Accelerating *PageRank* in Shared-Memory for Efficient Social Network Graph Analytics

Baofu Huang, Zhidan Liu*, Kaishun Wu
College of Computer Science and Software Engineering, Shenzhen University, China
Email: {huangbaofu2018@email.szu.edu.cn, liuzhidan@szu.edu.cn, wu@szu.edu.cn}

***Abstract*—*PageRank* has a wide applications in online social networks and serves as an important benchmark to examine graph processing frameworks. Many efforts have been made to improve the computation efficiency of *PageRank* in shared-memory platforms, where a single machine can be sufficiently powerful to handle a large-scale graph. Existing methods, however, still suffer from synchronization issues and irregular memory accesses, which will deteriorate their overall performance. In this paper, we present an accelerated parallel *PageRank* computation approach, named *APPR*. By investigating the characteristics of parallel *PageRank* computation and degree distributions of social network graphs, *APPR* proposes a series of optimization techniques to improve the efficiency of *PageRank* computation. Specifically, a destination-centric graph partitioning scheme is designed to avoid the synchronization issues when concurrently updating the common vertex data. By exploiting power-law structure of social network graphs, *APPR* can intelligently schedule the computations of vertices to save computing operations. The vertex messages are adjusted by *APPR* for transmission to further improve the locality of memory accesses. Empirical evaluations are performed based on a set of large real-world graphs. Experimental results show that *APPR* significantly outperforms the state-of-the-art methods, with on average 2.4x speedup in execution time and 16.4x reduction in DRAM communication.**

***Index Terms*—PageRank, parallel graph computation, graph partitioning, shared-memory, social network graph**

I. Introduction

PageRank [25] and its variants [10] have been widely used for user ranking [31], friend recommendation [9], *etc.* in online social networks. The tremendous social network data are usually modelled as a graph, where users are presented as the vertices and relationships among users form the edges, and then fed into some graph processing frameworks for fast *PageRank* computation. Instead of running graph analysis in distributed frameworks, *e.g.*, GraphLab [11], [20], a recent trend is focused on the shared-memory platforms because of their low communication overheads compared to the expensive across-machine communications [23] and the increasing DRAM capacity of modern systems [29]. Nowadays, a single machine can be equipped with several powerful CPU cores and massive memories, which have enabled shared-memory processing of extremely large graphs.

Graph-parallel computation largely relies on the emerging vertex-centric programming model by encoding *PageRank* computations as the vertex programs, which run in parallel and interact along the edges [22]. The vertices exchange their *PageRank* values and update their own data based on the received messages in an iterative manner. Specifically, a vertex can either push its *PageRank* value to update its out-neighbors, or it can pull *PageRank* values of its in-neighbors to update its own value. Recent studies report that push is a better choice than pull in most cases, since it can avoid many unnecessary communications when most vertex data have converged [7].

* Corresponding author: Zhidan Liu.

Efficient graph-parallel computation, however, is challenging even on a single machine due to some inherent properties of graph algorithms, *e.g.*, poor locality that introduces irregular memory access patterns [29]. Even worse, push based message exchanges may incur race conditions, where multiple threads update the common vertex data concurrently. As a result, expensive synchronization primitives are required, and it will harm the performance and scalability [7], [17].

Many remarkable efforts have been made to improve the efficiency of parallel *PageRank* computation in shared-memory platforms [7], [17], [18], [28]. For example, [7] proposes a greedy switch mechanism between push and pull to reduce conflicts, while it is difficult to make a wise switch decision. [28] accelerates *PageRank* computation with some special hardware. PCPM [17], [18] propose a partition-centric processing abstraction with a data structure to store neighbors' updates for each vertex partition. Although PCPM can avoid synchronization issues, it needs to traverse the graph almost twice in each iteration and thus introduces extra overheads.

In this paper, we present an Accelerated Parallel PageRank computation approach, named *APPR*, which is motivated by two important observations. By carefully investigating the parallel *PageRank* computation patterns, we observe that the synchronization issue mainly steams from the write-conflicts when multiple vertex programs executed by different threads are trying to update the common vertex data concurrently. Although multiple threads process a large graph in parallel, each thread actually loops through its assigned vertices sequentially. Therefore, if vertices that may incur write-conflicts are assigned to the same partition that is further handled by one thread, the synchronization issues can be completely avoided. We also observe that graphs derived from real-world social networks typically have skewed power-law in-degree distributions, and further analysis on their graph structures shows that updating *PageRank* values of low-degree vertices is highly independent of these high-degree vertices, while value updating of the latter ones heavily rely on the former. Such an observation implies that high-degree vertices could join the *PageRank* computations later when most low-degree vertices have converged. This computation scheduling would save unnecessary computation and communication overheads.

Although above observations are attractive and useful, developing *APPR* out of them encounters a set of challenges. First, conflict-free graph partitioning is challenging since the skewed graphs may lead to partitions with substantial load imbalances. Second, explicitly scheduling vertex computations is difficult. In practice, the relations among vertices are extremely complex and their computations may be mutually dependent. Third, poor locality of *PageRank* computation causes inefficient memory accesses and harms the overall efficiency. Thus we propose *APPR* to tackle these challenges by exploiting the structures of social network graphs. *APPR* formally models the conflict-free and load balanced graph partitioning problem and proposes a heuristic scheme that is simple yet efficient. Instead of fine-grained vertex computation scheduling, *APPR* activates all high-degree vertices at some proper time, which is wisely determined by analyzing the communication patterns of a graph. In addition, *APPR* carefully adjusts the transmission orders of vertex messages to reduce random memory accesses.

The contributions of this paper are summarized as follows:

- We identify the synchronization problem in push-based parallel *PageRank* computation, and formally model the conflict-free and load balanced graph partitioning problem, which is proved to be NP-complete.
- We propose *APPR* to improve *PageRank* computation efficiency by exploiting characteristics of parallel *PageRank* computation and structures of social network graphs.
- We conduct extensive experiments with a set of large-scale real-world graphs. Experimental results show that *APPR* significantly outperforms state-of-the-art methods, with 2.4x speedup in execution time and 16.4x reduction in DRAM communication for social network graphs.

The rest of this paper is organized as follows. Section II presents the preliminary and motivation. The design of *APPR* is elaborated in Section III, and further evaluated in Section IV. We review the related work in Section V. Section VI finally concludes this paper.

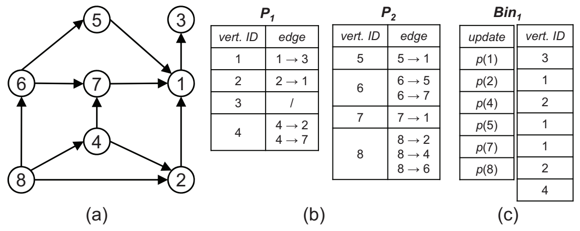
II. Preliminary and Motivation

In this section, we briefly introduce *PageRank*, and then analyze the characteristics of *PageRank* computation and social network graphs to motivate *APPR* design.

