Speedup Graph Processing by Graph Ordering

Hao Wei, Jeffrey Xu Yu, Can Lu, Xuemin Lin

Presentation by Sophia Luo
Problem Being Solved

• CPU cache performance is key to database system efficiency
• Cache miss latency can take >50% of execution time
Problem Being Solved

- NQ: operation to access neighbors of a node in a graph
- BFS: breadth fist search
- DFS: depth first search
- SCC: strongly connected component detection
- SP: shortest path by Bellman-Ford
- PR: PageRank algorithm
- DS: dominating set algorithm
- Kcore: graph decomposition algorithm
- Diam: graph diameter algorithm
Motivation for Problem Being Solved

• Graph algorithms don’t inherently take care of cache miss latency

• Thus, need general approach to enhance graph processing for all graph algorithms that is not specific to any particular algorithm or data structure
Main Result: Gorder
Structure of this Presentation

• Graph ordering
• Graph ordering algorithm
• Priority queue based algorithm
• Priority queue and its operations
• Some results from the evaluation
Definitions

• Directed graph $G = (V,E)$
• $V(G)$: set of nodes
• $E(G)$: set of edges
• $NO(u)$: out-neighbor set of $u$
• $NI(u)$: in-neighbor set of $u$

• $n = |V(G)|$, $m = |E(G)|$
• $dI(u) = |NI(u)|$, $dO(u) = |NO(u)|$
• $d(u) = dI(u) + dO(u)$
1: for each node $v \in N_O(u)$ do
2: the program segment to compute/access $v$
More definitions

• Neighbor relationship: nodes that are directly adjacent each other
• Sibling relationship: let $v_i$ and $v_j$ be in the outneighbor set of $u$. $v_i$ and $v_j$ are siblings

• Sibling relationship is the dominating factor

• Score function
  • $S(u,v) = Ss(u,v) + Sn(u,v)$
• Goal: find a permutation to maximize the sum of $S$ for close node pairs in $G$ that numbers all nodes in $G$ in some ordering
Problem Statement

\[ F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u,v) \]

\[ = \sum_{i=1}^{n} \sum_{j=\max\{1,i-w\}}^{i-1} S(v_i,v_j) \]
Graph Ordering (GO) algorithm

Algorithm 1 GO \((G, w, S(\cdot, \cdot))\)

1: select a node \(v\) as the start node, \(P[1] \leftarrow v\);
2: \(V_R \leftarrow V(G) \setminus \{v\}, i \leftarrow 2\);
3: while \(i \leq n\) do
4: \(v_{\text{max}} \leftarrow \emptyset, k_{\text{max}} \leftarrow -\infty\);
5: for each node \(v \in V_R\) do
6: \(k_v \leftarrow \sum_{j=\max\{1, i-w\}}^{i-1} S(P[j], v)\);
7: if \(k_v > k_{\text{max}}\) then
8: \(v_{\text{max}} \leftarrow v, k_{\text{max}} \leftarrow k_v\);
9: \(P[i] \leftarrow v_{\text{max}}, i \leftarrow i + 1\);
10: \(V_R \leftarrow V_R \setminus \{v_{\text{max}}\}\);
Theorem 3.1: The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.

- Same as the optimal maxTSP-w problem
- $F_w$: score of the optimal solution on G for the maxTSP-w problem
- $F_{go}$: $G$-score of the graph ordering by the GO algorithm

\[
\overline{F}_w = \max \sum_{i=1}^{n-1} \sum_{j>i} s_{ij} x_{ij}
\]

subject to \[
\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2w, i \in [1, n]
\]

$0 \leq x_{ij} \leq 1, \quad i, j \in [1, n]$
Theorem 3.1: The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.

$$F_w \leq \max_{0 \leq x_{ij} \leq 1} \sum_{i=1}^{n-1} \sum_{j>i} s_{ij} x_{ij} + \sum_{i=1}^{n} \alpha_i (2w - \sum_{j>i} x_{ij} - \sum_{j<i} x_{ji})$$

$$= \max_{0 \leq x_{ij} \leq 1} \sum_{i=1}^{n-1} \sum_{j>i} (s_{ij} - \alpha_i - \alpha_j) x_{ij} + 2w \sum_{i=1}^{n} \alpha_i \quad (6)$$
Theorem 3.1: The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.

