DiterGraph: Toward I/O-Efficient Incremental Computation over Large Graphs with Billion Edges

Yujie Du, Zhigang Wang*, Ning Wang, Luqing Xie, Zhiqiang Wei
College of Information Science and Engineering, Ocean University of China, Qingdao, China
Email: {duyujie, xieluqing}@stu.ouc.edu.cn, {wangzhigang, wangning8687, weizhiqiang}@ouc.edu.cn

***Abstract*—The growing demand for iterative computation over large-scale graphs has attracted a lot of enthusiasm. Distributed-disk systems can accommodate the high-level scalability requirement as graphs grow in size, but the computation is greatly expensive due to a large number of communications and a high frequency of random data-accesses. Alleviating the two limiting factors pose great challenges for graph partitioning, disk-oriented data management and the iterative mechanism. This paper derives insights from the natural locality of raw graphs and then proposes a lightweight partitioning algorithm GPNL with the goal of balancing load and accelerating communication. Accordingly, a hybrid index RC-Index is proposed to improve the I/O-efficiency by reducing disk-accesses for graph data and message data. We also introduce an across-iteration mechanism (AIM) based on the extended BSP model, and then design two policies AIMP and AIMC to prune the message scale and accelerate the message-spreading respectively. Comprehensive experiments versus the state-of-the-art solutions demonstrate significant performance gains over a broad spectrum of real-world and synthetic graphs with up to billion edges.**

***Index Terms*—incremental iteration, large-scale graph, graph partitioning, disk index**

I. Introduction

Graphs are of growing importance in modeling complicated structures and widely applied in multiple scenarios. However, graph volumes are growing rapidly, which seriously deteriorates the processing efficiency. As a prominent solution, memory-resident distributed processing frameworks have been extensively studied and employed. Nevertheless, the performance improvement greatly relies on an effective graph partitioning policy which impacts the communication costs and the load balance. Another key problem is the data processing capacity only based on memories is still insufficient for handling increasing large-scale graphs.

On the other hand, the performance of distributed-disk systems suffers from two limiting factors in scaling them up, poor locality and high frequencies of data-accesses. To make things worse, many graph algorithms naturally fit into the incremental iteration category with a high iterative frequency and sparse computational dependencies [2], [3], such as PageRank and connected component computation. That incurs extremely high I/O costs and exacerbates the computation latency.

Consequently, high-performance and scalable graph processing systems which can handle extremely large graph data (e.g. with ten billion edges) with limited resources to support complex iterative computations are pressingly desired, while facing non-trivial challenges involving graph partitioning, data management and iterative mechanism. We attempt to crack this hard nut in this paper.

A. Incremental Iteration over Large Graphs

A graph model can express the complicated relations among vertices, but the frequent state update only impacts its outbound neighbors instead of the other vertices. Ewen S et al. refer to it as sparse computation dependencies [2]. Further, most graph algorithms can be implemented asynchronously, namely incremental iterations [3].

Note that incremental iterations exhibit two features: (1) vertices become convergent at different speeds; (2) messages can be processed at any time instead of a synchronous point. This paper focuses on incremental iterations and leverages its features to accelerate disk-accesses and communications.

B. Challenges

Fast partitioning with multiple-objectives. The first key challenge derives from the need to distribute graph over a cluster, while satisfying threefold objectives: load balance, minimizing the number of across-task edges and prominent efficiency. That is a well-known NP-Hard problem [4] and there are two different research paradigms, hash-based and heuristic algorithms. The former focus on efficiency and vertex-balance, but omit locality, which incurs massive edge cuts [4]. The latter try to improve locality and balance [4]–[6], [24] by sophisticated yet time-consuming heuristics.

Expensive I/O costs in disk-accesses. A distributed disk-resident system can naturally solve the scalability problem. But graph algorithms are inefficient when data are resident on the disk, because they exhibit inherently poor locality, which incurs many random disk-accesses. Moreover, the high frequency iteration processing exacerbates this issue. Furthermore, the gradual convergence and sparse computation dependencies incurs considerable wasteful data-accesses since we need to read data from the local disk in block. Specially, the intermediate data (i.e., messages) need to be accessed repeatedly across iterations and the volume is comparable to that of outbound edges, which is a dominant disk-I/O cost. Although considerable endeavors have been devoted to managing data on the disk [10]–[14] from the local graph data perspective, seldom solutions are designed for optimizing messages resident on disks [25].

*Corresponding author.

Dilemma for the vertex-updating frequency and message-processing efficiency. Disk-accesses incurred by vertex updating and message-processing are two dominant costs for distributed disk-resident computations. However, existing iterative mechanisms for incremental iterations basically fall into one of the two categories [2], synchronous [7], [10]–[12] and asynchronous [3], [6] iterations, which cannot simultaneously improve the efficiency of aforementioned issues. The former (e.g., the well-known Bulk Synchronous Parallel model, *BSP* [16]), exhibits a deterministic and controllable computation complexity, which is more suitable for disk-resident large graphs. However, compared with the asynchronous mechanism, the latency of spreading messages will cause serious performance bottleneck. Note that we have tried to address this issue by extending *BSP* and designing a message-pruning policy [14] for the special max-min problem, such as the shortest path computation and connected component computation. However, it cannot work for many other widely-used summation algorithms, such as PageRank, Adsorption, and HTIS [3].

C. Our Contributions

In this paper, we design a prototype system, DiterGraph, with built-in support for iterative processing over disk-resident large graphs in parallel. It consists of graph partition method (GPNL), disk index (RC-Index) and iterative mechanism (AIM) to separately solve the three challenges discussed.