A. PageRank and Its Parallel Computation

PageRank was originally proposed to rank web pages [25], and nowadays has been widely used in online social networks for user ranking [31], friend recommendation [9], and so on. In addition, *PageRank* is frequently selected as a benchmark to examine various graph processing frameworks [11], [12], [20]. Thus, *PageRank* has become an important algorithm for both social network applications and graph-parallel computation.

Social network data can be modeled as a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, where users are presented as the vertices and the relationships among users form the edges. Then *PageRank* runs on the graph



P_1

vert. ID	edge
1	1 → 3
2	2 → 1
3	/
4	4 → 2 4 → 7

P_2

vert. ID	edge
5	5 → 1
6	6 → 5 6 → 7
7	7 → 1
8	8 → 2 8 → 4 8 → 6

Bin_1

update	vert. ID
p(1)	3
p(2)	1
p(4)	2
p(5)	1
p(7)	1
p(8)	2
	4

Fig. 1. A motivation example. (a) The sample graph. (b) Partitions of the graph. (c) The *bin* of partition P_1 to avoid synchronization issue.

G iteratively to determine *PageRank* value $p(v)$ of each vertex $v \in \mathrm{V}$. In each iteration, vertex data $p(v)$ is updated by the weighted sum of v's in-neighbors' latest values, *i.e.*,

$$p_{i+1}(v) = \frac{1-f}{|\mathrm{V}|} + f \times \sum_{u \in \mathrm{N}_i(v)} \frac{p_i(u)}{|\mathrm{N}_o(u)|}, \qquad (1)$$

where f is the damp factor, $\mathrm{N}_i(v)$ and $\mathrm{N}_o(u)$ represent v's in-neighbors and vertex u's out-neighbors, respectively [25]. *PageRank* typically iterates until the vertex data converge to within a specified tolerance ε.

The vertex-centric programming model [22] is usually used to parallelize *PageRank* computation, where a large graph is divided into partitions that are further processed by multi-threads in the shared-memory platform. In general, vertices are hash-partitioned [12] or range-partitioned [18] based on their IDs, and the edges are assigned along with their sources. For example, Figure 1(b) demonstrates the partitions of the sample graph in Figure 1(a), where partition P_1 needs to handle 4 edges and P_2 will process 7 edges, with imbalanced loads.

In each iteration, a thread loops through its assigned vertices with a user-defined vertex program, *i.e.*, *PageRank*, which instructs a vertex to exchange messages with neighbors and update its own data using Equation (1). A vertex will become inactive and stop exchanging messages when its data has converged. In graph-parallel computation, vertex v can either *push* $p(v)$ to update its out-neighbors, or *pull* v's in-neighbors' data to update $p(v)$. As graph computations processed, vertices will converge at different rates, leading to a rapidly shrinking active vertex set. Hence, push would be more efficient than pull, as it can do less work. Due to the vertex convergence, an optimization of *PageRank* computation is that vertices only push value difference between two consecutive iterations, called *delta*, to out-neighbors, so that the silence of converged vertices will not influence the computation of active vertices since their *deltas* can be treated as zeros [11], [29]. Algorithm 1 describes the *delta*-based *PageRank* computation.

The *delta*-based *PageRank* computation, however, is still not sufficiently efficient, because it may incur serious write-conflicts. In graph-parallel computation, vertices push updates to out-neighbors along edges, and the write-conflicts happen when multiple threads attempt to update the common destinations. As an example in Figure 1(b), vertex v_2 and v_5, which are processed by two different threads, may concurrently update v_1's value, and thus write-conflict happens. To guarantee the correctness of concurrent access to common out-neighbors,

Algorithm 1: *delta*-based *PageRank* Computation

```
Input: Graph $G = (V, E)$, tolerance threshold $\varepsilon$
Output: PageRank value $p(v)$ for each vertex $v \in V$
1  for $v \in V$ do
2      $p(v) = \frac{1}{|V|}$;
3      $\Delta(v) = \frac{p(v)}{|N_o(v)|}$;
4      $sum(v) = 0$;
5  $curV = V$;
6  $nextV = \emptyset$;
7  while $curV != \emptyset$ do
8      for $v \in curV$ in parallel do
9          for $u \in N_o(v)$ do
10             lock$\{sum(u) += \Delta(v)\}$;
11     for $v \in curV$ in parallel do
12         $temp = p(v)$;
13         $p(v) = \frac{1-f}{|V|} + f \cdot sum(v)$;
14         $\Delta(v) = \frac{p(v)-temp}{|N_o(v)|}$;
15         if $\Delta(v) > \varepsilon$ then
16             $nextV = nextV \cup \{v, N_o(v)\}$;
17     $swap(curV, nextV)$;
18     $nextV = \emptyset$;
```

synchronization primitives, *e.g.*, the *lock* operation in line 10 of Algorithm 1, are required. However, such synchronization operations will largely affect the performance and scalability of parallel *PageRank* computation.

To avoid the synchronization issue, recent works [5], [17], [18] propose to use contiguous memory spaces, called *bin*, for each partition to store the updates of their corresponding source vertices. Based on the bins, vertex v will first push its *delta* to the bins of partitions that contain v's out-neighbors, and then each destination vertex will gather in-neighbors' updates from bins to update its own data. Figure 1(c) illustrates the data structure of Bin_1 for partition P_1 in Figure 1(b). Although bins could break write-conflicts, they require *PageRank* computation to traverse a graph almost twice in each iteration, and thus introduce extra DRAM communication overheads, *i.e.*, the amount of data exchanged with main memory [18].

B. Motivation

The graphs of social networks share a similar characteristic as the web page graphs on degree distributions, where most vertices have relatively few neighbors while a few vertices have many neighbors. Such a property is called the *power-law degree distribution* that makes the graph-parallel computation especially challenging [11]. Figure 2(a) reports the in-degree distribution of a typical social network graph $orkut$ (see more details about the graph datasets in Section IV-A), which clearly demonstrates the skewed power-law in-degree distribution. Other social network graphs share similar in-degree distributions as $orkut$, we thus omit their results here. For graphs with such a structure, vertices with high in-degrees will suffer from extremely serious synchronization issues, since many vertices may concurrently update their data.

Although the power-law degree distribution causes troubles for efficient graph-parallel computation [11], they still provide

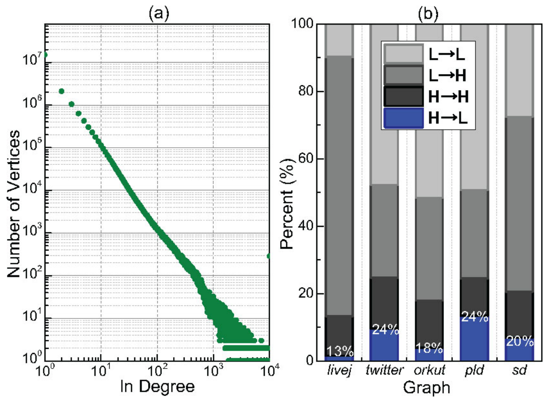


Fig. 2. Analysis of social network graphs. (a) The in-degree distribution of graph $orkut$. (b) The communication patterns among vertices of different types for 5 graphs, where $\rightarrow$ indicates the communication direction.

us opportunities to improve *PageRank* computation when we have carefully investigated its execution patterns. Specifically, we have the following two observations.