$$\alpha_i = \sum_{j=\max\{1,i-w+1\}}^{i} s_{j,i+1} \text{ for } i \in [1, n - 1] \text{ and } \alpha_n = 0.$$

$$\alpha_i \geq 0 \text{ and } \sum_{i=1}^{n} \alpha_i = F_{go}$$

$$s_{ij} - \alpha_i \leq 0$$

$$s_{ij} - \alpha_i - \alpha_j \leq 0.$$

$$F_w \leq \overline{F}_w \leq 2w \sum_{i=1}^{n} \alpha_i = 2w \cdot F_{go}$$
Runtime

**Theorem 3.2:** The GO Algorithm 1 is in $O(w \cdot d_{\text{max}} \cdot n^2)$, where $d_{\text{max}}$ denotes the maximum in-degree of the graph $G$. 
Priority Queue based Algorithm (GO-PQ)

- More efficiently select vertex with highest kv score
- In priority queue, Q
  - Key is kv for node v during computation
  - Node vmax with the largest kmax is popped from Q
  - That is, u appears before v if ku > kv regardless of window size
Priority Queue based Algorithm (GO-PQ)

- When the window is sliding, suppose $v_b$ is the node to leave the window and $v_e$ is the node to join the window.
- The algorithm incrementally updates $\text{key}(v)$ in three ways
  - Increase key
  - Decrease key
  - Find the max key
Priority Queue based Algorithm (GO-PQ)

• Increase key
  • When ve is newly added to P, v in Q will increase its key value by 1 if v and ve are considered local

• Decrease key
  • when vb is about to leave the window, v in Q will decrease its key value by 1 if v and vb are considered local

• Find max key
  • Just need to call Q.pop
Priority Queue based Algorithm (GO-PQ)

Algorithm 2 GO-PQ (G, w, S(·, ·))

1: for each node $v \in V(G)$ do
2: insert $v$ into $Q$ such that key($v$) $\leftarrow 0$;
3: select a node $v$ as the start node, $P[1] \leftarrow v$, delete $v$ from $Q$;
4: $i \leftarrow 2$;
5: while $i \leq n$ do
6: $v_e \leftarrow P[i - 1]$;
7: for each node $u \in N_O(v_e)$ do
8: if $u \in Q$ then $Q$.incKey($u$);
9: for each node $u \in N_I(v_e)$ do
10: if $u \in Q$ then $Q$.incKey($u$);
11: for each node $v \in N_O(u)$ do
12: if $v \in Q$ then $Q$.incKey($v$);
13: if $i > w + 1$ then
14: $v_b \leftarrow P[i - w - 1]$;
15: for each node $u \in N_O(v_b)$ do
16: if $u \in Q$ then $Q$.decKey($u$);
17: for each node $u \in N_I(v_b)$ do
18: if $u \in Q$ then $Q$.decKey($u$);
19: for each node $v \in N_O(u)$ do
20: if $v \in Q$ then $Q$.decKey($v$);
21: $v_{max} \leftarrow Q$.pop();
22: $P[i] \leftarrow v_{max}, i \leftarrow i + 1$;
Factors that affect/don’t affect overall GO-PQ

• Window size
  • Same 1/2w approximation as GO algorithm
  • Time complexity unrelated to w

• First node selection:
  • Selecting the node with the largest in-degree impacts overall graph ordering

• Computational cost reduction
  • Adding if statements to avoid calling incKey and decKey on the same node
  • If vb is not in NO(u) then... + If ve not in NO(u) then...

• Dealing with huge nodes
  • Take out a node u if dO(u) >= sqrt(n)
Priority queue and its operations

• Goal:
  • keep time complexity of increase key, decrease key, and pop max to a minimum

• Approach:
  • Implement priority queue as linked list with decrease key values
  • Lazy update strategy to reduce number of adjustments to linked list

• Main idea
  • Only adjust linked list of key of a vertex is changed
  • Let Qh be the head table of the queue
  • Qh keeps points to head and end of the queue
  • Keep a pointer to the node that has the largest key at all times
Priority queue and its operations

• When popping \( v_{\text{max}} \), maintain the true key of a vertex \( v_i \) such that
  • Key of the top node is the same
  • \( \text{Key}(v_i) \leq \text{new key}(v_i) \)
• We also maintain the following conditions

\[
\begin{align*}
\text{update}(\text{top}) &= 0 \\
\text{update}(v_i) &\leq 0 \quad \text{for } v_i \neq \text{top} \\
\overline{\text{key}}(\text{top}) &\geq \overline{\text{key}}(v_i) \\
\overline{\text{key}}(\text{top}) + \text{update}(\text{top}) &\geq \overline{\text{key}}(v_i) + \text{update}(v_i)
\end{align*}
\]
Priority queue and its operations