Firstly, we derive insights from the fact that raw graphs exhibit natural locality [17], and then tackle the multiple-objectives for graph partition by GPNL (graph partition based on the natural locality). Based on a continuous parallel method of loading data (*CP-Load*), we preserve the natural locality and guarantee the balance of *VE-load*, which is a novel metric taking the number of vertices and also outbound edges into account and can measure workload more accurately compared with vertex-only and edge-only metrics. Then, we back up vertices to minimize the edge-cut scale [6], [8], [9] and adopt a novel peer-to-peer edge exchanging algorithm to control the *VE-load* imbalance. Also, the efficiency of GPNL is prominent since it uses a range-based route table instead of sophisticatedly heuristic rules.

Then we present a hybrid RC-Index (row-column index) to optimize disk I/O costs, especially for message data. It is grafted into GPNL and inherits the advantages of our proposal for optimizing I/O costs of local graph data dynamically [14]. Note that its column index mechanism makes a considerable fraction of messages processed in a real-time memory stream, which significantly reduces the I/O costs of messages.

Finally, observing that most graph algorithms basically fall into one of two categories: max-min and summation problems, we design an across-iteration mechanism (AIM).

The remainder of this paper is organized as follows. Sec. II reviews related works. Sec. III describes the details of GPNL. Sec. IV proposes the architecture of RC-index. Sec. V puts forward the AIM solution. Sec. VI evaluates our methods versus the state-of-the-art work. Sec. VII concludes our work.

II. Related Work

Since a lot of efforts have been devoted to processing large graphs, we survey a few representative graph systems from four aspects: graph partition, load balance, disk index, and iterative mechanism.

A. Graph Partition and Load Balance

Existing systems, such as Pregel, Hama, Giraph, and Spark [7], [11]–[13], resort to a hash-based vertex placement. The straightforward solution is efficient to balance vertex-loads, but the poor locality incurs a large number of edge cuts, exacerbating communication overheads. To alleviate this issue, Trinity and GPS [8], [9] propose a vertex-backup mechanism to distribute edges of a high-outbound-degree vertex over tasks where their destination vertices are maintained. The performance gain relies on a metric parameter of the high-outbound-degree, whose appropriate value resorts to extensively experimental attempts. It is worth noting that the vertex-backup mechanism destroys the edge-load balance.

Another main line of research is devoted to heuristics. Recently, several central heuristics have been proposed in [4] and [5]. Note that in [4] and [5], outbound edges of one vertex must be entirely assigned to one task, which limits the effect of reducing the edge-cut scale. PowerGraph [6] proposes a distributed vertex-replication based partitioning method. The communication improvement is significant but the partitioning efficiency is still high because of maintaining a complex distributed lookup table. Note that our previous work tries to utilize locality in partitioning [24], but vertex replication is not used, and hence the effect of communication optimization is limited.

B. Disk Index

Due to the poor locality and high frequency of data-accesses, it is frustratingly difficult to efficiently manage data on the disk for iterative graph algorithms. Thus, many systems like PowerGraph and Spark [6], [8], [9], [13] acquire prominent performance gains under an assumption that data are resident in memory. Nevertheless, some systems attempt to alleviate the issue of insufficient memory capacity. In HaLoop [10], graph data are resident on the disk, and hence a static index relying on the inherent sort function of Hadoop [19] is designed to reduce costs of matching graph data and message data. Furthermore, a tunable hash index tries to minimize I/O overheads of graph data by leveraging a state-transition model of the shortest path [14]. In a recent work, GraphChi performs an efficient disk-based graph computation under an inherent limitation that it must be performed on just a PC due to the serial vertex update [20]. Besides, HGraph [25] tries to reduce message I/O costs but lacks in effective partitioning method and hence cannot utilize vertex-replication.

C. Iterative Mechanism

The state-of-the-art frameworks on iterative mechanisms seem to roughly be divided into two main lines of research. Existing disk-resident systems resort to a synchronous method which typically optimizes I/O costs, such as HaLoop, Pregel, Giraph, Hama and Spark [7], [10]–[13]. These systems have built-in support for the bulk synchronous parallel model (BSP) [16]. On the other hand, since there is no inherently synchronous requirement for incremental iterations, some memory-resident systems support asynchronous iteration to overcome the drawback of message-spreading latency of BSP, such as PowerGraph and Maiter [3], [6], [9]. However, it can incur many more I/O operations since vertices are updated more frequently. Ewen S et al. find that even for memory-resident systems, the former outperforms the latter over density graphs [2]. Our previous work extend BSP and propose an efficient message-pruning policy for the shortest path over disk-resident large graphs [14], which is only a special case of our unifying framework AIM.

III. GPNL: Graph Partition based on the natural locality

A. Preliminaries

Let $G = (V, E, w)$ be a weight directed graph with $|V|$ vertices and $|E|$ edges, where $w : E \to \mathbb{N}^*$ is a weight function and w_{ij} denotes an impacting metric from v_i to v_j. We assume that G is organized with the adjacency list and each vertex is assigned a unique ID which is numbered consecutively. For vertex *v*, $\Gamma(v)$ indicates its outbound neighbors and $\Gamma^{in}(v)$ is the inbound neighbor set. Then the incremental iteration computation is expressed by Formula 1. At the *(k+1)th* iteration, the incremental value of v_j is obtained by applying the incremental update operation '$\oplus$' on the impact value Δv_{ij}^{k+1}, where $v_i \in \Gamma_{sub}^{in}(v_j) \subseteq \Gamma^{in}(v_j)$.