(1) Sequential vertex executions within a thread. In *PageRank* computation, although the whole graph is processed by multi-threads in parallel, vertex programs of the same partition are executed by the thread in sequence. Thus these vertices, which point to the common destination and meanwhile are assigned to the same partition, have no write-conflicts at all, since they update the common vertex orderly. For example in Figure 1(b), although both vertex v_5 and v_7 point to vertex v_1, they push their *deltas* to update destination v_1 sequentially by the same thread. Therefore, if we can assign the in-neighbors of a vertex v to the same partition, no more synchronization primitives are required for correctly updating $p(v)$.

(2) Imbalanced communication patterns between vertices of high-degree and low-degree. In this paper, we define that a vertex v is called as *high-degree vertex* (denoted as H vertex) if its in-degree $|N_i(v)|$ is larger than a threshold λ; Otherwise, v is called as *low-degree vertex* (denoted as L vertex). If vertex v points to vertex u, we say v will communicate messages to u. We study the communication patterns among vertices of different types, and show the results of 5 graphs in Figure 2(b). In this study, we set the threshold λ as the average in-degree of each graph. From Figure 2(b), we see that most communications are originated by L vertices. The percentages of communications initiated by H vertices for all the five graphs are $< 25\%$. In particular, the communications belonging to $H \rightarrow L$ is as low as $2\% \sim 9\%$. These statistics imply that H vertices have limited impacts on data updates of L vertices. Instead, their data largely depend on the *PageRank* values of L vertices. This observation motivates us to schedule the activities of H vertices later to avoid unnecessary computations and communications in early iterations.

As a concrete example in Figure 1(a), the *PageRank* value $p(1)$ of H vertex v_1 (with in-degree as 3, which is higher than the average in-degree 1.4) only decides the data update of v_3. Therefore, we can schedule the computations of v_1 and v_3 after the convergences of other vertices in the graph.

Challenges. Developing techniques out of above insights to improve *PageRank* computation, however, entails several challenges. First, how to partition a social network graph to avoid synchronization issues while retaining the load balances among threads is difficult. The power-law degree distributions of social network graphs could lead to substantial work imbalances, where some partitions may undertake much more loads than others. Second, the arrangement of vertex computation orders remains challenging because of the complex topology of social networks, where links among users are irregular and unpredictable. Third, the poor locality of graph computations causes inefficient memory accesses, and it becomes especially challenging for the large-scale social network graphs.

III. The Design of *APPR*

In this section, we present the overview of *APPR*, and then elaborate each component in the following subsections.

A. Overview of APPR

The system architecture of *APPR* is illustrated in Figure 3. At a high level, *APPR* takes a raw social network graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ as the input for parallel *PageRank* computation, and outputs *PageRank* values of all vertices to support various social network applications [10]. During loading a graph G, *APPR* scans G to derive the total numbers of vertices $|\mathrm{V}|$ and edges $|\mathrm{E}|$, average in-degree $\bar{d}$, in-neighbors $\mathrm{N}_i(v)$ and out-neighbors $\mathrm{N}_o(v)$ for each vertex v, and labels v as H or L vertex according to its in-degree and the given threshold λ.

For conflict-free *PageRank* computation, *APPR* invokes the destination-centric graph partitioning module to divide graph G into m partitions while guaranteeing their load balances. In particular, the in-neighbors of a vertex will be assigned to the same partition. A partition P_j maintains a list of destination vertices $P_j.dst$, a list of corresponding source vertices $P_j.src$, and the associated edges. During *PageRank* computation, vertices are scheduled according to their in-degrees. Specifically, L vertices compute for the initial iterations, and H vertices are activated at certain time when most L vertices have converged. Such a schedule will save a lot of unnecessary computations and communications for these H vertices. Benefiting from the shared memory, *APPR* stores all vertices' updates (*i.e.*, *delta*) and *PageRank* values as two globally accessible vectors. Thanks to the graph partitioning scheme, source vertices of a partition can concurrently push their updates to the common destinations, and meanwhile destination vertices of a partition can calculate their latest *PageRank* values simultaneously. All operations are conflict-free. As an improvement on the locality of *PageRank* computation, *APPR* proposes a message controller module that allows a vertex to continuously send its update and status messages to each of its out-neighbors. This adjustment promotes the locality of memory accesses and will not affect the computation results.

B. Destination-Centric Graph Partitioning

For simplicity, some existing graph partitioning schemes primarily rely on either hash-partitioning [11], [12] or range-partitioning [5], [17], [18] based on the IDs of all vertices.

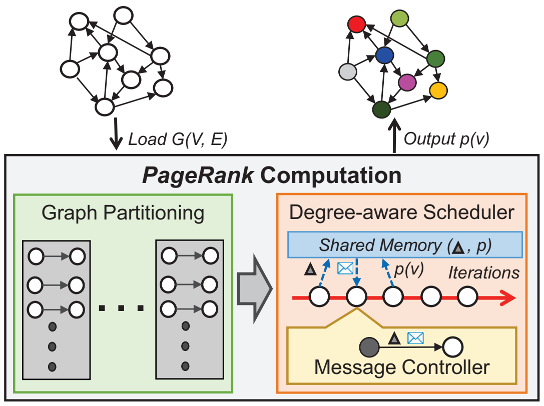


Fig. 3. The system architecture of *APPR*.

Advanced graph partitioning tools, *e.g.*, METIS [14], mainly aim to divide a large graph into partitions of nearly equal sizes, while minimizing the total vertex/edge-cuts [8], [19]. These schemes, however, do not consider the write-conflict issues in shared-memory platforms, and thus may not be suitable for efficient *PageRank* computation. To avoid the synchronization issues, we should assign the in-neighbors of a vertex to the same partition. A straightforward approach is to equally dividing all vertices into m groups and assigning their in-neighbors to the corresponding partitions. This approach, however, will result in serious load imbalances, due to the skewed in-degree distributions of graphs.

Since communication is usually more expensive than computation in the graph-parallel computation [11], we thus use the number of edges assigned to partitions as the measure of loads. Thus, we will divide the edges of a graph into partitions of nearly equal sizes. Formally, we define the conflict-free and load-balanced graph partitioning problem as:

Definition 1. ***(Destination-centric graph partitioning problem)*** *Given a social network graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$*, a graph partitioning scheme should divide* G *into* m *partitions* $\mathbb{P} = \{P_j, j = 1, 2, \cdots, m\}$*, which minimizes the variance of partition sizes, i.e.,*

$$\min \frac{1}{m} \sum_{j=1}^{m} (|P_j| - \mu)^2, \tag{2}$$

where $|P_j|$ *represents the number of edges assigned to partition* P_j *and* $\mu = \frac{|\mathrm{E}|}{m}$ *is the desired partition size. For conflict-free parallel PageRank computations, a destination vertex is assigned to one partition only, while a source vertex can be replicated among multiple partitions.*

This problem can be reduced as the well-known *number partitioning problem*, which is NP-complete [15].

