Supplementary File of the TKDE manuscript

by Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, *Senior Member, IEEE,* Jeffrey Xu Yu, *Senior Member, IEEE,* and Zhiqiang Wei, *Member, IEEE*
wangzhigang@ouc.edu.cn

Abstract—This supplementary file contains the supporting materials of the TKDE manuscript—"HGraph: I/O-efficient Distributed and Iterative Graph Computing by Hybrid Pushing/Pulling". It improves the completeness of the TKDE manuscript.

1 Proof of Theorem 1

Theorem 1. *Let $F[\mathcal{V}]$ denote the number of* fragments *related to u as* svertex. *$\mathbb{E}(F[\mathcal{V}])$ is the expected $F[\mathcal{V}]$. Then $\mathbb{E}(F[\mathcal{V}]) \propto \mathcal{V}$.*

Proof. Given u in Vblock b_i, let the indicator Z_j denote the event that there exists at least one of its outgoing edges in Eblock g_{ij}, i.e., one fragment exists in g_{ij}. The expectation Z_j is $\mathbb{E}(Z_j) = 1-(1-P[\mathcal{V}])^{d[u]}$, where $d[u]$ stands for the out-degree of u. Here, $P[\mathcal{V}]$ is the probability of putting an edge into g_{ij} among $\mathcal{V}$ Eblocks, and $P[\mathcal{V}] \propto \frac{1}{\mathcal{V}}$. The expected number of fragments is

$$g(\mathcal{V}) = \mathbb{E}(F[\mathcal{V}]) = \sum_{j=1}^{\mathcal{V}} \mathbb{E}(Z_j) = \mathcal{V}\big(1-(1-P[\mathcal{V}])^{d[u]}\big).$$

As can be inferred, the first derivative is

$$g^{'}(\mathcal{V}) = 1-(1+\frac{d[u]-1}{\mathcal{V}})(1-\frac{1}{\mathcal{V}})^{d[u]-1}.$$

Considering that $d[u]>1$ for most graphs, the second derivative is less than zero. Thus, we have: $g^{'}(\mathcal{V}) \geq g^{'}(\mathcal{V}\to+\infty)=0$. Suppose $\mathcal{V}_1 \leq \mathcal{V}_2$. Clearly, $\mathbb{E}(F[\mathcal{V}_1]) \leq \mathbb{E}(F[\mathcal{V}_2])$. □

2 Proof of Theorem 2

Proof. We use S_m, S_v, S_e, S_f represent the average size of per message, per vertex value, per edge, and auxiliary data of each fragment, respectively. We first analyze the features of accessing edges. For BPull, at each superstep, edges are involved in pullRes() to broadcast messages. By contrast, they are accessed in compute() in Push. When all vertices broadcast messages to their neighbors along outgoing edges, $|E^{t-1}| = |E| = \mathcal{M}$. Accordingly, we can compute the I/O cost of edges in Push and BPull, $IO(\mathcal{E}^t) = IO(E^{t-1}) = S_e|E| = S_e\mathcal{M}$. Further, for Push, the I/O bytes of accessing disk-resident messages is: $2(|E|-B)S_m$. For BPull, because $S_m \geq S_v$ and $S_m \geq S_f$, then:

$$IO(F^t) + IO(V^t_{rr}) \leq f(S_f+S_v) \leq 2fS_m.$$

Finally, suppose $B \leq (|E|-f)$, we can infer that:

$$C_{io}(\text{Push}) - C_{io}(\text{BPull}) \geq 2S_m(|E|-B-f) \geq 0.$$

□

3 Proof of Theorem 3

Theorem 2. *Assume that $B_i = B_j$ for $\forall 1 \leq i,j \leq \mathcal{T}$. Let $\mathcal{V}$ and $\mathcal{V}^\alpha$ respectively denote the total numbers of* Vblocks *without and with the limitation of the responding parallelism. Then $\mathcal{V} \geq \mathcal{V}^\alpha$ for algorithms supporting Combiner.*

Proof. By equations shown in Eq. (5) for $1 \leq i \leq \mathcal{T}$, we have

$$\mathcal{V}_i = \frac{2|V_i|}{B_i - \frac{1}{\mathcal{T}+2}\sum_{j=1}^{\mathcal{T}} B_j}$$

Similarly, by Eq. (13), we can infer that

$$\mathcal{V}_i^\alpha = \frac{2|V_i|}{B_i - \frac{1}{\alpha+2}\sum_{j\in\Phi} B_j}$$

When $B_i = B_j$ for $\forall 1 \leq i,j \leq \mathcal{T}$ as assumed, we know

$$\frac{1}{\mathcal{T}+2}\sum_{j=1}^{\mathcal{T}} B_j = \frac{B_i}{1+\frac{2}{\mathcal{T}}} \geq \frac{B_i}{1+\frac{2}{\alpha}} = \frac{1}{\alpha+2}\sum_{j\in\Phi} B_j$$