Since the incremental iteration satisfies constraints of the accumulative iteration proposed by Zhang Y et al. [3], '$\odot$' has the following properties : $(\sum \oplus x) \odot y = \sum \oplus (x \odot y)$, $x \oplus y = y \oplus x$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$, and $x \oplus \mathbf{0} = x$.

$$\begin{cases} v_j^{k+1} = v_j^k \oplus \Delta v_j^{k+1} \\ \Delta v_j^{k+1} = \sum \oplus (\Delta v_{ij}^{k+1}) \\ \Delta v_{ij}^{k+1} = v_i^k \odot w_{ij} \end{cases} \tag{1}$$

Now, we introduce a new metric of the load balance, *VE-load*. Intrinsically, the computation load per iteration of one task P_i, $CompLoad(P_i)$, is threefold: loading graph data (vertices and edges) from disk, updating vertex values and broadcasting messages for neighbors along edges. Apparently, $CompLoad$ is affected by the number of vertices $|V_i|$ and edges $|E_i|$ on P_i. However, existing graph partition methods evaluate the skew degree only by $|V_i|$ or $|E_i|$. That is obviously imprecise in contrast with *VE-load*.

Note that for *CP-Load*, raw data are consecutively separated into several parts with even byte size, and then each task is assigned only one part. Thus, *VE-load* of $|P|$ tasks can be viewed to be balanced. Since every vertex ID is numbered consecutively, we can build a range-route table which denotes the range of vertex IDs per task. Then, given a vertex ID, to locate the associated adjacency list, we need to first 1) locate the task that contains it based on the range-route, and then 2) locate it from local graph data.

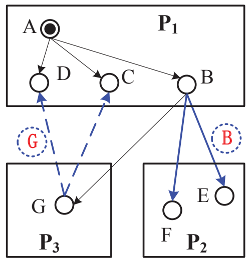


Fig. 1. Locality analysis of raw graphs

B. Locality Analysis of Raw Graphs

Although the balanced graph partition is a well-regard NP-hard problem [4], this issue may be tackled approximately by leveraging the natural locality of raw graphs. Before discussing details of locality analysis, we introduce the definition of *Backtracking Edge*.

Definition 1. *During building the raw graph G_{raw}, let V_{exist} be vertices which have existed in G_{raw}. Then for a new vertex* x *and its outbound neighbors $\Gamma(x)$, we define $e_i =< x, y_i >$ as Backtracking Edge if $y_i \in V_{exist}$.*

To the best of our knowledge, the main stream technique of building raw graphs is breadth-first search (BFS) [17]. Note that the raw graph exhibits a natural locality [24]. As illustrated in Figure 1, neighbors of one vertex trend to be placed in the raw graph file nearly together. That means they can be maintained in the same task in *CP-Load*. For example, source vertex *A* in BFS and its neighbors {*B*, *C*, *D*} belong to the same task P_1. Network communications between such neighbors and the source vertex are thereby reduced. Note that, since *CP-Load* splits the raw data file evenly in size, neighbors of vertex *B* are assigned to P_2 and P_3, which impairs the locality. Also, backtracking edges $\langle G, C\rangle$ and $\langle G, D\rangle$ also incurs communication costs.

Apparently, in comparison with the straightforward hash-based method, *CP-Load* alleviates the communication bottleneck due to preserving the locality of raw graphs. However, the existence of *Backtracking Edges* still incurs considerable communication costs, which is addressed by our GPNL.

C. GPNL for Graph Partitioning

The high level goal of GPNL is to minimize the number of cross-task edges by leveraging a θ-backup policy, while preserving the *VE-load* balance provided by *CP-Load*.

1) θ-backup Policy: As shown in Figure 1, we can back up a ghost vertex $\mathbb{G}(v_i)$ for v_i and transfer the associated edge information onto the task which maintains partial outbound neighbors of v_i, and then compute impact values (i.e., messages) and send them to its neighbors locally. For incremental iterations defined by Formula 1, the impact value Δv_{ij} only depends on v_i and the special edge information w_{ij}. Thus,

$\mathbb{G}(v_i) \odot w_{ij} = v_i \odot w_{ij}$. For instance, by backing up G and B, only two messages are required to synchronize the values of $\mathbb{G}(G)$ and $\mathbb{G}(B)$ (i.e., ghost vertices) via network, and hence the message scale is reduced by two.

Now, we describe *θ-ghost vertex set* to measure the communication costs of synchronizing ghost vertices.

Definition 2. *Suppose that task P_i backs up vertices on task P_j and the ghost vertex $\mathbb{G}(v_i)$ exists on P_j if $|\Gamma(v_i) \cap V_j| > \theta$, where the parameter θ denotes a backup threshold, then the* θ-ghost vertex set *on P_j is:*

$$V_{ij}^{\theta} = \{v_i | v_i \in V_i, |\Gamma(v_i) \cap V_j| \geq \theta\}.$$

Obviously, the message scale can be reduced only if $\theta \geq 2$ in theory. We will discuss a recommendable value in experiments in detail.

2) GPNL: Now our GPNL tries to minimize the communication cost and preserve *VE-load* balance by adjusting V_{ij}^{θ}. It formalizes the two objectives by Formula 2 and Formula 3 respectively. Here, $|V_{kj}^{\theta}|$ is the newly incurred synchronizing cost. $|\Delta E_{k \to j}|$ is the number of edges transferred from P_k to P_j, which denotes the reduced message communication costs.

$$\max\{\sum_{k=1}^{|P|} \sum_{j=1}^{|P|} (|\underset{k \to j}{\Delta E}| - |V_{kj}^{\theta}|)\}, k \neq j \tag{2}$$

$$\max\{|V_k| + |E_k| + \sum_{j=1}^{|P|} |\underset{j \leftrightarrow k}{\Delta E}|\} \leq \rho_{ve} \cdot \frac{|V| + |E|}{|P|}, k \neq j \tag{3}$$

