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?

\[
\begin{align*}
\text{set}_a & : & 1 & 2 & 3 & 4 & 5 & 7 & 9 & 11 & \cdots \\
\text{set}_b & : & 1 & 3 & 4 & 5 & 7 & 8 & 9 & 10 & \cdots 
\end{align*}
\]
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 $\text{Deg}(v) \leq \text{Deg}(u)$;
- **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**:
  - $I = \text{INTERSECT}(N^+(v), N^+(u))$
  - $\Delta = \Delta \cup \{(v, u) \times I\}$
Motivation

• Maximal Clique Detection

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

$\forall v \in P'$ do:

- $R' = R \cup \{v\}$
- $P' = \text{INTERSECT}(P, N(v))$
- $X' = \text{INTERSECT}(X, N(v))$
- Call $\text{BroKerbosch}(R', P', X')$
- $P = P \setminus \{v\}$
- $X = X \cup \{v\}$

$\text{BroKerbosch}(R, P, X)$:

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

• Subgraph Isomorphism

Neighbor Connection Pruning


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 $(u_1, u_3)$

- $N(v_1) = \{v_3\} \cap C(u_1) = \emptyset$

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

Let us see some experiment results

Profiling of 3 Representative Graph Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Set Inter. Calls</th>
<th>Set Inter. Time</th>
<th>Total Time</th>
<th>Prop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle Counting</td>
<td>21274216</td>
<td>9.9s</td>
<td>10.5s</td>
<td>94.3%</td>
</tr>
<tr>
<td>Maximal Clique</td>
<td>254503699</td>
<td>120.7s</td>
<td>164.1s</td>
<td>73.6%</td>
</tr>
<tr>
<td>Subgraph Matching</td>
<td>120928579</td>
<td>31.5s</td>
<td>54.1s</td>
<td>58.2%</td>
</tr>
</tbody>
</table>

Set Intersection plays an important role!
Motivation

• Why Set-Intersection Important?

Primitive Operations

- Triangle Counting/Listing
- Clique Detection
- Subgraph Matching

Clustering Coefficient

\[ \text{Triangle-based Community Detection (} K - \text{truss}) \]

Maximal Common Subgraph

\[ \text{Dense Subgraph Detection (} K - \text{clique}) \]

SPARQL Query

\[ \text{Social Group Finding} \]

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

- **SIMD Instructions**

  SIMD: Single instruction multiple data.

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

**Diagram:**

```
<table>
<thead>
<tr>
<th>a0</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

_mm_shuffle_epi32(a,b)
```
**Related Work**

- **SIMD Instructions (continued)**

  SIMD: Single instruction multiple data.

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

![Diagram](Image)
Related Work

• Pairwise Set-Intersection

Merge-based Solution

```
Algorithm 1: Merge-based Intersection (non-SIMD)
1  int i = 0, j = 0, size_c = 0;
2  while i < size_a && j < size_b do
3      if set_a[i] == set_b[j] then
4          set_c[size_c++] = set_a[i];
5          i++; j++;
6      elseif set_a[i] < set_b[j] then i++;
7      else j++;
8  return set_c, size_c;
```

# of comparisons:
Best case: 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.

\[ \text{set}_a: \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 7 \\
\end{array} \quad \text{set}_b: \begin{array}{cccccc}
1 & 3 & 4 & 5 & 7 & 8 \\
\end{array} \]

\[ i += 4 \]

Final mask.

\[ \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
\end{array} \]
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 <= size_b
1 i = 0; j = 0; size_c = 0;
2 while i < size_a && j < size_b do
3     sequential search the smallest r (r = 2^0, 2^1, 2^2, ...),
4     such that set_b[j + r] >= set_a[i];
5     binary search the smallest r' in range [r/2, r] such
6     that set_b[j + r'] >= set_a[i];
7     if set_a[i] == set_b[j + r'] then
8         set_c[size_c++] = set_a[i];
9         i++; j = j + r';
10    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

**base values:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>256</th>
</tr>
</thead>
</table>

**state chunks:**

<table>
<thead>
<tr>
<th></th>
<th>0011</th>
<th>1110</th>
<th>0110</th>
<th>1011</th>
</tr>
</thead>
</table>

**bitmap:**

<table>
<thead>
<tr>
<th></th>
<th>0011</th>
<th>0...0</th>
<th>1110</th>
<th>0110</th>
<th>0...0...0</th>
<th>1011</th>
</tr>
</thead>
</table>

\[w = 32\]

**all-zero chunk**

Neighbor Set:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>65</th>
<th>66</th>
<th>67</th>
<th>97</th>
<th>98</th>
<th>8192</th>
<th>8193</th>
<th>8195</th>
</tr>
</thead>
</table>
Our Algorithm-QFilter

- **INPUTS:** two sets in BSR format
  \[(bv_a, sv_a ); (bv_b, sv_b)\]

<table>
<thead>
<tr>
<th>(bv_a)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>259</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sv_a)</td>
<td>...0100</td>
<td>...1001</td>
<td>...0111</td>
<td>...1101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(bv_b)</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sv_b)</td>
<td>...0011</td>
<td>...1110</td>
<td>...0110</td>
<td>...1011</td>
</tr>
</tbody>
</table>

- **OUTPUT:** the intersection set \((bv_c, sv_c)\)

<table>
<thead>
<tr>
<th>(bv_c)</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sv_c)</td>
<td>...0110</td>
</tr>
</tbody>
</table>
Our Algorithm - QFilter

Overview

Load 4 base values

Filter by byte-checking

No match?