Theorem 1. *The conflict-free and load-balanced graph partitioning problem is NP-complete.*

Proof. We prove this theorem by reducing from the NP-complete number partitioning problem. A number partitioning problem can be described as follows: Given a set $\mathcal{S}$ of positive integers, the goal is to find a division of $\mathcal{S}$ into two subsets $\mathcal{S}_1$

Algorithm 2: Destination-centric Graph Partitioning

```
Input: Graph $G = (V, E)$, number of partitions $m$
Output: Partitions $\mathbb{P} = \{P_j, j = 1, 2, \cdots, m\}$
1 $\mu = \frac{|E|}{m}$, $j = 1$;
2 for $v \in V$ do
3     if $P_j$ and $N_i(v)$ meet Equation (3) then
4         $j = j + 1$;
5     $P_j.dst = P_j.dst \cup \{v\}$;
6     $P_j.src = P_j.src \cup N_i(v)$;
```

and $\mathcal{S}_2$ such that the sum of the numbers in $\mathcal{S}_1$ equals the sum of the numbers in $\mathcal{S}_2$. For a given number partitioning problem, we can transform it to an instance of our problem. We consider a special case of our problem, where we divide the graph into $m = 2$ partitions. For each vertex $v \in V$, we can transform it to a positive integer that is the number of v's in-neighbors, *i.e.*, $|N_i(v)|$. Then all vertices are transformed to a set $\mathcal{S}$ of positive integers, and the graph partitioning problem is to divide $\mathcal{S}$ into two partitions P_1 and P_2. The total number of in-neighbors in P_1 equals the total number of in-neighbors in P_2. We find that the special case of our problem is a number partitioning problem, which is known NP-complete [15]. Therefore, our problem is also NP-complete. ■

Although some dynamic programming methods can be used to find a feasible solution for the number partitioning problem [15], these methods need to traverse the graph multiple times, and thus are associated with high computation overheads. Therefore, our problem cannot be efficiently solved by existing algorithms that are proposed for number partitioning problem.

In practice, we usually expect the pre-processing time for graph partitioning is as short as possible. Since we know the sum of numbers of each subset in advance, *i.e.*, the average partition size $\mu = \frac{|E|}{m}$, we thus propose a heuristic approach to partition G by scanning all vertices and edges only once. The key idea is that partition P_j will continuously accommodate in-neighbors $N_i(v)$ of vertex v until that including $N_i(v)$ into P_j makes the partition size $|P_j|$ deviate from μ much more than excluding $N_i(v)$ from P_j, *i.e.*,

$$|P_j| + |N_i(v)| - \mu > \mu - |P_j|. \quad (3)$$

The pseudocode of our graph partitioning scheme is presented in Algorithm 2. It loops all vertices and heuristically adds their in-neighbors to partitions such that the contained edges of all partitions are as nearly equal as possible.

From Algorithm 2, each vertex v as a source may be assigned and replicated to multiple partitions (*i.e.*, $P_j.src$), while as a destination it will be assigned to only one partition (*i.e.*, $P_j.dst$). The replicas of a source vertex v can simultaneously access the global *delta* vector and push the same $\Delta(v)$ to v's destination vertices with no conflicts. As a destination, vertex v will also receive *delta* values from its in-neighbors to update its *PageRank* value solely, with no write-conflict. Since $|E|$ is pretty large while the number of partitions m is much smaller, Algorithm 2 could return a valid solution. In extreme cases, the in-neighbors of a vertex, which has a relatively large $|N_i(v)|$ (*e.g.*, $> 2\mu$), can be distributed among several partitions.

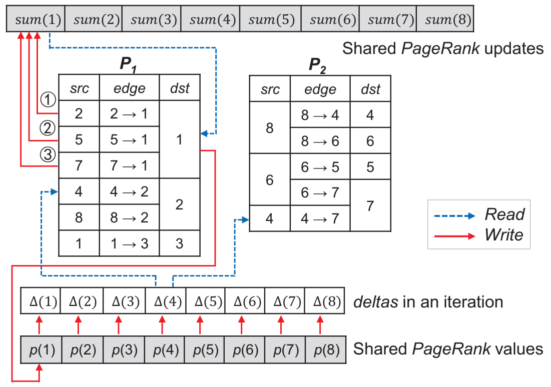


Fig. 4. Destination-centric graph partitioning results for graph of Figure 1(a). The numbers in the circles indicate the execution orders within a partition.

Figure 4 demonstrates the partitioning results for the sample graph in Figure 1(a), where the graph is divided into partition P_1 and P_2. Such partitions can benefit both pushing *deltas* and calculating *PageRank* values, with no conflicts at all. For example, we observe that source vertex v_4 is assigned to both partitions, and its replicas in P_1 and P_2 can concurrently read and then push $\Delta(4)$ to its destination vertex v_2 and v_7. As another example, vertex v_1's in-neighbors $N_i(1) = \{v_2, v_5, v_7\}$ are all assigned to partition P_1, and thus they can push their *deltas* to update v_1's partial sum $sum(1)$ orderly by the thread that processes P_1. On the other hand, vertices in both partitions can simultaneously read their corresponding sum to calculate new *PageRank* values, and derive the *deltas* by comparing with previous *PageRank* values of last iteration. Those *deltas* are used for *PageRank* computation in the next iteration.

C. Degree-Aware Computation Scheduler

During *PageRank* computation, vertices will converge with different rates. In general, the L vertices would converge much faster than the H vertices, whose values heavily depend on the computation results of many L vertices. As a concrete example, we run Algorithm 1 on social network graph $orkut$ and record convergence statuses of all vertices. Figure 5 plots the vertex convergences for graph $orkut$, where almost all L vertices can converge within 10 iterations while most H vertices need to compute for 15 iterations. After five iterations, about 80% L vertices have converged while only 20% H vertices have converged. By further considering imbalanced communication patterns among H and L vertices as shown in Figure 2(b), it suggests that an intelligent vertex computation scheduling should accelerate the convergence of *PageRank* computation. More specifically, L vertices should be computed ahead of H vertices.

It is beneficial to schedule vertex computations. On one hand, since H vertices rarely affect *PageRank* computations of L vertices, they can keep inactive in the early iterations to avoid unnecessary computations and communications. On the other hand, the convergences of L vertices would accelerate the convergence rate of H vertices. For example, if vertex v's in-neighbors have already converged, then $p(v)$ can be finalized immediately.