We assume that ghost vertices are resident in memory, which can be guaranteed by peer-to-peer edge exchanging described later in this section. On the other hand, every ghost vertex only receives the synchronous message once per iteration. Thus, the cost of maintaining ghost vertices is determined by their outbound edges. That implies the number of transferred edges affects the *VE-load* balance. Given task P_k, for any other task P_j, the difference between receiving and sending edges is evaluated by Formula 4. Thus, Formula 3 can make sure that the maximum degree of imbalance is smaller than $\rho_{ve} \cdot (|V| + |E|)/|P|$, where, ρ_{ve} is an imbalance parameter which denotes the maximum skew degree.

$$|\underset{j \leftrightarrow k}{\Delta E}| = |\underset{j \to k}{\Delta E}| - |\underset{k \to j}{\Delta E}| \tag{4}$$

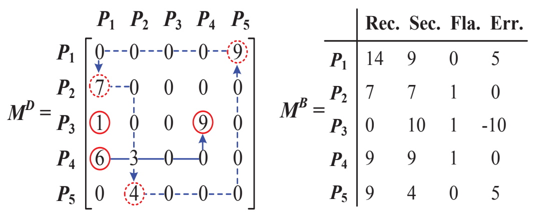
Now we outline the overview of GPNL. During *CP-Load*, we suppose all outbound edges will be transferred on the corresponding tasks, and then obtain a statistics histogram set H and an edge exchanging matrix M^E. $\forall p_{ij} \in M^E$, p_{ij} denotes the total number of edges which can be transferred from task P_i to task P_j. While, a statistics histogram H_{ij} describes the gain distribution of edges from P_i to P_j. For instance, $H_{ij}[5, 10] = 40$ denotes that, $\exists v_i \in V_i$, $5 \leq \Gamma(v_i) \cap V_j \leq 10$, and the number of satisfied vertices in V_i is 40. That means the communication gain is between $40 \times (5 - 1) = 160$ and $40 \times (10 - 1) = 360$.

Based on H and M^E, GPNL performs a peer-to-peer edge exchanging algorithm to adjust V_{ij}^{θ} to satisfy the given ρ_{ve} constraint. After that, we obtain an adjusted matrix M^A and its element is the number of really transferred edges. And then, $\forall p_{ij} \in M^A$, we choose optimal edge compounding to be transferred on P_j according to H_{ij}.

3) Peer-to-Peer Edge Exchanging: Note that if two tasks exchange the same number of edges, their *VE-load* will not be destroyed. Thus, before analyzing the maximum number of transferred edges, we first preprocess M^E by $M^D = M^E - M^S$, where M^S is a symmetric matrix based on M^E. Given ρ_{ve}, if M^D satisfies the constraint of Formula 5, then we view it as a *VE-load* balance matrix, which is the objective of our peer-to-peer edge exchanging algorithm.

$$\max\{\sum_{j=1}^{|P|} |\underset{j \leftrightarrow k}{\Delta E}|\} \leq (\rho_{ve} - 1) \cdot \frac{|V| + |E|}{|P|} = \mu(\rho_{ve}) \tag{5}$$

To improve the computation efficiency, we incur a balanced-controlling matrix M^B. As illustrated in Figure 2, M^B is constructed based on M^D. For P_i, $Rec.(P_i) = \sum_{k=1}^{|P|} p_{ki}$, $Sec.(P_i) = \sum_{k=1}^{|P|} p_{ik}$, where $p_{ki}, p_{ik} \in M^D$. $Err.(P_i) = Rec.(P_i) - Sec.(P_i) - \mu(\rho_{ve})$. $Fla.(P_i) = 0$ if $Err.(P_i) > 0$, otherwise, $Fla.(P_i) = 1$.



$$M^D = \begin{array}{c|ccccc} & P_1 & P_2 & P_3 & P_4 & P_5 \\ \hline P_1 & 0 & 0 & 0 & 0 & 9 \\ P_2 & 7 & 0 & 0 & 0 & 0 \\ P_3 & 1 & 0 & 0 & 9 & 0 \\ P_4 & 6 & 3 & 0 & 0 & 0 \\ P_5 & 0 & 4 & 0 & 0 & 0 \end{array}$$

(a) Difference Matrix M^D

$M^B =$

	Rec.	Sec.	Fla.	Err.
P_1	14	9	0	5
P_2	7	7	1	0
P_3	0	10	1	-10
P_4	9	9	1	0
P_5	9	4	0	5

(b) Balance Matrix M^B

Fig. 2. Difference matrix and balance-control matrix for edge exchanging

Clearly, a basic implementation of edge exchanging algorithm is to scan M^B and then, for P_i, reduce the scale of edges transferred by P_k, if $Fla.(P_i) = 0$, $i \neq k$. Let Δ be the reduced edge scale, $\Delta = Err.(P_i)$. Specifically, we first reduce tasks whose $Err. < 0$ and the reduced maximum scale of P_k is $\Delta_{ki} = \min\{\Delta, p_{ki}, |Err.(P_k)|\}$. We sort Δ_{ki} in descending order and reduce them one by one until the total reduced scale is equal to Δ. If $Err.(P_i)$ is still more than 0, then we process tasks whose $Err. \geq 0$ similarly but the reduced maximum scale of P_k is $\Delta_{ki} = \min\{\Delta, p_{ki}\}$. After that, if $Fla.(P_k) = 0$, then it must be processed recursively. The scanning operation is terminated if $\forall P_i, Fla.(P_i) = 0$, which is the convergent condition.

IV. RC-Index for Disk-Resident Data

A. The Architecture of RC-Index

The high level goal of our RC-Index is to optimize I/O costs of data-accesses, including a row-index for graph data and a column-index for ghost data. Here, ghost data consists of ghost vertices and the associated edge information.

