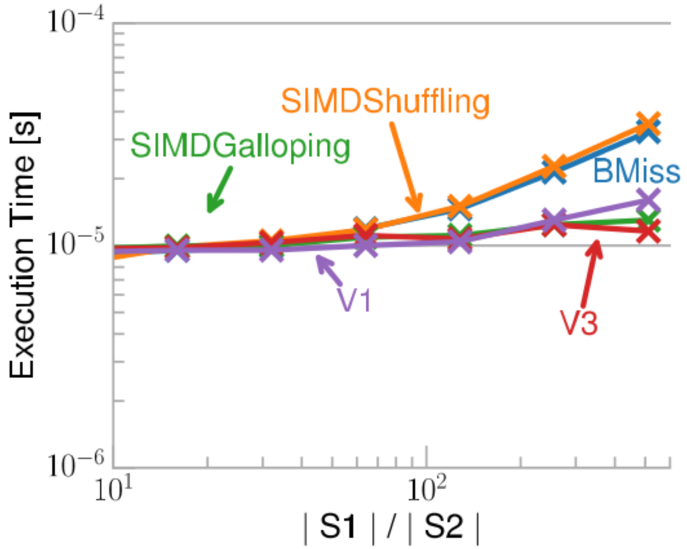
Execution Engine

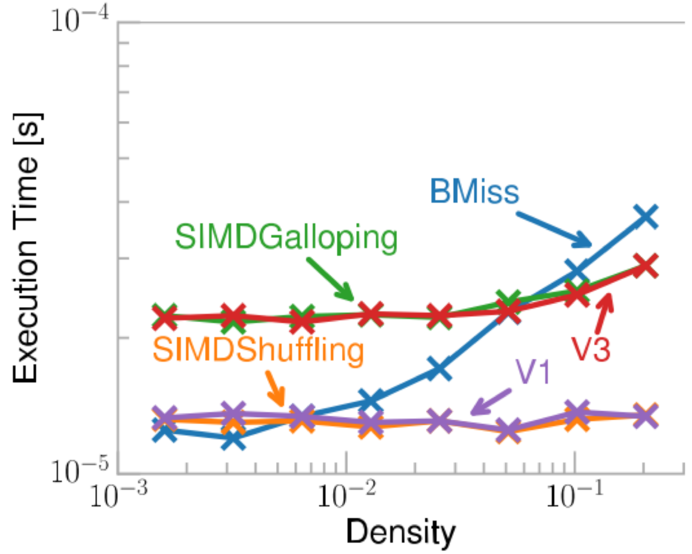
SIMD Set Intersection Algorithms:

- Directly use some off-the-shelf algorithms:
  - SIMDShuffling [11]
  - V1, V3
  - SIMDGalloping [17]
  - Bmiss [18]

Execution Engine

SIMD Set Intersection Algorithms:



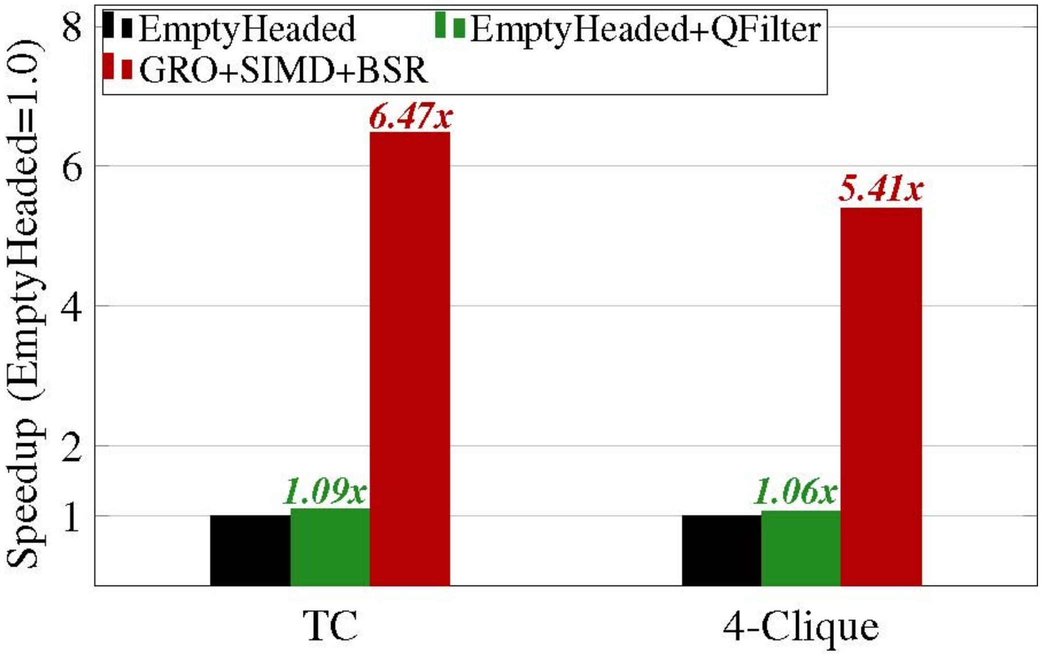


Automatically switch between SIMDShuffling and SIMDGalloping at the run time by considering the ***skew ratio*** ($|S1|/|S2|$)

Evaluation Results

Comparing with state-of-the-arts

VS. **EmptyHeaded**, in TODS 2017 [9]



Conclusions

- BSR Layout is used to represent node ID sets, which is tailored for accelerating set intersection using SIMD instructions
- A byte-checking strategy is proposed in our Qfliter algorithm, with some theoretical analysis.
- We propose a new graph ordering algorithm to find a better graph ordering to save the compactness of BSR representation.
- Qfilter+SIMD+GRO does improve the graph performance greatly (***3-10x** speedup*)

Our codes are here https://github.com/Caesar11/GraphSetIntersection.git

References

- [1] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In SIGKDD. ACM, 672–680.
- [2] Coen Bron and Joep Kerbosch. 1973. Finding all cliques of an undirected graph (algorithm 457). Commun. ACM 16, 9 (1973), 575–577.
- [3] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. TPAMI 26, 10
- (2004), 1367–1372
- [4] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In SIGMOD. ACM, 337–348.
- [5] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, Tianyu Wo: Strong simulation: Capturing topology in graph pattern matching. ACM Trans. Database Syst. 39(1): 4:1-4:46 (2014)
- [6] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. TODS 42, 4 (2017), 20.

References

- [7] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, Dongyan Zhao: gStore: a graph-based SPARQL query engine. VLDB J. 23(4): 565-590 (2014)
- [8] Julian R. Ullmann: An Algorithm for Subgraph Isomorphism.J. ACM 23(1): 31-42 (1976)
- [9] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In SODA (2000)
- [10] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with simd instructions by reducing branch mispredictions. PVLDB 8, 3 (2014), 293–304.
- [11] Ilya Katsov. 2012. Fast intersection of sorted lists using SSE instructions. (2012). https://highlyscalable.wordpress.com/2012/06/05/
- fast-intersection-sorted-lists-sse/
- [12] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. 2011. Fast sorted-Set intersection using SIMD instructions. In ADMS@VLDB. 1–8.

- [13] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2017. Roaring bitmaps: Implementation of an optimized software library. SPE (2017).
- [14] JianguoWang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In SIGMOD. ACM, 993–1008.
- [15] Hung Q. Ngo, Ely Porat, Christopher Ré, Atri Rudra: Worst-case Optimal Join Algorithms. J. ACM 65(3): 16:1-16:40 (2018)
- [16] Hung Q. Ngo, Christopher Ré, Atri Rudra: Skew strikes back: new developments in the theory of join algorithms. SIGMOD Record 42(4): 5-16 (2013)
- [17] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD compression and the intersection of sorted integers. SPE 46, 6 (2016), 723–749.
- [18] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. 2014. Faster set intersection with simd instructions by reducing branch mispredictions. PVLDB 8, 3 (2014), 293–304.

Thanks