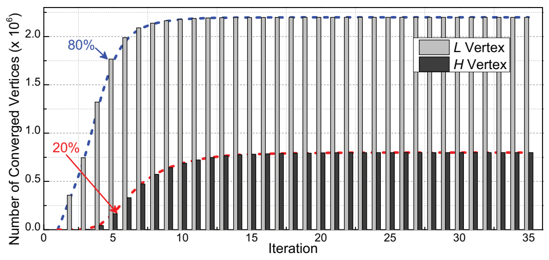


Fig. 5. The number of converged vertices for graph *orkut* in each iteration.

It is feasible to schedule vertex computations while achieving the correct *PageRank* computation results. This is because for graph algorithms especially *PageRank*, the vertex data $p(v)$ can be determined only by the initial value and the update messages, regardless of the orders of these messages [30]. An expected schedule strategy is that the calculation is carried on for L vertices first, and an H vertex is scheduled to compute when its in-neighbors have all converged. Such a strategy needs to track the statuses of in-neighbors for each H vertex, and thus is prohibited due to tremendous tracking cost.

Instead, *APPR* adopts a simple yet efficient batch scheduling strategy, which makes L vertices be active at early iterations and schedules all H vertices to join computations when most L vertices have converged. Considering the communication patterns among vertices, when H vertices are inactive, some L vertices that rely on $H \to L$ communications cannot achieve the true convergences. Only these vertices that merely depend on $L \to L$ communications can make successful convergences. We conservatively estimate the number of such vertices as $N_\ell = \frac{|\mathrm{E}_{L\to L}|}{\bar{d}}$, where $|\mathrm{E}_{L\to L}|$ represents the number of edges linking two L vertices and $\bar{d}$ is the average in-degree. As an explicit indicator, when N_ℓ vertices have converged, *APPR* will activate all H vertices to join the *PageRank* computations. H vertices will push their delayed update messages to these L vertices, which rely on $H \to L$ communications. Finally, all vertices can derive the results after a number of iterations.

D. Message Controller

The graph-parallel computation usually follows the *gather-apply-scatter* (*GAS*) model [11], where a vertex firstly gathers updates from its in-neighbors, applies these updates to calculate a new value, and then scatters its update to the out-neighbors. Furthermore, if this vertex has not converged, it will send status messages to its out-neighbors by keeping them be active for receiving updates in the next iteration [11]. Similarly, *delta*-based *PageRank* computation shown in Algorithm 1 also follows this model. In each iteration, vertex v pushes its *delta* to out-neighbors (*i.e.*, line 10), updates $p(v)$ using the weighted sum (*i.e.*, line 13), and then pushes status messages to out-neighbors if v is not converged (*i.e.*, line 16). Figure 6(a) illustrates this procedure. This model, however, will traverse all edges two times, resulting in poor spatial and temporal locality of memory accesses [5], [18].

To improve the memory access efficiency, we propose the message controller that allows a vertex to push *delta* and status messages at the same time. As shown in Figure 6(b),

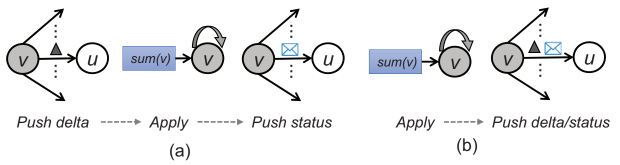


Fig. 6. The operations in each iteration. (a) The *GAS* model. (b) The message controller in *APPR*.

during each iteration, vertex v will calculate new $p(v)$ value based on the weighted sum, and then successively push *delta* and status messages to its out-neighbors. Since vertex v continuously operates on its out-neighbors that are cached, message controller can boost cache hit rate, leading to better memory accesses and improved computation efficiency.

Although *APPR* changes the message transmission orders, it will not affect the *PageRank* computation results. By exploiting an example edge $v \to u$, we compare message controller with *GAS* model to analyze their execution procedures. If both v and u have converged, there will be no difference between *GAS* model and message controller. Similarly, if they both have not converged, u can receive all the *delta* and status messages in both models. We analyze the other two cases as follows:

- *u converged while v not.* In both models, u will be activated by v to join computation in the next iteration. In the message controller, u will not push *delta* messages to its out-neighbors once it has converged in the next iteration. In the *GAS* model, however, u will push such messages before its convergence. Since u has converged, its *delta* should be sufficiently small, thus u's out-neighbors can omit $\Delta(u)$ and safely update their data, with no error introduced in the *delta*-based *PageRank*.
- *v converged while u not.* In *GAS* model, v will push *delta* message to u before it becomes converged. In contrary, v will firstly update $p(v)$ and then be converged, without pushing *delta* to u when the message controller is adopted. Similar as the above case, $\Delta(v)$ is sufficiently small and can be safely omitted by u for computation.

E. Put It Together

Algorithm 3 presents the details of *APPR* that incorporate above optimizations. Overall, *APPR* runs on the destination-centric graph partitions $\mathbb{P}$ and outputs the *PageRank* values for all vertices. At the beginning, *APPR* only sets L vertices as the active vertices (line 8) and lets them push the initial *delta* values (line 9-12). Then *APPR* iteratively updates each vertex's *PageRank* value, which is executed in parallel for all partitions (line 13-29). Specifically, in each iteration, each active vertex v updates $p(v)$ based on the received *delta* sum from last iteration, and then calculates the diffidence $\Delta(v)$ of *PageRank* values between two consecutive iterations (line 14-18). If the change is larger than a threshold ε, v will be active in the next iteration, and meanwhile v will let its out-neighbors $\mathrm{N}_o(v)$ keep active as well and push $\Delta(v)$ to them (line 20-24); Otherwise, v has converged. At the end of each iteration, *APPR* will check whether there are sufficient converged L vertices, and it will activate the H vertices when more than N_ℓ vertices have converged (line 25-27). This schedule is executed

Algorithm 3: Accelerated Parallel *PageRank* (*APPR*)

```
Input: $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, $\mathbb{P} = \{P_j, j = 1, 2, \cdots, m\}$, $\varepsilon$, $N_\ell$
Output: PageRank value $p(v)$ for each vertex $v \in \mathrm{V}$
1  $\mathrm{V}_L = \emptyset$;  $\mathrm{V}_H = \emptyset$;  $flag = false$;
2  for $v \in \mathrm{V}$ do
3      $p(v) = \frac{1}{|\mathrm{V}|}$;  $\Delta(v) = \frac{p(v)}{|\mathrm{N}_o(v)|}$;
4      if $|\mathrm{N}_i(v)| < \lambda$ then
5          $\mathrm{V}_L = \mathrm{V}_L \cup \{v\}$;
6      else
7          $\mathrm{V}_H = \mathrm{V}_H \cup \{v\}$;
8  $nextV = \emptyset$;  $curV = \mathrm{V}_L$;      // active vertex set
9  for $P_j \in \mathbb{P}$ in parallel do
10     for $v \in curV$ & $v \in P_j.src$ do
11         for $u \in \mathrm{N}_o(v)$ do
12             $cursum(u) += \Delta(v)$;  // initial delta
13 while $curV != \emptyset$ do
14     for $P_j \in \mathbb{P}$ in parallel do
15         for $v \in curV$ & $v \in P_j.dst$ do
16             $temp = p(v)$;
17             $p(v) = \frac{1-f}{|\mathrm{V}|} + f \cdot cursum(v)$;
18             $\Delta(v) = \frac{p(v) - temp}{|\mathrm{N}_o(v)|}$;
19     for $P_j \in \mathbb{P}$ in parallel do
20         for $v \in curV$ & $v \in P_j.src$ do
21             if $\Delta(v) > \varepsilon$ then
22                 $nextV = nextV \cup \{v, \mathrm{N}_o(v)\}$;
23                 for $u \in \mathrm{N}_o(v)$ do
24                     $nextsum(u) += \Delta(v)$;
25     if $!flag$ & $(|\mathrm{V}_L| - |nextV|) \geq N_\ell$ then
26         $flag = true$;
27         $nextV = nextV \cup \mathrm{V}_H$;          // activation
28     $cursum = nextsum$;  $nextsum \leftarrow \mathbf{0}$;
29     $swap(curV, nextV)$;  $nextV = \emptyset$;
```