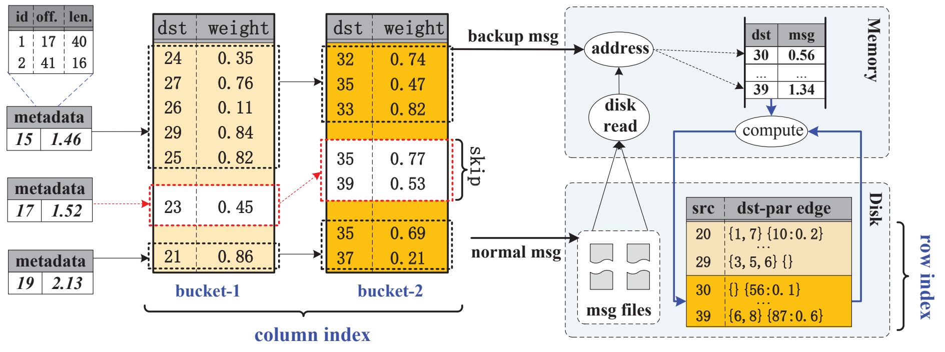


Fig. 3. RC-index

An inherent cost of distributed graph computations is to match messages and the local graph data. To alleviate the impact of poor locality, we partition local graph data by a simple Hash function, such as $v.ID\%number\_of\_buckets$. And message data are organized as the same Hash function. Since we consume messages and update vertices in the same order of $hash\_bucket\_id$, we only need to cache messages of one bucket and then access the corresponding graph data from the local disk. In this case, the disk seek operations are only conducted in a forward manner, and in the worst case, the input cache is sequentially scanned from local disk only once in each iteration.Furthermore, vertices (dynamical data) and outbound edges (static data) are separated into two files, and each vertex has an associated pointer to its associated outbound edges. Edge data are read-only and hence the I/O cost is reduced.

Although GPNL reduces network communication costs, the computation of generating messages is only transferred from the backtracking task to the destination task. I/O costs of messages are still considerably huge. Therefore, we create a column-index for ghost data to address this issue. Ghost data are also organized as the adjacency list, but outbound edges are partitioned as the same Hash function of the local graph data. Each ghost vertex maintains offsets of its associated outbound edges in different buckets. Based on the assumption that ghost vertices are resident in memory (that can be implemented by controlling the backup vertex scale in GPNL), we postpone the message generation until they are required by the corresponding bucket of local graph data.

Figure 3 depicts a concrete instance of TRC-Index. Since local graph data are partitioned into two buckets $\{20, 29\}$ and $\{30, 39\}$, outbound edges of ghost vertices are also separated into two files. For instance, for the ghost vertex with $VertexID = 15$, its metadata is composed of a series of triples $\langle$*bucketId*, *offset*, *length*$\rangle$. When local graph data in bucket-2 are processed, we set a memory block to cache messages sent to it. According to Formula 1, we only need to save Δv_j^{k+1} instead of all Δv_{ij}^{k+1}. messages are generated from too sources. Some parts are generated by reading outer edges in bucket-2 of ghost vertices, and the other parts are read from the local disk or the temporary-receiving cache. The latter are generated by mirror vertices and sent via network, while the former are generated by ghost vertices and sent via local memory. Note that the adjacency list will be skipped if the ghost vertex has not received a synchronous value.

B. Analysis of the I/O Costs

In comparison with the existing row-index [14], the I/O efficiency of TRC-index mainly relies on the column-index. Therefore, we focus on analyzing its improvement in this section. Suppose that α_i denotes the ratio of transferred edges among the whole inbound edge set E_i^{in} of P_i. h_i is the number of buckets created for local graph data. And the memory capacity, B_i, is split into $|P|{\cdot}h_i$ units to cache messages for every local buckets from different tasks respectively. Then, Formula 6 describes I/O costs without the column-index. $3\alpha_i|E_i^{in}|$ denotes costs of data-accesses: reading edges, writing and reading messages. Here, Θ is the disk seek cost.

$$\sum_{i=1}^{|P|}(3\alpha_i|E_i^{in}| + \Theta(\max\{h_i, \frac{\alpha_i|E_i^{in}|\cdot h_i|P|}{B_i}\})) \quad (6)$$

On the other hand, the cost of our TRC-index is evaluated by Formula 7, where δ_i denotes reading costs of destination task IDs for mirror vertices. $\delta_i < |E_i^{in}|/\theta$, because there is no weight information for destination task IDs and θ is the backup threshold.

$$\sum_{i=1}^{|P|}(\alpha_i \cdot (\delta_i + |E_i^{in}|) + \Theta(h_i)) \quad (7)$$

Obviously, the I/O cost of TRC-index is less than that of the original row-index.

V. AIM BASED ON EBSP

A. The Architecture of AIM

An extended BSP model (EBSP) makes messages between two consecutive iterations visible, while simultaneously inherits the synchronous update mechanism of BSP. Based on EBSP, the update operation '$\oplus$' in Formula 1 can be expressed

src	value	k_state	(k+1)_state
a	1.4	1	1
b	1.7	1	0
c	1.3	0	1

Table A: gost vertex

dst	msg	active
d	0.56	1
e	0.68	0
f	0.56	0

Table B: received messages

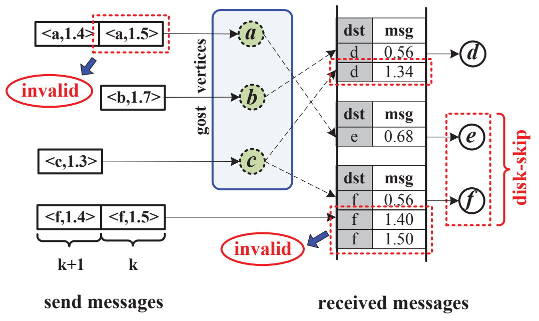


Fig. 4. Across-iteration mechanism for SSSP

as $v_j^{k+1} = v_j^k \oplus \Delta v_j^{k+1} \oplus \Delta v_j^{k+2}$, which is defined as the across-iteration mechanism (AIM).

