Speedup Graph Processing by Graph Ordering

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

Presented by: Bishesh Khadka
MIT 6.886 - Graph Analytics
Motivation

● Graphs are important
● CPU cache performance is key issue in efficiency in DBS
(a) The original order

(b) Gorder
Motivation

- Graphs are important
- CPU cache performance is key issue in efficiency in DBS
  - Cache stalls take a large proportion of time
- Can better locality via ordering help?
  - Store frequently accessed nodes close in memory
- How can a generalized solution reduce cache stall rates?
Graph Access Patterns

- Most common access pattern:
  1. for each node $v \in N_O(u)$ do
  2. the program segment to compute/access $v$

- Locality between neighboring nodes are important

- Locality among sibling nodes even more important

- Let “closeness” heuristic be $S(u, v) = S_s(u, v) + S_n(u, v)$
Graph Partitioning isn’t sufficient

- Real graphs have poor edge cuts because of power law degree distributions
  - Nodes with high degrees
- Fixed sized caches
  - What partition size?
- Data alignment

Assume a cache line holds 3

Figure 3: By Graph Partitioning
Graph Ordering does better

- Optimal permutation $\phi$ among
- Frequently accessed nodes within window $w$
- Reorder graph id’s
- Sort in all adj. lists

*Figure 4: By Graph Ordering*
Graph Ordering does better cont’d

- Locality is continuous for any sliding window
  - Assumes little of data alignment
- Considers sibling and neighbor locality

![Diagram](image)

**Figure 4: By Graph Ordering**
Problem Statement

- Find the optimal permutation $\phi$ that maximizes aggregate locality defined by $F(\phi)$ for all sliding windows of size $w$.

\[
F(\phi) = \sum_{0<\phi(v)-\phi(u)\leq w} S(u, v) \quad (2)
\]

\[
= \sum_{i=1}^{n} \sum_{j=\max\{1, i-w\}}^{i-1} S(v_i, v_j) \quad (3)
\]
Key Contributions

- Locality scoring function
- Prove NP-hardness of graph ordering
  - Graph ordering is a variant of maximum TSP
    - Maximize reward for sliding windows $w$
- Propose two algorithms for graph ordering
  - GO
  - GO-PQ
- Evaluation of improved efficiency
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: \textbf{while} \(i \leq n\) \textbf{do}
4: \(v_{max} \leftarrow \emptyset, k_{max} \leftarrow -\infty\);
5: \textbf{for each node} \(v \in V_R\) \textbf{do}
6: \(k_v \leftarrow \sum_{j=\text{\text{max}}\{1,i-w\}}^{i-1} S(P[j], v)\);
7: \textbf{if} \(k_v > k_{max}\) \textbf{then}
8: \(v_{max} \leftarrow v, k_{max} \leftarrow k_v\);
9: \(P[i] \leftarrow v_{max}, i \leftarrow i + 1\);
10: \(V_R \leftarrow V_R \setminus \{v_{max}\}\);
GO algorithm

- Greedily maximize $F(\phi)$ by inserting $v$ with the largest aggregate $S()$ in previous window $w$
- Randomly select starting node
- Redundantly computes eq. 4 $w$-times for same pair ($v_j, v$) while in same window
- Scans through even nodes w/o neighbor/sibling relationships

\[
k_v = \sum_{j=\max\{1, i-w\}}^{i-1} S(v_j, v)
\]

<table>
<thead>
<tr>
<th></th>
<th>$w = 3$</th>
<th></th>
<th>$w = 5$</th>
<th></th>
<th>$w = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$F_{go}$</td>
<td>$F_w$</td>
<td>$F_{go}$</td>
<td>$F_w$</td>
<td>$F_{go}$</td>
</tr>
<tr>
<td>Facebook</td>
<td>149,073</td>
<td>172,526</td>
<td>231,710</td>
<td>275,974</td>
<td>308,091</td>
</tr>
<tr>
<td>AirTraffic</td>
<td>2,420</td>
<td>3,468</td>
<td>2,993</td>
<td>4,697</td>
<td>3,465</td>
</tr>
</tbody>
</table>

*Table 1: $F_{go}$ and $F_w$*
Algorithm 2 GO-PQ ($G, w, S(\cdot, \cdot))$

1: for each node $v \in V(G)$ do
2:     insert $v$ into $Q$ such that $\text{key}(v) \leftarrow 0$;
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:     $v_{\text{max}} \leftarrow Q$.pop();
22:     $P[i] \leftarrow v_{\text{max}}, i \leftarrow i + 1$;
GO-PQ algorithm

- Similar to GO
- Uses PQ to maintain sliding window
- $Q[v] = k_v$ as computed by Eq. 4
- When $V_e$ joins, $v$ in $W$ increment their keys if there is a neighbor and/or sibling relation
- $V_b$ leaves, $v$ w/ relations decrements key
- Pops largest key as $V_b

$$k_v = \sum_{j=\max\{1, i-w\}}^{i-1} S(v_j, v)$$
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 $G$.

Theorem 3.3: The time complexity of the GO-PQ algorithm is $O(\mu \cdot \sum_{u \in V} (d_0(u))^2 + n \cdot \phi)$, where $\mu$ denotes the time complexity for the updates (incKey(·) and decKey(·)) and $\phi$ denotes the time complexity for finding the max node (pop(·)).
Evaluation

(a) $F(\cdot)$ by Different Orderings
### 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>LGD</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)**
# 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>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>20.7</td>
<td>62.5</td>
<td>21.8</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><strong>8.9</strong></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><strong>40.0</strong></td>
<td><strong>12.1</strong></td>
<td><strong>4.6</strong></td>
<td><strong>7.2</strong></td>
<td>10.8</td>
<td><strong>31.5</strong></td>
<td><strong>16.9</strong></td>
<td><strong>14.5</strong></td>
<td><strong>9.5</strong></td>
</tr>
</tbody>
</table>

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

- Applying Gorder to distributed graph systems is complicated b/c unclear how graph partitioning happens.
Conclusion

● CPU stalling is an important barrier to efficiency
● This paper presents a generalized optimization for graph algorithms with the common access pattern
  ○ 1: for each node $v \in N_O(u)$ do
  2: the program segment to compute/access $v$
●
References

- Hao Wei, Jeffrey Xu Yu, Can Lu, Xuemin Lin
  Speedup Graph Processing by Graph Ordering