Load 4 state chunks

Align and match

Gather results and write back

Advance array pointers

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

2. compare 4 base pairs

1. align chunks

3. bitwise-AND state chunks

Filter vector
Align and match

\[ bv\_mask \]
\[
\begin{array}{cccc}
1.\ldots 1 & 0.\ldots 0 & 1.\ldots 1 & 0.\ldots 0 \\
\end{array}
\]

2. bitwise-AND masks

\[ non\_empty\_mask \]
\[
\begin{array}{cccc}
0.\ldots 0 & 1.\ldots 1 & 1.\ldots 1 & 1.\ldots 1 \\
\end{array}
\]

\[ bv\_a \]
\[
\begin{array}{cccc}
0 & 1 & 2 & 259 \\
\end{array}
\]

3. shift mask to a 4-bit value

\[ sv\_c \]
\[
\begin{array}{cccc}
...0000 & ...1000 & ...0110 & ...0100 \\
\end{array}
\]

4. gather matched base values

\[ 0 0 1 0 \]
\[ rm(4\ bits) \]

4. gather valid state chunks

\[ bv\_c \]
\[
\begin{array}{cccc}
2 & & & \\
\end{array}
\]

\[ sv\_c \]
\[
\begin{array}{cccc}
...0110 & & & \\
\end{array}
\]

Output
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.

\textit{Intra-chunk Parallelism} + \textit{Inter-chunk Parallelism} \implies \textit{More than 10x speedup!}
Quantitative Analysis

Why “multi-match” rarely happens?

A match that includes at least one “multi-hit” is called “multi-match”.

\[ \text{by}_a \]

\[ \text{by}_b \]

\[ \text{filter (16 bits)} \]

\[
\begin{array}{ccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

multi-hit  no-hit  one-hit
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:

<table>
<thead>
<tr>
<th></th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>True</strong></td>
<td>( p )</td>
<td>( \frac{2^w - 2^{w-b}}{2^w - 1} (1 - p) )</td>
</tr>
<tr>
<td><strong>False</strong></td>
<td>( \frac{2^{w-b} - 1}{2^w - 1} (1 - p) )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

\[
P_{\{\text{no-hit}\}} = P_{TN}^4; \\
P_{\{\text{one-hit}\}} = 4 \times P_{TN}^3 \times (P_{TP} + P_{FP}); \\
P_{\{\text{multi-hit}\}} = 1 - P_{\{\text{no-hit}\}} - P_{\{\text{one-hit}\}};
\]

\[
P_{\{\text{multi-match}\}} = 1 - \left( P_{\{\text{no-hit}\}} + P_{\{\text{one-hit}\}} \right)^4
\]

\[\text{4} \]
Quantitative Analysis

Why “multi-match” rarely happens?

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

Fewer CPU cycles than other methods

High pruning power of our Qfilter byte-checking approach.
Let us see some experiments

Why “multi-match” rarely happens?

<table>
<thead>
<tr>
<th>Condition</th>
<th>TC</th>
<th>MC</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>skew_ratio ( \leq 1/32 )</td>
<td>5.04%</td>
<td>47.72%</td>
<td>25.37%</td>
</tr>
<tr>
<td>skew_ratio ( &gt; 1/32 )</td>
<td>94.96%</td>
<td>52.28%</td>
<td>74.63%</td>
</tr>
<tr>
<td>selectivity ( \leq 0.3 )</td>
<td>91.75%</td>
<td>95.60%</td>
<td>96.68%</td>
</tr>
<tr>
<td>“No-Match” Cases</td>
<td>36.54%</td>
<td>26.07%</td>
<td>43.53%</td>
</tr>
<tr>
<td>“One-Match” Cases</td>
<td>58.06%</td>
<td>26.10%</td>
<td>30.41%</td>
</tr>
<tr>
<td>“Multi-Match” Cases</td>
<td>0.35%</td>
<td>0.12%</td>
<td>0.69%</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>V</th>
<th>N(V)</th>
<th>V'</th>
<th>N(V')</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>v₃</td>
<td>v₄, v₆, v₈, v₁₄</td>
<td>v₂</td>
<td>v₀, v₁, v₃, v₄</td>
</tr>
<tr>
<td>v₄</td>
<td>v₃, v₈, v₁₄</td>
<td>v₄</td>
<td>v₀, v₂, v₃</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>27 state chunks in total</td>
<td>17 state chunks in total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph Reordering

BSR Compactness Score

\[ S(G, w, f, \alpha) = \sum_{v_i \in V} \alpha_o(v_i) \cdot |\widetilde{N}_o(v_i)| + \alpha_i(v_i) \cdot |\widetilde{N}_i(v_i)| \]

- \(w\): the state chunk size of BSR;
- \(f\): the node ID assignment function, \(f. V \rightarrow \{0, 1, \ldots, |V| - 1\}\);
- \(|\widetilde{N}_o(v_i)|\) (or \(|\widetilde{N}_i(v_i)|\)): 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).
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 $f: V \rightarrow \{0, 1, \ldots, |V| - 1\}$, which minimizes the compactness score $S(G, w, f, \alpha)$. 

---

**Graph Reordering**
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.
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

References


Thanks