To minimize extra costs of AIM, only memory-resident messages of the next iteration are consumed at the current iteration. Our GPNL and TRC-Index strengthen the performance gain of AIM. The reason is that ghost vertices and messages generated by them are resident in memory, which increases the number of messages involved in AIM.

B. AIMP and AIMC

In this section, we describe the corresponding different implementations of AIM, AIMP (across-iteration message pruning) and AIMC (across-iteration message combination), exemplified by SSSP (the single source shortest path) and PageRank, which are two representatives respectively.

1) AIMP for the Min-Max Problem: Figure 4 illustrates our AIMP method for SSSP. The objective of SSSP is to find a shortest distance value away from a given source vertex. For instance, at kth iteration, $\mathbb{G}(a)$ receives two synchronous messages, 1.5 and 1.4, sent by mirror vertex a at iteration k and *k+1* respectively. Since $1.5 > 1.4$, then the former is invalid. For AIMP, it is ignored, and $\mathbb{G}(a)$ only stores 1.4. Then, as shown in Table A, its *k_state* and *(k+1)_state* are both set up to 1, which means $\mathbb{G}(a)$ will execute '$\odot$' at the two consecutive iterations. Note that $\mathbb{G}(b)$ receives $\langle$b, 1.7$\rangle$ of iteration k, then only *k_state* is set up to 1. Specially, $\mathbb{G}(c)$ just receives $\langle$c, 1.3$\rangle$, then only *(k+1)_state* is 1.

Once the associated local graph data are being processed, ghost vertices with *k_state=1* or *(k+1)_state=1* perform '$\odot$' and send messages into the message cache. Specially, if *(k+1)_state* is 1 and the message is less than the current value in the cache, then the cached value will be updated and its *active* is set up to 0, which means it is only used to prune invalid messages. For instance, $\langle$d, 1.34$\rangle$, $\langle$f, 1.40$\rangle$ and $\langle$f, 1.50$\rangle$ are pruned. And that is suitable for normal messages sent by mirror vertices at iteration *(k+1)*. By contrast, we cache messages with smaller values at iteration k and set *active* up to 1. Only messages with *active=1* are performed $\oplus$ by the associated local graph data. For the concrete example in Figure 4, vertex d, e and f are skipped at iteration k. After iteration k, the value of *(k+1)_state* is transferred into *k_state* and *(k+1)_state* is reset to 0.

2) AIMC for the Summation Problem: Compared with SSSP, the scenario of PageRank is relatively simple. Once receiving synchronous messages and normal messages sent for iteration *(k+1)*, we refer to them as messages for iteration k in AIMC if messages for iteration k have been received. Otherwise, for synchronous messages, we perform '$\oplus$' for ghost vertices, and then set *k_state=0* and *(k+1)_state=1*. Note that only ghost vertices with *k_state=1* perform $\odot$ at iteration k in AIMC. And for normal messages for iteration *(k+1)*, we just store them and consume them at iteration *(k+1)*. Thus, we can accelerate the message-spreading and prune invalid operations ($\oplus$ and $\odot$).

VI. Experimental Evaluation

To evaluate our patterns, we have developed a prototype system *DiterGraph*. Data sets are listed in Table I. All optimization policies are evaluated over real graphs. Note that the performance gains of Spark mainly benefits from managing data in memory and it lacks built-in optimization for disk I/O. Thus, we validate the data processing capacity of DiterGraph over synthetic data sets using Giraph [12] and HaLoop [10].

TABLE I
Graph Data Set

Data Set	Vertices	Edges	Avg.	Disk Size
S-LJ [21]	4.8M	68M	14.23	0.9GB
USA-RN [22]	23.9M	5M	0.244	1.2GB
Wiki-PP [23]	5.7M	130M	22,76	1.5GB
Tiwtter [1]	41.7M	1.47B	35.25	12GB
Syn-D_x	100M	4-10B	40-100	33-84GB

Our cluster is composed of 31 nodes which are connected by gigabit Ethernet to a switch. Every node is equipped with 2 Intel Core i3-2100 CPUs, 2GB available RAM and a Hitachi disk (500GB and 7,200 RPM).

A. Evaluation of GPNL

GPNL can acquire a compromising performance in terms of edge cut fraction (λ), ρ_{ve} and efficiency. Table II describes these metrics of different methods over Twitter, where LDG is a prominently central-heuristic algorithm [4]. In addition, the running time of GPNL is only 160s ($|P| = 30$), which outperforms LDG (620s).

And then we analyze the effect of θ and ρ_{ve}. In our experiments, we set the number of tasks to 10 and collect the statistics of the first 5 iterations by running PageRank. First, we set $\rho_{ve} = \infty$ and as shown in Figure 5(a), a higher θ results in the increasing of λ because the scale of edges allowed to be transferred decreases. Nevertheless, the overall performance and loading efficiency are improved when θ increases from 1 to 4 (Figure 5(b) and (c)). In theory, the communication gain must be positive when $\theta \geq 2$. However, the cost of maintaining

TABLE II
THE EDGE-CUT FRACTION AND LOAD BALANCE OF LDG, HASH AND GPNL

#cluster($\|P\|$)	LDG				Hash Partition				GPNL			
	λ	ρ_v	ρ_e	ρ_{ve}	λ	ρ_v	ρ_e	ρ_{ve}	λ	ρ_v	ρ_e	ρ_{ve}
18	79.20%	1.5	1.05	1.03	93.07%	1	1.12	1.11	80.40%	2.79	1.03	1.02
24	81.37%	1.9	1.05	1.03	94.40%	1	1.16	1.15	82.57%	2.94	1.03	1.02
30	83.07%	1.84	1.05	1.03	95.10%	1	1.65	1.63	83.72%	3.03	1.03	1.02

