Front.Comput.Sci.
DOI

LETTER

Online Resource 1 of "A Lock-free Approach to Parallelizing Personalized PageRank Computations on GPU"

Zhigang Wang[1], Ning Wang (✉)[1], Jie Nie[1], Zhiqiang Wei[1], Yu Gu[2], Ge Yu[2]

1 Faculty of Information Science and Engineering, Ocean University of China, Qingdao 266100, China
2 School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

This supplementary file gives the detailed description about switching timing estimation in our *Hybrid Framework* in Section 4. First of all, we should state that in a GPU device, threads work following the single-instruction-multiple-data (SIMD) design. Every w threads are grouped into a *warp*—the basic instruction execution unit, and every b *warp*s are further grouped into a *block*. w is value-fixed but b can be specified by programmers for flexibility. Clearly, a GPU with total n_{thd} threads has $\frac{n_{thd}}{wb}$ *block*s and $\frac{n_{thd}}{w}$ *warp*s.

Performance analysis The overall performance of PPR is dominated by the number of required iterations and the runtime specific to each iteration. The former is directly ignored because the path and the speed of propagating messages are consistently identical under Push and Pull, which yields the same number of iterations. We thereby focus on the latter. Further, since the two models follow the same vertex update design, we simply remove its discussion and restrict our analysis to message processing. Intuitively, the *Lock-free* Forward Pull removes all atomic operations in both duplication detection and message management. However, compared with *Lightweight* Forward Push, its frontier vertex array is statically fixed, i.e., all vertices are assumed to be active. That is not true for PPR especially during the convergent stage where only a few vertices hold active. Pull thereby generates many useless reads when pulling messages from inactive in-neighbors. In general, Push and Pull have very different runtime features as analyzed below, which finally affects runtime performance.

For Push, the main job of pushing messages is to update values of destination vertices, incurring random writes and the proportional number of locks for atomic add. We then use the number of random writes Φ_k^{rdw} to measure the runtime cost of the k-th iteration. Recall that the massive threads provided by GPU are organized in batches. Threads involved in one batch actually contribute to runtime only once because of parallel writes. Then a source vertex v with out-degree d_v^+ generates $\lceil \frac{d_v^+}{wb} \rceil$ random writes if $d_v^+ \geq wb$; or $\lceil \frac{d_v^+}{w} \rceil$, otherwise. Besides, a *block* finishes its workload only when all source vertices assigned to it have been processed. In the parallel environment, *block*s in a GPU device form a set $\mathbb{B}$ and they work independently. The final performance is then dominated by the slow, straggling *block*, i.e., the one with the maximal writes. For every *warp* W_j in the set $\mathbb{W}_i$ associated with a *block* $B_i \in \mathbb{B}$, we have the same conclusion. Accordingly, Eq. (1) shows how to compute Φ_k^{rdw}. Here vertices as sources processed by threads in B_i (resp. $W_j \in \mathbb{W}_i$) form V_i^B (resp. V_{ij}^W). V_k^{act} is the set of *active vertices* in the frontier array at the k-th iteration.

$$\Phi_k^{rdw} = \max_{B_i \in \mathbb{B}} \left\{ \sum_{v \in V_i^B \cap V_k^{act}} \left\lceil \frac{d_v^+}{wb} \right\rceil + \max_{W_j \in \mathbb{W}_i} \left\{ \sum_{v \in V_{ij}^W \cap V_k^{act}} \left\lceil \frac{d_v^+}{w} \right\rceil \right\} \right\} \tag{1}$$

Collecting messages in Pull includes two works. For every destination vertex v, they are remotely reading messages from in-neighbors, and locally accumulating messages into v's value, leading to random reads Φ_k^{rdr} and sequential writes Φ_k^{sqw}. Similarly with Φ_k^{rdw}, Φ_k^{rdr} is computed by counting the maximal reads among *block*s and *warp*s, shown in Eq. (2). The only difference is removing the constraint of *active vertices* (i.e., vertices required to be updated) as we assume all vertices are active in Pull.

$$\Phi_k^{rdr} = \max_{B_i \in \mathbb{B}} \left\{ \sum_{v \in V_i^B} \left\lceil \frac{d_v^-}{wb} \right\rceil + \max_{W_j \in \mathbb{W}_i} \left\{ \sum_{v \in V_{ij}^W} \left\lceil \frac{d_v^-}{w} \right\rceil \right\} \right\} \tag{2}$$