, since $\alpha \leq \mathcal{T}$. We then have $\mathcal{V}_i \geq \mathcal{V}_i^\alpha$ and hence $\mathcal{V} = \sum_{i=1}^{\mathcal{T}} \mathcal{V}_i \geq \mathcal{V}^\alpha = \sum_{i=1}^{\mathcal{T}} \mathcal{V}_i^\alpha$. □

4 Proof of Theorem 4

Theorem 3. *Suppose that $\mathcal{T}_F$ of $\mathcal{T}$ tasks/nodes fail at superstep-(t+1).* CpPull *outperforms* CpLog *if $\mathcal{T}_F \leq \lambda\mathcal{T}$, where $\lambda \geq \frac{1}{2}$.*

Proof. Before proof, we first introduce some important symbols. At any superstep, given a task, C_{lgr} and C_{lgm} denote the costs of logging *Responding Bitmap* and outgoing messages, respectively. Correspondingly, C_{ldr} and C_{ldm} are the loading costs. C_{prm} means how much time a task takes to produce messages.

Assume that $\mathcal{T}_F$ of $\mathcal{T}$ tasks/nodes fail at superstep-(t+1). At superstep-x where $x \leq t$, we define the overall cost of a fault-tolerant framework as the sum of data-logging cost and failure recovery cost. The former consists of two parts. First, we have logged the bitmap or messages before the current failure happens. Second, for recovering possible failures later, a restarted task also logs its bitmap or messages in recovery. That means, in CpLog, messages for $(\mathcal{T}-\mathcal{T}_F)$ surviving tasks are also produced in recovery, although such messages are not sent. The data-logging cost thereby equals $2C_{lgr}$ or $\big(2C_{lgm} + (1-\frac{\mathcal{T}_F}{\mathcal{T}})C_{prm}\big)$. The recovery cost is dominated by getting messages required for $\mathcal{T}_F$ restarted tasks. For CpPull, such messages are produced based on re-loaded *Responding Bitmap*. Hence, the cost is equal to

$(C_{ldr} + \frac{\mathcal{T}_F}{\mathcal{T}} C_{prm})$. While, for CpLog, a surviving task can directly read logged messages with cost $\frac{\mathcal{T}_F}{\mathcal{T}} C_{ldm}$. Finally, Eq. (1) and Eq. (2) mathematically show the costs of CpLog and CpPull.

$$C(\text{CpLog}) = 2C_{lgm} + \frac{\mathcal{T}_F}{\mathcal{T}} C_{ldm} + (1 - \frac{\mathcal{T}_F}{\mathcal{T}}) C_{prm} \quad (1)$$

$$C(\text{CpPull}) = 2C_{lgr} + (C_{ldr} + \frac{\mathcal{T}_F}{\mathcal{T}} C_{prm}) \quad (2)$$

Logging *Responding Bitmap* is nearly cost-free, compared with logging messages. We thereby ignore C_{lgr} and C_{ldr}. Let $C(\text{CpLog}) \geq C(\text{CpPull})$, we can infer the sufficient condition that makes CpPull outperform CpLog, as shown in Eq. (3). That is, the number of tasks failing at the same superstep must be limited. A straightforward explanation is that in CpPull, the message re-generation cost increases with the number of failed tasks, and then gradually offsets the logging cost in CpLog.

$$\mathcal{T}_F \leq \frac{C_{prm} + 2C_{lgm}}{2C_{prm} - C_{ldm}} \cdot \mathcal{T} = \lambda \mathcal{T}, \;\; s.t. \;\; 0 \leq \mathcal{T}_F < \mathcal{T} \quad (3)$$

Because sequential reads are faster than sequential writes, we have $0 < C_{ldm} < C_{lgm}$, and hence $\lambda > 50\%$. □

5 Algorithms in Experiments

We test six graph algorithms, namely, PageRank [1], Single Source Shortest Path (SSSP) [1], Label Propagation Algorithm (LPA) [2], Simulating Advertisement (SA) [3], Maximal Independent Sets (MIS) [4] and Bipartite Matching (BM) [1]. Because PageRank has already been introduced in Section 2, we simply review other algorithms.

SSSP: It finds the shortest distance between a given source vertex to any other one. Initially, the given source vertex is active and has the shortest distance 0 as its value. In every superstep, a vertex minimizes its distance based on the values of in-neighbors. A newly found distance will be broadcasted to all out-neighbors.

LPA: It is a near linear community detection algorithm based on label propagation. The label value of each vertex is initialized by its own unique vertex id. In the following supersteps, the value is updated by the label that a maximum number of its in-neighbors have. All vertices must broadcast their labels per superstep, in order to collect the whole information from in-neighbors.

SA: It is to simulate advertisements on social networks. Each vertex represents a person with a list of favorite advertisements as its value. A selected vertex is identified as the source and broadcasts its value to out-neighbors. For one vertex, any received advertisement is either further forwarded to out-neighbors or ignored, which is decided by his/her interests. Here we assume that a vertex updates its value by the advertisement that a maximum number of its responding in-neighbors have.