• Only update the queue in the following 2 cases
  • When update(vi) > 0 after updating vi, we then make update(vi) <= 0 by performing the following
    • Key(vi) = key(vi) + update(vi)
    • Update(vi) = 0
  • When selecting vmax to be popped, we make update(top) = 0
Priority queue and its operations

Algorithm 3 decKey \((v_i)\)

1: \(\text{update}(v_i) \leftarrow \text{update}(v_i) - 1;\)
Algorithm 4 incKey \((v_i)\)

1: update\((v_i)\) $\leftarrow$ update\((v_i)\) + 1;
2: if update\((v_i)\) > 0 then
3: update\((v_i)\) $\leftarrow$ 0, \(x \leftarrow \text{key}(v_i)\), \(\text{key}(v_i) \leftarrow \text{key}(v_i) + 1\);
4: delete \(v_i\) from \(\mathcal{Q}\);
5: insert \(v_i\) into \(\mathcal{Q}\) in the position just before head\([x]\);
6: update the head \(\mathcal{Q}_h\) array accordingly;
7: if \(\text{key}(v_i) > \text{key}(\text{top})\) then
8: top $\leftarrow$ \(v_i\);

Priority queue and its operations
Priority queue and its operations

Algorithm 5 pop ()

1: while update(top) < 0 do
2: \( v_t \leftarrow \text{top} \);
3: \( \text{key}(v_t) \leftarrow \text{key}(v_t) + \text{update}(v_t) \);
4: \( \text{update}(v_t) \leftarrow 0 \);
5: if \( \text{key}(\text{top}) \leq \text{key}(\text{next}(\text{top})) \) then
6: \( \) adjust the position of \( v_t \) and insert \( v_t \) just after \( u \) in \( Q \), such that
7: \( \text{key}(u) \geq \text{key}(\text{top}) \) and \( \text{key}(\text{next}(u)) < \text{key}(\text{top}) \);
8: \( \text{top} \leftarrow \text{next}(\text{top}) \);
9: update the head array;
10: \( v_t \leftarrow \text{top} \);
11: remove the node pointed by top from \( Q \) and update \( \text{top} \leftarrow \text{next}(\text{top}) \);
12: return \( v_t \);
Some results from the evaluation

<table>
<thead>
<tr>
<th>Order</th>
<th>L1-ref</th>
<th>L1-mr</th>
<th>L3-ref</th>
<th>L3-r</th>
<th>Cache-mr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>11,109M</td>
<td>52.1%</td>
<td>2,195M</td>
<td>19.7%</td>
<td>5.1%</td>
</tr>
<tr>
<td>MINLA</td>
<td>11,110M</td>
<td>58.1%</td>
<td>2,121M</td>
<td>19.0%</td>
<td>4.5%</td>
</tr>
<tr>
<td>MLOGA</td>
<td>11,119M</td>
<td>53.1%</td>
<td>1,685M</td>
<td>15.1%</td>
<td>4.1%</td>
</tr>
<tr>
<td>RCM</td>
<td>11,102M</td>
<td>49.8%</td>
<td>1,834M</td>
<td>16.5%</td>
<td>4.1%</td>
</tr>
<tr>
<td>DegSort</td>
<td>11,121M</td>
<td>58.3%</td>
<td>2,597M</td>
<td>23.3%</td>
<td>5.3%</td>
</tr>
<tr>
<td>CHDFS</td>
<td>11,107M</td>
<td>49.9%</td>
<td>1,850M</td>
<td>16.7%</td>
<td>4.4%</td>
</tr>
<tr>
<td>SlashBurn</td>
<td>11,096M</td>
<td>55.0%</td>
<td>2,466M</td>
<td>22.2%</td>
<td>4.3%</td>
</tr>
<tr>
<td>LDG</td>
<td>11,112M</td>
<td>52.9%</td>
<td>2,256M</td>
<td>20.3%</td>
<td>5.4%</td>
</tr>
<tr>
<td>METIS</td>
<td>11,105M</td>
<td>50.3%</td>
<td>2,235M</td>
<td>20.1%</td>
<td>5.2%</td>
</tr>
<tr>
<td>Gorder</td>
<td>11,101M</td>
<td>37.9%</td>
<td>1,280M</td>
<td>11.5%</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Table 3: Cache Statistics by PR over Flickr (M = Millions)