Received month dd, yyyy; accepted month dd, yyyy

E-mail: wangning8687@ouc.edu.cn

The number of sequential writes Φ_k^{sqw} is contributed by two operations: saving pulled messages into consecutive *m_rec* spaces which corresponds to Φ_k^{rdr}, and accumulating saved messages into $r(v, s)$ with binary combination if at least one in-neighbor sends a non-zero message to v (i.e., the flag δ_k^v shown in Eq. (3) equals 1). Binary combination runs in logarithmic time, while updating $r(v, s)$ incurs one add operation. Together, we can infer Φ_k^{sqw} using Eq. (4).

$$\delta_k^v = \begin{cases} 1, V_k^{act} \cap \Gamma_v^- \neq \emptyset \\ 0, otherwise. \end{cases} \tag{3}$$

Till now, the performance comparison between *Lightweight* Forward Push and *Lock-free* Forward Pull has been simplified into computing the difference among Φ_k^{rdw}, Φ_k^{rdr} and Φ_k^{sqw}. Purely from the I/O complexity perspective, $\Phi_k^{rdw} \leq \Phi_k^{rdr}$ on average because of the constraint of *active vertices*, and the condition for equality is $V_k^{act} = V$. We then know that at any iteration, $\Phi_k^{rdw} < \Phi_k^{rdr} + \Phi_k^{sqw}$. That is to say, Pull generates more I/Os than Push. However, random writes cost more time than random reads or sequential writes. More importantly, atomic locks associated with the former create another significant performance gap. Thus, we can reasonably infer that when the number of *active vertices* increases, Pull possibly outperforms Push since many expensive random writes and locks are incurred in the latter. Otherwise, perhaps Push beats Pull because of use-less random reads. Consider that the number of *active vertices* dynamically changes with iterations. Neither of them can consistently work best during all iterations. Obviously, the runtime comparison between Pull and Push is non-deterministic. A prominent framework should have the capability of switching models between Push and Pull to dynamically balance the cost of random writes together with locks and random reads. We explore a path to such a target by a hybrid solution on top of Push and Pull.

$$\begin{aligned} \Phi_k^{sqw} = \max_{B_i \in \mathbb{B}} \left\{ \sum_{v \in V_i^B} \left(\left\lceil \frac{d_v^-}{wb} \right\rceil + \delta_k^v \cdot (1 + \log_2^{wb}) \right) + \right. \\ \left. \max_{W_j \in \mathbb{W}_i} \left\{ \sum_{v \in V_{ij}^W} \left(\left\lceil \frac{d_v^-}{w} \right\rceil + \delta_k^v \cdot (1 + \log_2^{w}) \right) \right\} \right\} \end{aligned} \tag{4}$$

Switching timing Now a key issue is how to compute the right switching timing to select the better model for a specific iteration.

Since the performance comparison between Push and Pull depends on Φ_k^{rdw}, Φ_k^{rdr} and Φ_k^{sqw}, we should know what statistics are required when computing the three variables. Seeing their expressions, such information includes which source vertex is *active*, which destination vertex receives non-zero valid messages, and how vertices are distributed across *block*s and *warp*s. The latter is statically available after data assignment, which can be easily collected without any additional runtime cost. However, the former two dynamically change during iterations and they are required by performance estimation for Push and Pull respectively. Thus, for a real-time comparison result, we continuously update Φ_k^{rdw} once encountering an *active vertex* even though the current iteration is running Pull, and compute Φ_k^{sqw} for involved destination vertices, vice versa. Note that some atomic adds are essentially required to collect these statistics, but the cost of vertex-level locking is acceptable.

Another important task is to set weight factors λ_1, λ_2, and λ_3 for reads and writes-locks to indicate the different unit costs when computing the comparison result Ψ_k, as shown in Eq. (5). However, such factors heavily depend on the hardware configurations. Given a GPU, before iterations, we can run all required classic operations on benchmark datasets to get the prior knowledge, e.g., random writes and associated atomic adds concerned in Φ_k^{rdw}. In our experiments, we test the costs of random write-lock, random read, and sequential write, by respectively running the following benchmark operations: 1) atomically saving a pre-fixed value into destination vertices along outgoing edges; 2) reading values of source vertices along incoming edges; and 3) updating PPR scores of every vertex. By recording the runtime and the number of specific operations, we know the unit cost on average. Specifically, we have λ_1:λ_2:λ_3=6.3:3.2:1 for *GPUL*, and 2.3:1.1:1 for *GPUH*. The latter clearly has prominent reading and writing-locking performance.

$$\Psi_k = \lambda_1 \Phi_k^{rdw} - \lambda_2 \Phi_k^{rdr} - \lambda_3 \Phi_k^{sqw} \tag{5}$$

Ψ_k is available at the end of the k-th iteration and then used to predict the comparison result of the (k+1)-th iteration. The framework should run Push if $\Psi_k < 0$, and Pull, otherwise.