MIS: An independent set is a set of vertices, no two of which are adjacent. Such a set is maximal, denoted by V_{MIS}, if it is not a subset of any other independent set. Let V_{Not} be the set of vertices that are not in V_{MIS}. MIS aims to assign any vertex $u \in V$ to either V_{MIS} or V_{Not}. Assume that V_{UN} is the set of vertices not yet assigned. Initially, both V_{MIS} and V_{Not} are empty, while $V_{UN} = V$. MIS proceeds by alternatively running two kinds of supersteps S_0 and S_1. In S_0, $\forall u \in V_{UN}$, it is moved into V_{Not} if any one of its neighbors is in V_{MIS}. Meanwhile, other vertices in V_{UN} broadcast their ids to indicate they are still unassigned. In S_1, a vertex in V_{UN} is moved into V_{MIS}, if it has the smallest id among all adjacent vertices in V_{UN} (inferred by messages from S_0). Then, new messages are broadcasted to put neighbors into V_{Not} in S_0 of the next round. MIS converges until V_{UN} becomes empty.

BM: The goal of BM is to find a maximal matching on a bipartite graph with two distinct sets of vertices, labelled *Left* and *Right*. Such a matching is a subset of edges without common endpoints, termed E_{BM}. No additional edge can be added to E_{BM}. Computing E_{BM} requires a series of phases. One phase P_i is further divided into four supersteps (P_{i0}-P_{i3}). Specifically, in P_{i0}, each left vertex u broadcasts its id as invitation messages to its neighbors. In P_{i1}, a right vertex v randomly accepts one of invitations and sends back v.id as an acceptance notification. In P_{i2}, several acceptances may be received for any selected inviter, but only one is confirmed by replying a new message. In P_{i3}, a right vertex will receive one confirmation at most. The selected inviter in P_{i2} and the confirmed accepter in P_{i3} mark themselves as matched and then a newly found edge is added to E_{BM}. A matched vertex will not be processed in remaining supersteps. BM terminates when E_{BM} keeps invariant.

Clearly, in LPA, SA, and BM, messages for one vertex u must be processed as a whole to update u. They are non-commutative and non-associative. Hence, PushM lacks built-in support for the three algorithms. Further, Pull and BPull, can only concatenate message values. While, messages in PageRank, SSSP, and MIS, are commutative and associative due to the summing or minimizing update policy. As a result, all tested solutions can run these algorithms and combine messages if possible. From the perspective of hybrid solutions, the periodic vertex behaviors in MIS and BM will suddenly and frequently change the runtime per superstep. We thereby test the two *Sudden Change* algorithms using GHS, while other algorithms are run under BHS.

6 Data management polices for the limited memory scenario

Without loss of generality, we use 5 nodes for small graphs livej, wiki, and ork, and 30 nodes for other large graphs. *Giraph* has multiple disk-based data management policies, and we take the one chosen by Zhou et al. [5]. That is, graph data reside on disk, and, also, partial message data are spilled onto disk if B_i is full. We set the buffer as B_i=0.5 million for small graphs livej, wiki and ork, 1 million for the large graph twi, and 2 million for larger graphs fri and uk. Correspondingly, *MOCgraph* and *GraphLab* use the buffer to cache vertices. In particular, *GraphLab* replicates vertices across tasks for communication optimization, introducing excessive memory space consumption. We thereby set a larger buffer for it (B_i=2.5 million) to guarantee that most vertices ($> 70\%$) reside in memory. Otherwise the runtime is unacceptable. Like *Giraph*, the LRU replacing strategy is used to manage vertices in *GraphLab*. *HGraph* divides B_i on each task into two parts and then assigns them to BS$_i$ and BR$_i$. Graph data in *HGraph* are always on the external storage, but B_i can significantly affect the number of Vblocks and then the I/O-efficiency. For *GraphD*, we follow its default parameter setting given in Section 3 in [6]. More specifically, in our cluster with $\mathcal{T} = 30$ at most, the memory consumption in each *GraphD* task is mainly contributed by the buffers of sending and receiving messages (fixed as 16MB), and the space of sort-merging messages for combination (around 64MB). Consider that one message in our benchmark algorithms occupies 12 bytes at most (4 bytes for an integer destination vertex id and 8 bytes for a double message value). *GraphD* clearly costs more memory resources than *HGraph* ($2\times10^6\times12\approx23$ MB). Even so, our experiments show that *HGraph* still outperforms *GraphD*.

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of SIGMOD*. ACM, 2010, pp. 135–146.
[2] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.
[3] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proc. of Eurosys*. ACM, 2013, pp. 169–182.
[4] D. Peleg, *Distributed computing: a locality-sensitive approach*. SIAM, 2000.
[5] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *In Proc. of the VLDB Endowment*, vol. 8, no. 4, pp. 377–388, 2014.
[6] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd: distributed vertex-centric graph processing beyond the memory limit," *TPDS*, vol. 29, no. 1, pp. 99–114, 2018.