only once, and is ensured by indicator $flag$. *APPR* terminates when there are no more active vertices.

Discussion. *PageRank* converges when vertex data do not change remarkably [3], [25]. For practical uses, existing implementations in the popular graph-parallel processing frameworks [11], [12], [16], [20], however, usually execute *PageRank* for a specified number of iterations. In this paper, we present *APPR* with a set of optimization techniques to improve the efficiency of computation and memory accesses of parallel *PageRank* computation. These optimizations will not harm the convergence of *PageRank*. Instead, we find that *APPR* could accelerate the convergence rates of *PageRank* on most real-world graphs, as demonstrated from the experimental result in Figure 9 of Section IV-C.

Furthermore, the optimizations proposed by *APPR* can be generalized to a wide range of applications called as *sparse matrix-vector* (SpMV) multiplication [17], [18]. In fact, many graph algorithms, including *PageRank* [3], [25], can be modeled as a series of SpMV operations. For example, *PageRank* computation can be rewritten in the SpMV form as follows:

$$\mathbf{p}_{i+1}^T = f\mathbf{p}_i^T\mathbf{A} + (1-f)\mathbf{p}_i^T\frac{\mathbf{e}\mathbf{e}^T}{|\mathrm{V}|}, \tag{4}$$

TABLE I
REAL-WORLD GRAPH DATASETS (*M: million*)

Graph	Description	#vertices	#edges	$\bar{d}$	Disk size
livej	Social network	$7.5\,M$	$112.3\,M$	15	$1.6\,GB$
twitter	Social network	$21.3\,M$	$265.0\,M$	12	$5.2\,GB$
orkut	Social network	$3.0\,M$	$106.3\,M$	35	$1.6\,GB$
pld	Web pages	$42.9\,M$	$623.1\,M$	15	$10.9\,GB$
sd	Web pages	$94.9\,M$	$1937.5\,M$	20	$34.4\,GB$

where f is the damp factor, $\mathbf{p}_i$ is a column vector that stores *PageRank* values of all vertices in the i-th iteration, $\mathbf{A}$ is the adjacency matrix of the input graph, and $\mathbf{e}$ is a unit column vector [28]. SpMV is communication-bounded, and thus the techniques of *APPR* to reduce communications can be extended to optimize SpMV as well.

IV. Performance Evaluation

In this section, we conduct extensive experiments to evaluate the performance of *APPR* on large-scale real-world graphs.

A. Experimental Setup

We conduct empirical experiments on a powerful server, which is equipped with two 10-core Intel(R) Xeon(R) E5-2630 v4 processors @2.20GHz and $192\,GB$ memory, running CentOS Release 6.9. For performance evaluations, we compare *APPR* with 3 baseline methods on a set of large-scale graphs.

Graph datasets. The input graphs used in our experiments are summarized in Table I. All the graphs consist of millions of vertices and edges. Specifically, $livej$, $twitter$, and $orkut$ are follower graphs from social networks; pld and sd are web page graphs obtained by the web crawlers. The average in-degree $\bar{d}$ of the five graphs are 15, 12, 35, 15, and 20, respectively. The storage sizes of these graphs range from $1.6\,GB$ to $34.4\,GB$. All graphs are available from Network Repository [1][26].

Baseline methods. We compare *APPR* with the following implementations of parallel *PageRank* computation.

- *PullPR* implements *PageRank* in the pull direction, where each vertex v pulls *delta* values of its in-neighbors to update its own *PageRank* value $p(v)$. The traditional range-partitioning scheme is adopted to divide an input graph. Because a vertex v will pull data from its in-neighbors no matter whether they have converged or not, *PullPR* thus omits the convergence statuses of vertices.
- *PushPR* implements *PageRank* in the push direction as shown in Algorithm 1, where each vertex v pushes its *delta* and status messages to out-neighbors and updates its own *PageRank* value $p(v)$ based on the accumulated *delta* from in-neighbors. For parallel *PageRank* computation, vertices are range-partitioned as well and synchronization primitives are used to resolve the write-conflicts.
- *PCPM* is the state-of-the-art method that optimizes the parallel *PageRank* computation based on a partition-centric processing methodology, which uses extra memory spaces called bin as the intermediate storage of source vertices' updates to avoid write-conflicts [17], [18].

Implementation details. We realize *APPR* and the baseline methods using C++ and compile them with g++ 4.8.4 at the

TABLE II
COMPARISONS ON EXECUTION TIME (*Unit: seconds*)

Graph	***PullPR***	***PushPR***	***PCPM***	***APPR***	*Ratio*
livej	1.4	2.5	4.0	1.0	$1.4 \sim 4.0$
twitter	5.6	14.6	7.5	3.7	$1.5 \sim 3.9$
orkut	1.9	3.0	1.7	0.5	$3.4 \sim 6.0$
pld	29.5	59.5	13.6	11.6	$1.2 \sim 5.1$
sd	94.9	99.8	35.1	29.5	$1.2 \sim 3.4$

highest optimization level. Specifically, we use the *PullPR* implementation from UC Berkeley GAP benchmarks [4], and implement *PushPR* following the Algorithm 1. In addition, we directly adopt the open-sourced PCPM implementation [17], [18] for comparisons, and adjust its parameters of bin settings to fit our hardware. Since both *PullPR* and *PCPM* do not consider the convergences of vertices, we thus make them run the same number of iterations as *PushPR*. We test these baseline methods and adopt the configurations that achieve their best performances. To configure *APPR*, we empirically set λ as the average in-degree $\bar{d}$ of the input graph to label each vertex as H or L vertex. We also calculate the activation indicator N_ℓ for each graph accordingly. For all methods, we set the damp factor $f = 0.15$, the convergence threshold $\varepsilon = 10^{-3}$, and the default number of partitions $m = 20$. The average results of five executions are reported.