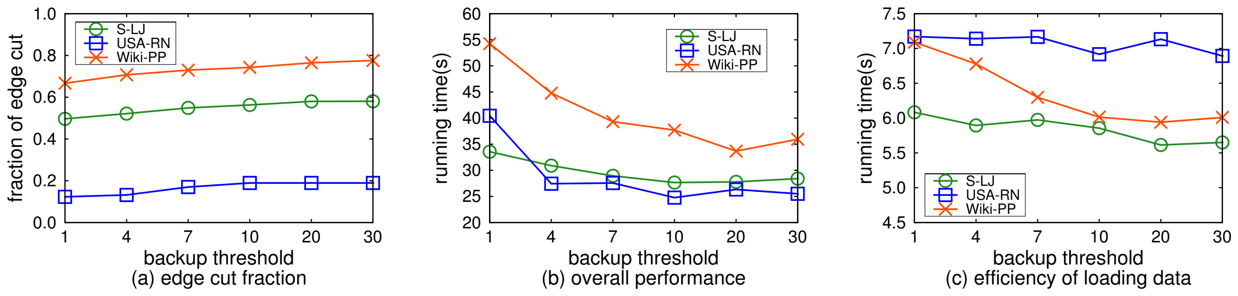


Fig. 5. Impact of different θ over real graphs (S-LJ, USA-RN and Wiki-PP)

ghost data offsets the communication gain. Therefore, from the overall performance perspective, a reasonable value is 4 in our experiments. Note that the overall performance increases when $\theta > 4$ due to the negative impact on λ. Another interesting phenomenon is that the negative impact is tiny when θ is more than the average outbound degree, for most vertices, $|V_{ij}^{\theta}| < \theta$.

B. Evaluation of RC-Index

Figure 6 depicts a poignant contrast from the perspective of the disk-resident message scale, exemplified by PageRank. In this suite of experiments, $\theta = 4$ and $\rho_{ve} = 1.0$. Except for the first iteration without received messages, RC-Index has a substantial gains for subsequent iterations. Especially, USA-RN has the best improvement and nearly 50% messages are processed in memory instead of the local disk due to its lower natural-skew of *VE-load*. However, for S-LJ and Wiki-PP, the gain is not as obvious as that of USA-RN since GPNL reduces the ghost data scale to guarantee the *VE-load* balance.

C. Evaluation of AIMP and AIMC

This suite of experiments is used to examine the effect of AIMP and AIMC by running Connected Computations and PageRank respectively, where $\theta = 4$, $\rho_{ve} = 1.0$, $|P| = 10$, and we collect statistics of partial iterations. As shown in Figure 7(a), the scale of updating vertices has been reduced by up to 45% since a large number of invalid messages have been pruned. And hence, the number of sending messages has also been reduced by 27% on average. On the other hand, the convergent speed of PageRank is improved by 16.4% on average (Figure 7(c)). The reason is that AIMC can process messages of the next iteration, and hence accelerates the message-spreading.

D. Evaluation of Scalability

In this section, we validate the data processing capacity of DiterGraph by comparing it with existing disk-resident systems, Giraph and Haloop. We set the number of tasks to the number of nodes. First, as shown in Figure 8(a), when the vertex size varies from 0.5 million to 5 million (avg. outer degree is 19.5), the running time of DiterGraph is at least 18 times faster than that of Giraph and Haloop. Second, we evaluate the scalability of DiterGraph by varying graph sizes and node number (Figure 8(b)). Given 30 nodes and suppose that every synthetic graph has 100 million vertices, when the number of edges varies from 4 billion to 10 billion, the increase from 212 seconds to 1070 seconds demonstrates that the running time increases linearly with the graph size. On the other hand, given the graph size, such as 10 billion edges, the running time decreases from 3988 seconds to 1070 seconds when the number of nodes increases from 10 to 30.

DiterGraph is insensitive to the outbound edge scale since GPNL reduces the communication costs and TRC-Index significantly optimizes the scale of messages spilled onto the local disk. Thus, for a given cluster with 30 commodity PCs and 60 GB available memory capacity, we can perform an extremely large graph even with ten billion edges.

VII. CONCLUSIONS

In this paper, we have demonstrated a distributed system, DiterGraph, for incremental iterations of disk-resident large graphs. It employs GPNL, RC-index and AIM techniques to improve overall performance from aspects of partitioning, data management and message processing. The extensive experiments demonstrate the great performance gains on a broad set of real-world and synthetic graphs.

VIII. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (61902366 and 61902365), the Fundamental Research Funds for the Central Universities (202042008), the Project funded by China Postdoctoral Science Foundation (2020T130623, 2019M652474 and

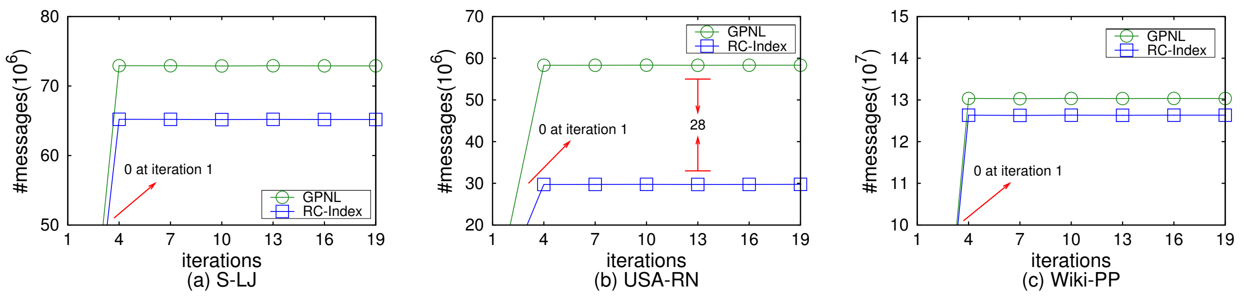


Fig. 6. Analysis of RC-Index over real graphs (S-LJ, USA-RN and Wiki-PP)

