Speedup Graph Processing by Graph Ordering

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

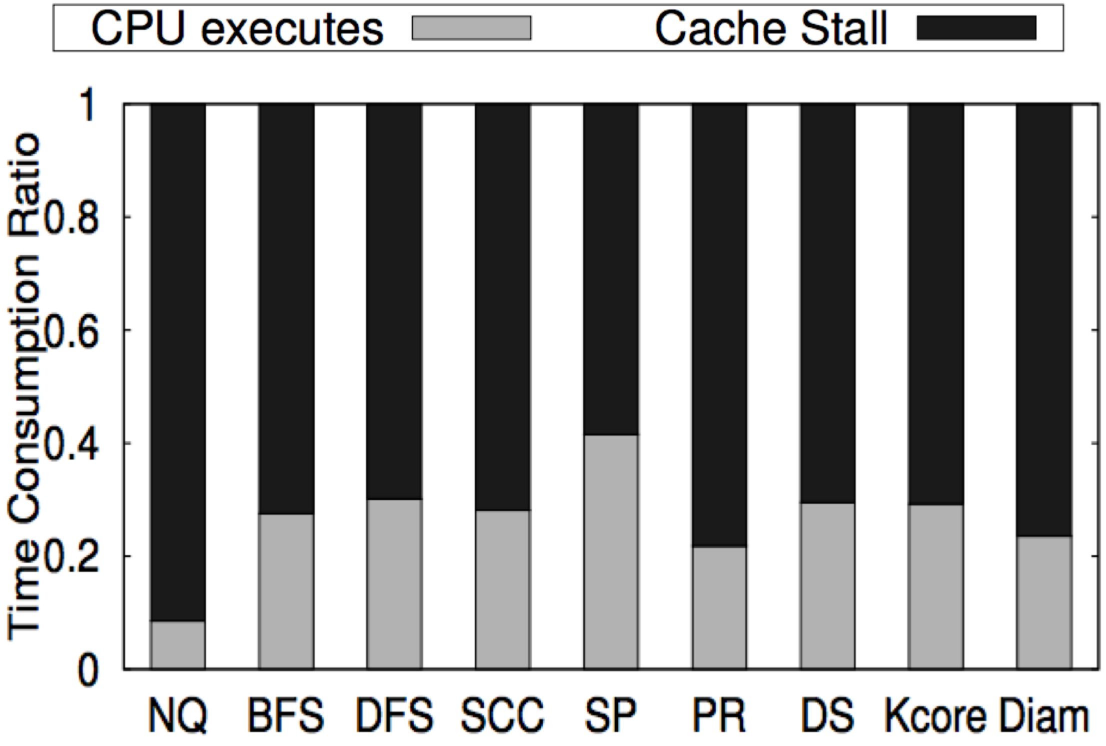
Presentation by Sophia Luo

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

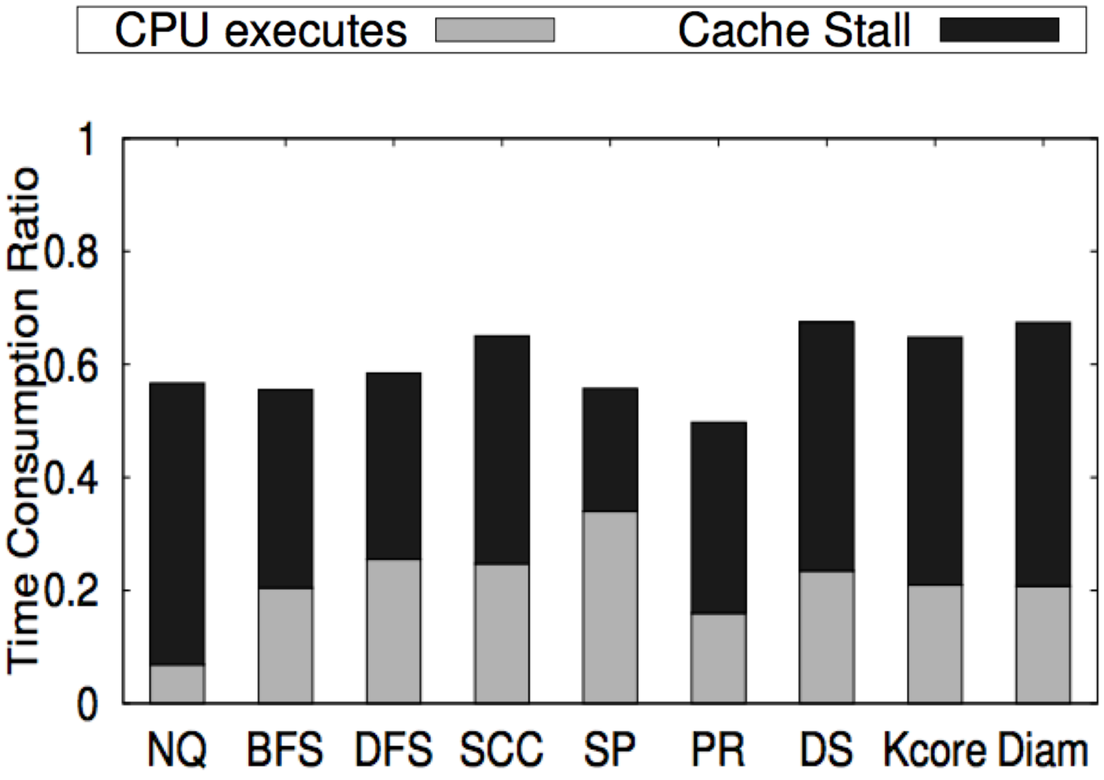


(a) The original order

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



(b) 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

- $n = |V(G)|$, $m = |E(G)|$
- $dI(u) = |NI(u)|$, $dO(u) = |NO(u)|$
- $d(u) = dI(u) + dO(u)$

Context

for each node $v \in N_O(u)$ **do**

    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

$$
\begin{aligned}
F(\phi) &= \sum_{0<\phi(v)-\phi(u)\leq w} S(u,v) \\
&= \sum_{i=1}^{n} \sum_{j=\max\{1,i-w\}}^{i-1} S(v_i, v_j)
\end{aligned}
$$

Graph Ordering (GO) algorithm

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

select a node v as the start node, $P[1] \leftarrow v$;
$V_R \leftarrow V(G) \setminus \{v\}$, $i \leftarrow 2$;
while $i \leq n$ **do**
  $v_{max} \leftarrow \emptyset$, $k_{max} \leftarrow -\infty$;
  for each node $v \in V_R$ **do**
    $k_v \leftarrow \sum_{j=\max\{1,i-w\}}^{i-1} S(P[j], v)$;
    if $k_v > k_{max}$ **then**