B. Overall Performance

Execution time. We compare the execution time of the four methods over all graphs in Table II, where we list the *ratios* between the execution time of each compared approach and the execution time of *APPR* in the last column. Although *PullPR* does not consider vertex convergences and thus will generate abundant unnecessary messages, it still runs a bit faster than *PushPR*, which suffers from serious synchronization issues. It suggests that the overheads caused by the synchronization primitives overweight the unnecessary communication costs. Thanks to the data structure bin, *PCPM* can avoid synchronization issues and thus runs much faster than *PushPR* on most graphs except $livej$, which is an extremely skewed graph.

Among the four methods, *APPR* has the best performance, with obvious speedup on the execution time as shown in the last column of Table II. Overall, *APPR* outperforms the baseline methods with speedup 1.2x $\sim$ 6.0x. We find that *APPR* has greater advantages on social network graphs (*i.e.*, $livej$, $twitter$, and $orkut$) than web page graphs (*i.e.*, pld and sd). The reason might be that social network graphs are more skewed than web page graphs, and *APPR* benefits more from such graph structures. According to our statistics, the percentages of H vertices are only 4%, 9%, 7%, 15%, and 10% for the $livej$, $twitter$, $orkut$, pld and sd, respectively. Compared to the state-of-the-art method, *APPR* improves *PCPM* in the execution time over the five graphs with speedup 1.2x $\sim$ 4.0x, and the average speedup is as high as 2.4x. In particular for social network graphs, *APPR* outperforms *PCPM* with an average speedup as high as 3.2x.

TABLE III
COMPARISONS ON PRE-PROCESSING TIME (*Unit: seconds*)

Method	$livej$	$twitter$	$orkut$	pld	sd
PCPM	0.04	0.34	0.08	0.20	0.54
APPR	0.11	0.50	0.18	0.55	1.39

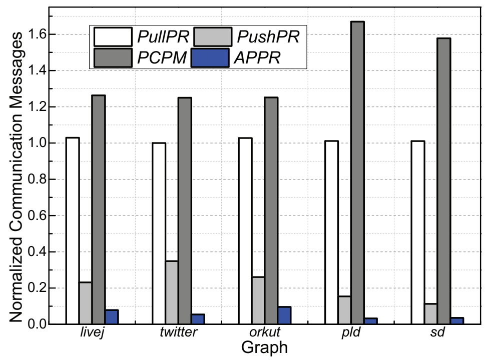


Fig. 7. The normalized communication messages.

Pre-processing time. Since both *PushPR* and *PullPR* almost have no pre-processing overheads, we thus only present the pre-processing time of *APPR* and *PCPM* in Table III. Specifically, *PCPM* needs to divide the graph and construct the bin for each partition, and the heuristic graph partitioning of *APPR* also takes time. Table III shows the pre-processing time of both methods is proportional to the graph size, *i.e.*, a larger graph needs more pre-processing time. *APPR* spends slightly more time to pre-process a graph than *PCPM*. From Table II and Table III, the total time of *APPR* (including both execution time and pre-processing time) is still much smaller than the total time of *PCPM*. In fact, for most graphs pre-processing time is negligible when compared to execution time.

Communication messages. In graph-parallel computation, vertices need to exchange messages along edges for updating their own data. Such messages will cause communication costs among threads even in the shared-memory platforms. Similar as PCPM [17], [18], the communication costs indicate the amount of data exchanged with main memory. In Figure 7, we compare the four methods on the normalized communication messages, which is total messages normalized by total edges and number of iterations. In general, we find that push-based methods, *i.e.*, *PushPR* and *APPR*, usually have fewer messages than pull-based methods, *i.e.*, *PullPR*. This is because converged vertices in push mode will stop propagating messages. Instead, *PullPR* has about 1 message per edge per iteration. Since a vertex in *PCPM* needs to push its updates to bin and gather neighbors' updates from bin, it has the largest normalized communication messages among the four methods, approaching 2 on all graphs. *APPR* further improves *PushPR* by avoiding H vertices' early communication costs.Therefore, *APPR* has the smallest normalized communication messages. In particular, *APPR* averagely improves *PCPM* by 16.4x in communication messages for the social network graphs.

C. Evaluation of APPR Design

In this subsection, we will conduct some micro-benchmark experiments to examine the optimization designs of *APPR*.

Impact of destination-centric graph partitioning. We study the impact of partition number m on *APPR*, and present the results in Figure 8. In general, the initial increase of num-

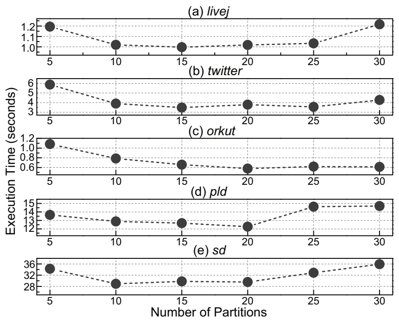


Fig. 8. Impact of number of partitions.

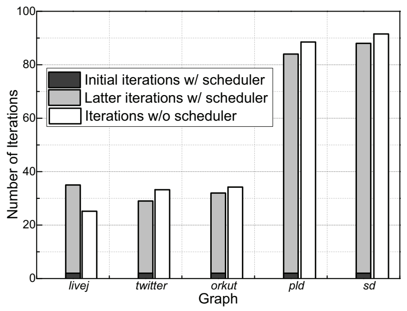


Fig. 9. Comparison on total iterations.

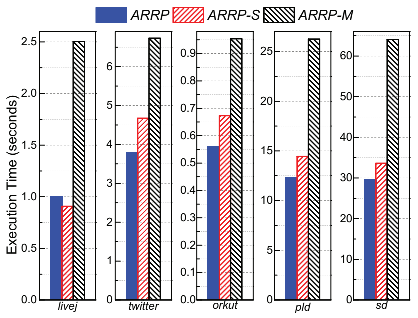


Fig. 10. Evaluation of *APPR* modules.

TABLE IV
VARIANCES OF DIFFERENT GRAPH PARTITIONING SCHEMES

Graph	**Vertex-centric**	**Edge-greedy**	***APPR***
livej	1.0×10^{14}	7.9×10^{8}	3.6×10^{8}
twitter	1.3×10^{15}	7.1×10^{8}	1.6×10^{8}
orkut	2.1×10^{13}	2.6×10^{7}	1.5×10^{7}
pld	1.7×10^{12}	4.1×10^{5}	3.3×10^{5}
sd	1.6×10^{14}	2.6×10^{8}	1.1×10^{8}

ber of partitions can accelerate parallel *PageRank* computation, while more partitions (*e.g.*, > 20) may even slow down the execution. Such a trend exists for all the five graphs. We find that *APPR* can achieve the best performance for all graphs when we set $m = 20$. This might be that our server has 20 CPU cores, and > 20 partitions will cause some threads process more partitions of data, resulting in load imbalance that harms the overall performance of *APPR*.