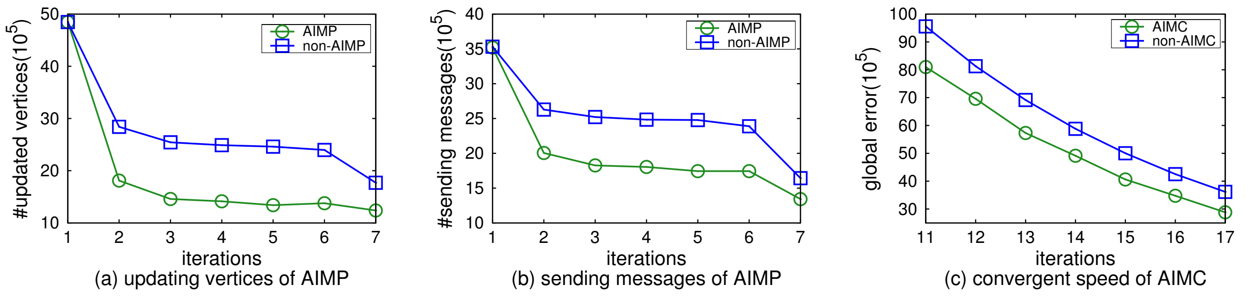
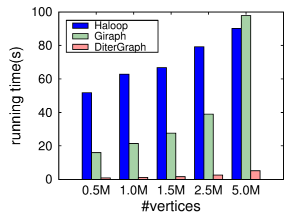
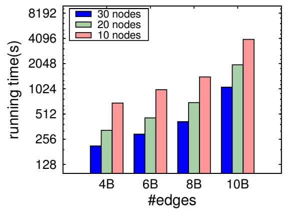


Fig. 7. Analysis of AIMP and AIMC over the real graph S-LJ



(a) DiterGraph .vs others



(b) Scalability

Fig. 8. Overall scalability analysis

2019M652473), the Projects funded by Postdoctoral Creative Foundation in Shandong Province and Postdoctoral Application&Research Foundation in Qingdao City, the Qingdao Independent Innovation Major Project (20-3-2-12-xx), and Postgraduate Professional Development Fund Project of Ocean University of China(CSZS2021003).

REFERENCES

[1] Kwak H, Lee C, Park H, et al. What is Twitter, a Social Network or a News Media?[C]. WWW, 2010.

[2] Ewen S, Tzoumas K, Kaufmann M, et al. Spinning fast iterative data flows[J]. In Proc. of the VLDB Endowment, 2012, 5(11): 1268-1279.

[3] Zhang Y, Gao Q, Gao L, et al. Accelerate large-scale iterative computation through asynchronous accumulative updates[C]. In Proc. of the 3rd workshop on Scientific Cloud Computing Date, 2012: 13-22.

[4] Stanton I, Kliot G. Streaming graph partitioning for large distributed graphs[C]. SIGKDD, 2012.

[5] Tsourakakis C E, Gkantsidis C, Radunovic B, et al. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs[R]. 2012.

[6] Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed graph-parallel computation on natural graphs[C]. OSDI, 2012.

[7] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]. SIGMOD, 2010.

[8] http://infolab.stanford.edu/gps/

[9] Shao B, Wang H, Li Y. Trinity: A distributed graph engine on a memory cloud[C]. SIGMOD, 2013.

[10] Bu Y, Howe B, Balazinska M, et al. HaLoop: Efficient iterative data processing on large clusters[J]. In Proc. of the VLDB Endowment, 2010, 3(1-2): 285-296.

[11] http://hama.apache.org/

[12] http://giraph.apache.org/

[13] http://spark.incubator.apache.org/

[14] Wang Z, Gu Y, Zimmermann R, et al. Shortest Path Computation over Disk-Resident Large Graphs Based on Extended Bulk Synchronous Parallel Methods[C]. DASFAA, 2013.

[15] Herschel M, Naumann F, Szott S, et al. Scalable iterative graph duplicate detection[J], TKDE, 2012, 24(11): 2094-2018.

[16] Valiant L G. A bridging model for parallel computation[J]. Communications of the ACM, 1990, 33(8): 103-111.

[17] Najork M, Wiener J L. Breadth-first crawling yields high-quality pages[C]. WWW, 2001.

[18] Karypis G, Kumar V. Multilevel graph partitioning schemes[C]. ICPP (3). 1995: 113-122.

[19] http://hadoop.apache.org/

[20] Kyrola A, Blelloch G, Guestrin C. GraphChi: Large-scale graph computation on just a PC[C]. OSDI, 2012.

[21] http://snap.stanford.edu/data/

[22] http://www.dis.uniroma1.it/challenge9/ download.shtml

[23] http://haselgrove.id.au/wikipedia.htm

[24] Ning Wang, Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu. TSH: Easy-to-be distributed partitioning for large-scale graphs [J]. Future Generation Computer Systems-The International Journal of eScience (FGCS), 2019, 101:804-818

[25] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, Jeffrey Xu Yu, Zhiqiang Wei. HGraph: I/O-efficient Distributed and Iterative Graph Computing by Hybrid Pushing/Pulling [J]. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2020, 33(5): 1973-1987