<table>
<thead>
<tr>
<th>Order</th>
<th>L1-ref</th>
<th>L1-mr</th>
<th>L3-ref</th>
<th>L3-r</th>
<th>Cache-mr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>623.9B</td>
<td>58.4%</td>
<td>180.0B</td>
<td>28.8%</td>
<td>18.6%</td>
</tr>
<tr>
<td>MINLA</td>
<td>628.8B</td>
<td>62.5%</td>
<td>196.6B</td>
<td>31.2%</td>
<td>14.8%</td>
</tr>
<tr>
<td>MLOGA</td>
<td>620.0B</td>
<td>62.1%</td>
<td>189.6B</td>
<td>30.5%</td>
<td>14.3%</td>
</tr>
<tr>
<td>RCM</td>
<td>628.9B</td>
<td>44.9%</td>
<td>103.8B</td>
<td>16.5%</td>
<td>10.2%</td>
</tr>
<tr>
<td>DegSort</td>
<td>632.2B</td>
<td>55.1%</td>
<td>149.5B</td>
<td>23.6%</td>
<td>15.9%</td>
</tr>
<tr>
<td>CHDFS</td>
<td>630.3B</td>
<td>38.0%</td>
<td>101.2B</td>
<td>16.1%</td>
<td>10.9%</td>
</tr>
<tr>
<td>SlashBurn</td>
<td>628.8B</td>
<td>44.5%</td>
<td>121.0B</td>
<td>19.3%</td>
<td>13.7%</td>
</tr>
<tr>
<td>LDG</td>
<td>637.9B</td>
<td>58.4%</td>
<td>186.2B</td>
<td>29.2%</td>
<td>18.6%</td>
</tr>
<tr>
<td>Gorder</td>
<td>620.3B</td>
<td>31.5%</td>
<td>79.5B</td>
<td>12.8%</td>
<td>8.2%</td>
</tr>
</tbody>
</table>

Table 4: Cache Statistics by PR over sd1-arc (B = Billions)
Some results from the evaluation

<table>
<thead>
<tr>
<th>Order</th>
<th>NQ</th>
<th>BFS</th>
<th>DFS</th>
<th>SCC</th>
<th>SP</th>
<th>PR</th>
<th>DS</th>
<th>Kcore</th>
<th>Diam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>50.8</td>
<td>15.3</td>
<td>5.4</td>
<td>7.8</td>
<td>21.5</td>
<td>52.1</td>
<td>21.9</td>
<td>20.8</td>
<td>14.9</td>
</tr>
<tr>
<td>MINLA</td>
<td>51.8</td>
<td>18.0</td>
<td>5.5</td>
<td>8.1</td>
<td>24.6</td>
<td>58.1</td>
<td>22.1</td>
<td>21.5</td>
<td>17.9</td>
</tr>
<tr>
<td>MLOGA</td>
<td>41.7</td>
<td>16.3</td>
<td>5.1</td>
<td>7.2</td>
<td>21.9</td>
<td>53.1</td>
<td>21.1</td>
<td>20.6</td>
<td>16.4</td>
</tr>
<tr>
<td>RCM</td>
<td>49.1</td>
<td>12.1</td>
<td>4.6</td>
<td>6.6</td>
<td>15.9</td>
<td>49.7</td>
<td>20.3</td>
<td>20.2</td>
<td>12.4</td>
</tr>
<tr>
<td>DegSort</td>
<td>45.7</td>
<td>16.7</td>
<td>4.8</td>
<td>7.0</td>
<td>24.9</td>
<td>58.3</td>
<td>21.4</td>
<td>18.6</td>
<td>17.0</td>
</tr>
<tr>
<td>CHDFS</td>
<td>42.1</td>
<td>12.3</td>
<td>4.1</td>
<td>5.8</td>
<td>18.5</td>
<td>49.9</td>
<td>21.1</td>
<td>20.6</td>
<td>12.9</td>
</tr>
<tr>
<td>SlashBurn</td>
<td>46.2</td>
<td>16.0</td>
<td>4.5</td>
<td>6.2</td>
<td>22.1</td>
<td>55.0</td>
<td>20.7</td>
<td>21.3</td>
<td>15.8</td>
</tr>
<tr>
<td>LDG</td>
<td>50.7</td>
<td>15.9</td>
<td>5.8</td>
<td>8.2</td>
<td>21.8</td>
<td>52.9</td>
<td>22.4</td>
<td>21.2</td>
<td>14.9</td>
</tr>
<tr>
<td>METIS</td>
<td>63.0</td>
<td>18.2</td>
<td>7.7</td>
<td>10.1</td>
<td>20.8</td>
<td>50.3</td>
<td>23.0</td>
<td>21.7</td>
<td>16.7</td>
</tr>
<tr>
<td>Gorder</td>
<td>35.4</td>
<td>11.1</td>
<td>3.6</td>
<td>5.2</td>
<td>12.8</td>
<td>37.9</td>
<td>18.7</td>
<td>18.1</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Table 6: L1 Cache Miss Ratio on Flickr (in percentage %)