      $v_{max} \leftarrow v$, $k_{max} \leftarrow k_v$;
  $P[i] \leftarrow v_{max}$, $i \leftarrow i + 1$;
  $V_R \leftarrow V_R \setminus \{v_{max}\}$;

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
- Fw: score of the optimal solution on G for the maxTSP-w problem
- Fgo: Gscore of the graph ordering by the GO algorithm

$$
\begin{aligned}
\overline{F}_w = \quad & \text{maximize} && \sum_{i=1}^{n-1} \sum_{j>i} s_{ij} x_{ij} \\
& \text{subject to} && \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = 2w, i \in [1, n] \\
& && 0 \le x_{ij} \le 1, \qquad\qquad i, j \in [1, n]
\end{aligned}
$$

Theorem 3.1: *The algorithm GO gives $\frac{1}{2w}$-approximation for maximizing $F(\phi)$ to determine the optimal graph ordering.*

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

$$= \max_{0 \leq x_{ij} \leq 1} \sum_{i=1}^{n-1} \sum_{j>i} (s_{ij} - \alpha_i - \alpha_j) x_{ij} + 2w \sum_{i=1}^{n} \alpha_i \qquad (6)$$

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

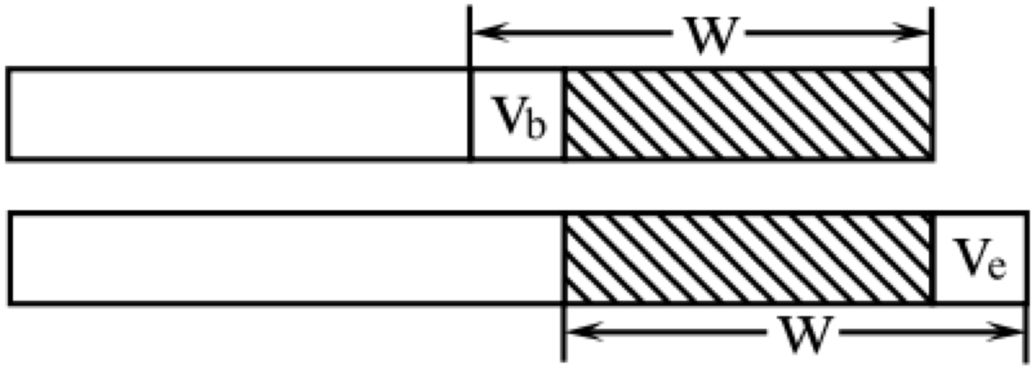
Theorem 3.2: *The GO Algorithm 1 is in* $O(w \cdot d_{max} \cdot n^2)$, *where* d_{max} *denotes the maximum in-degree of the graph G.*