<table>
<thead>
<tr>
<th>Order</th>
<th>NQ</th>
<th>BFS</th>
<th>DFS</th>
<th>SCC</th>
<th>SP</th>
<th>PR</th>
<th>DS</th>
<th>Kcore</th>
<th>Diam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>76.5</td>
<td>20.0</td>
<td>9.4</td>
<td>13.0</td>
<td>17.5</td>
<td>58.4</td>
<td>21.7</td>
<td>20.0</td>
<td>17.5</td>
</tr>
<tr>
<td>MINLA</td>
<td>76.0</td>
<td>22.7</td>
<td>10.2</td>
<td>12.8</td>
<td>17.5</td>
<td>62.5</td>
<td>21.3</td>
<td>20.5</td>
<td>18.3</td>
</tr>
<tr>
<td>MLOGA</td>
<td>76.0</td>
<td>21.7</td>
<td>9.4</td>
<td>12.3</td>
<td>19.8</td>
<td>62.1</td>
<td>21.8</td>
<td>20.6</td>
<td>18.5</td>
</tr>
<tr>
<td>RCM</td>
<td>61.6</td>
<td>14.4</td>
<td>7.5</td>
<td>8.7</td>
<td>8.9</td>
<td>44.9</td>
<td>18.2</td>
<td>17.5</td>
<td>11.7</td>
</tr>
<tr>
<td>DegSort</td>
<td>59.3</td>
<td>18.7</td>
<td>8.0</td>
<td>12.1</td>
<td>16.6</td>
<td>55.1</td>
<td>21.9</td>
<td>16.9</td>
<td>15.5</td>
</tr>
<tr>
<td>CHDFS</td>
<td>50.0</td>
<td>14.2</td>
<td>5.1</td>
<td>8.3</td>
<td>13.2</td>
<td>38.0</td>
<td>18.4</td>
<td>16.1</td>
<td>10.4</td>
</tr>
<tr>
<td>SlashBurn</td>
<td>56.6</td>
<td>16.8</td>
<td>6.7</td>
<td>9.3</td>
<td>10.2</td>
<td>44.5</td>
<td>18.9</td>
<td>16.8</td>
<td>13.5</td>
</tr>
<tr>
<td>LDG</td>
<td>74.7</td>
<td>22.7</td>
<td>10.0</td>
<td>13.6</td>
<td>18.7</td>
<td>58.4</td>
<td>22.0</td>
<td>20.3</td>
<td>17.9</td>
</tr>
<tr>
<td>Gorder</td>
<td>40.0</td>
<td>12.1</td>
<td>4.6</td>
<td>7.2</td>
<td>10.8</td>
<td>31.5</td>
<td>16.9</td>
<td>14.5</td>
<td>9.5</td>
</tr>
</tbody>
</table>

Table 7: L1 Cache Miss Ratio on sd1-arc (in percentage %)
Strengths and Weaknesses

• Strengths
  • Well-organized
  • Thorough algorithm description
  • Comprehensive evaluation strategy

• Weakness
  • Too much time spent on describe sub-optimal GO algorithm
  • Redundant in some places of the text, especially in the early sections of the paper
Discussion questions

• The basic algorithm is bounded by an approximation that depends on the window size $w$. What are some ways we can find the optimal $w$?
• Are there any cases that GO performs worse than other graph orderings?
• What are some other methods of reducing CPU cache miss ratios?