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 vb is the node to leave the window and ve is the node to join the window.
- The algorithm incremently updates 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* $(G, w, S(\cdot,\cdot))$

for each node $v \in V(G)$ **do**
  insert v into $\mathcal{Q}$ such that key$(v) \leftarrow 0$;
select a node v as the start node, $P[1] \leftarrow v$, delete v from $\mathcal{Q}$;
$i \leftarrow 2$;
while $i \leq n$ **do**
  $v_e \leftarrow P[i-1]$;
  for each node $u \in N_O(v_e)$ **do**
    if $u \in \mathcal{Q}$ **then** $\mathcal{Q}$.incKey(u);
  for each node $u \in N_I(v_e)$ **do**
    if $u \in \mathcal{Q}$ **then** $\mathcal{Q}$.incKey(u);
    for each node $v \in N_O(u)$ **do**
      if $v \in \mathcal{Q}$ **then** $\mathcal{Q}$.incKey(v);
  if $i > w + 1$ **then**
    $v_b \leftarrow P[i-w-1]$;
    for each node $u \in N_O(v_b)$ **do**
      if $u \in \mathcal{Q}$ **then** $\mathcal{Q}$.decKey(u);
    for each node $u \in N_I(v_b)$ **do**
      if $u \in \mathcal{Q}$ **then** $\mathcal{Q}$.decKey(u);
      for each node $v \in N_O(u)$ **do**
        if $v \in \mathcal{Q}$ **then** $\mathcal{Q}$.decKey(v);
  $v_{max} \leftarrow \mathcal{Q}$.pop();
  $P[i] \leftarrow v_{max}, i \leftarrow i + 1$;

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 vmax, maintain the true key of a vertex v_i such that
  - Key of the top node is the same
  - Key(v_i) <= new key(v_i)
- We also maintain the following conditions

$$\begin{aligned} \mathsf{update}(\mathsf{top}) &= 0 \\ \mathsf{update}(v_i) &\leq 0 \quad \text{for } v_i \neq \mathsf{top} \\ \overline{\mathsf{key}}(\mathsf{top}) &\geq \overline{\mathsf{key}}(v_i) \\ \overline{\mathsf{key}}(\mathsf{top}) + \mathsf{update}(\mathsf{top}) &\geq \overline{\mathsf{key}}(v_i) + \mathsf{update}(v_i) \end{aligned} \tag{8}$$

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: $\mathsf{update}(v_i) \leftarrow \mathsf{update}(v_i) - 1;$

Priority queue and its operations

Algorithm 4 incKey (v_i)

update$(v_i) \leftarrow$ update$(v_i) + 1$;
if update$(v_i) > 0$ **then**
    update$(v_i) \leftarrow 0$, $x \leftarrow \overline{\mathsf{key}}(v_i)$, $\overline{\mathsf{key}}(v_i) \leftarrow \overline{\mathsf{key}}(v_i) + 1$;
    delete v_i from $\mathcal{Q}$;
    insert v_i into $\mathcal{Q}$ in the position just before head$[x]$;
    update the head $\mathcal{Q}_h$ array accordingly;
    if $\overline{\mathsf{key}}(v_i) > \overline{\mathsf{key}}(\mathsf{top})$ **then**
        top $\leftarrow v_i$;

Priority queue and its operations

Algorithm 5 pop ()

while $\mathsf{update}(\mathsf{top}) < 0$ **do**
    $v_t \leftarrow \mathsf{top}$;
    $\overline{\mathsf{key}}(v_t) \leftarrow \overline{\mathsf{key}}(v_t) + \mathsf{update}(v_t)$;
    $\mathsf{update}(v_t) \leftarrow 0$;
    if $\overline{\mathsf{key}}(\mathsf{top}) \leq \overline{\mathsf{key}}(\mathsf{next}(\mathsf{top}))$ **then**
        adjust the position of v_t and insert v_t just after u in $\mathcal{Q}$, such that $\overline{\mathsf{key}}(u) \geq \overline{\mathsf{key}}(\mathsf{top})$ and $\overline{\mathsf{key}}(\mathsf{next}(u)) < \overline{\mathsf{key}}(\mathsf{top})$;
        $\mathsf{top} \leftarrow \mathsf{next}(\mathsf{top})$;
    update the head array;
$v_t \leftarrow \mathsf{top}$;
remove the node pointed by top from $\mathcal{Q}$ and update $\mathsf{top} \leftarrow \mathsf{next}(\mathsf{top})$;
return v_t;

Some results from the evaluation

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

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

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

Table 4: Cache Statistics by *PR* over sd1-arc (B = Billions)

Some results from the evaluation

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

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

Order	*NQ*	*BFS*	*DFS*	*SCC*	*SP*	*PR*	*DS*	*Kcore*	*Diam*
Original	76.5	20.0	9.4	13.0	17.5	58.4	21.7	20.0	17.5
MINLA	76.0	22.7	10.2	12.8	20.7	62.5	21.8	20.5	18.3
MLOGA	76.0	21.7	9.4	12.3	19.8	62.1	21.8	20.6	18.5
RCM	61.6	14.4	7.5	8.7	**8.9**	44.9	18.2	17.5	11.7
DegSort	59.3	18.7	8.0	12.1	16.6	55.1	21.9	16.9	15.5
CHDFS	50.0	14.2	5.1	8.3	13.2	38.0	18.4	16.1	10.4
SlashBurn	56.6	16.8	6.7	9.3	10.2	44.5	18.9	16.8	13.5
LDG	74.7	22.7	10.0	13.6	18.7	58.4	22.0	20.3	17.9
Gorder	**40.0**	**12.1**	**4.6**	**7.2**	10.8	**31.5**	**16.9**	**14.5**	**9.5**

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?