We compare our heuristic graph partitioning scheme with two alternative graph partitioning schemes, *i.e.*, *vertex-centric* and *edge-greedy*. Specifically, *vertex-centric* equally divides vertices into partitions and assigns the edges along with their sources. The *edge-greedy* scheme works in a greedy manner. It constantly assigns vertices and their associated edges to a partition P_j until the size $|P_j|$ exceeds the expected partition size μ. We compare them and present their variances of partition sizes in Table IV, where each scheme divides a graph into 20 partitions. Due to the power-law in-degree distribution, *vertex-centric* schemes has the largest variance, *i.e.*, the most serious load imbalance. Compared to the *edge-greedy* scheme, *APPR* can heuristically decide whether putting a vertex and its associated edges to current partition or not, so that to minimize the variance of partition sizes. As a result, *APPR* improves *edge-greedy* with an average reduction of variance by 140%. In particular, *APPR* achieves the largest reduction by 3.5x on graph $twitter$, which is a typical social network graph.

Impact of degree-aware scheduler. We compare *APPR* with the version that disables degree-aware scheduler (*APPR*-S), and present the comparison in Figure 10. Indeed, the degree-aware scheduler benefits the parallel *PageRank* computation. *APPR* outperforms *APPR*-S on almost all graphs except $livej$. The gap on $livej$ is quite small as 0.04 second. For the other graphs, degree-aware scheduler brings $14\% \sim 24\%$ improvements on execution time, and the largest improvement 24% is derived from $twitter$ graph.

Since the scheduler will compute L vertices at the initial iterations and activate H vertices later. We thus decompose all iterations of *APPR* into two parts, *i.e.*, *initial iterations* that only involves L vertices and *latter iterations* that involves both L and H vertices, and compare with the total iterations of *APPR*-S in Figure 9. For graph $livej$, few H vertices take a long time to be converged with *APPR*. Except $livej$, we see clear reductions on total iterations of *APPR*, with an average reduction of 3 iterations. The comparison results demonstrate that an intelligent vertex computation scheduling indeed accelerates the convergences of all vertices.

Impact of message controller. *APPR* adjusts the transmission orders of *delta* and status messages to improve graph locality. We examine this optimization design by comparing *APPR* with the version that disables message controller (*APPR*-M). Figure 10 shows that *APPR*-M doubles the execution time for all graphs when compared to *APPR*. The message controller allows each vertex v to successively push *delta* and status messages to its out-neighbors $\mathrm{N}_o(v)$. Therefore, the cached out-neighbor data by the operation of pushing *delta* messages could be reused by the latter operation of pushing status messages. The traditional implementation, however, separates the two operations, which leads to inefficient memory accesses. On average, the message controller module accelerates *PageRank* computation by 104%.

V. Related Work

Initially proposed for ranking web pages [25], nowadays *PageRank* and its variants [3] have been widely used for various graph analysis of online social networks, biology, neuroscience, physics, and *etc.* [10]. In particular for social networks, *PageRank* can be used to find the leaders of social network community [31] and recommend friends to users by analyzing the corresponding follower graphs [9]. Besides, *PageRank* is usually selected as the benchmark to examine various graph-parallel processing frameworks [11], [12], [29].

There exist tremendous efforts that have been made to improve the *PageRank* computation [6]. Some works target to derive fast *PageRank* approximation by exploiting techniques like random walk [24] and Monte Carlo methods [2]. A more attractive direction is to parallelize *PageRank* computation with advanced hardware, *e.g.*, GPUs [13], ASIC [28], or FPGA [27]. With the emergence of "think like a vertex", *PageRank* has been implemented in various graph-parallel processing frameworks [22], which accelerate the *PageRank* computation of graphs with billions of vertices on the clustered machines. These implementations, however, either merely derive approximation results or rely on some expensive hardware or clusters.

Instead of running *PageRank* in the distributed frameworks, a recent trend is to customize and optimize graph analytics (*e.g.*, *PageRank*) on shared-memory platforms because of their low communication costs and the increasing memory capacity [17], [18], [23]. Scott Beamer *et al*. propose a cache blocking technique that restricts the range of randomly accessed vertices to increase graph locality of *PageRank* computation [5]. The extremely sparse nature of social network graphs, however, reduces the reuse rate of cached vertex data. PCPM [17], [18] proposes a partition-centric processing abstraction to optimize parallel *PageRank* computation, while it still needs to traverse the entire graph almost twice in each iteration, leading to inefficient computation. Ma *et al*. have designed a general graph processing platform to efficiently process large graphs with the hybrid CPU-GPU on a single machine [21]. Different from existing works, we have optimized both computation and communication of *PageRank* by exploiting the characteristics of its parallel-computation patterns and the power-law structures of social network graphs.

VI. Conclusion

In this paper, we present *APPR* to accelerate parallel *PageRank* computation in the shared-memory platforms for large-scale graphs. By investigating the characteristics of parallel *PageRank* computation and the power-law degree distributions of social network graphs, *APPR* proposes a set of optimizations, including destination-centric graph partitioning to avoid synchronization issues, degree-aware computation scheduler to reduce unnecessary operations, and message controller to improve the efficiency of memory accesses. Experimental results from real-world graphs demonstrate that *APPR* significantly outperforms state-of-the-art methods with on average 2.4x speedup in execution time and 16.4x reduction in communication messages for social network graphs.

Acknowledgment

This work was supported in part by China NSFC Grant (No.61802261) and the grant of Guangdong Basic and Applied Basic Research Foundation (No.2020A1515011502). This research was also partially supported by China NSFC Grant (No.61872248), Guangdong NSF No.2017A030312008, Shenzhen Science and Technology Foundation (No.ZDSYS20190902092853047), Guangdong Science and Technology Foundation (No.2019B111103001 and No.2019B020209001), GDUPS (2015).

References

[1] Network Repository. http://networkrepository.com. [Online; accessed 22-July-2020].
[2] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized PageRank on MapReduce. In *ACM SIGMOD*, 2011.
[3] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010.
[4] S. Beamer, K. Asanović, and D. Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.
[5] S. Beamer, K. Asanović, and D. Patterson. Reducing PageRank communication via propagation blocking. In *IEEE IPDPS*, 2017.
[6] P. Berkhin. A survey on PageRank computing. *Internet Mathematics*, 2(1):73–120, 2005.
[7] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: on reducing communication and synchronization in graph computations. In *ACM HPDC*, 2017.
[8] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.
[9] A. Dash, A. Mukherjee, and S. Ghosh. A network-centric framework for auditing recommendation systems. In *IEEE INFOCOM*, 2019.
[10] D. F. Gleich. PageRank beyond the web. *SIAM Review*, 57(3):321–363, 2015.
[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *USENIX OSDI*, 2012.
[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: graph processing in a distributed dataflow framework. In *USENIX OSDI*, 2014.
[13] W. Guo, Y. Li, M. Sha, and K.-L. Tan. Parallel personalized PageRank on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):93–106, 2017.
[14] G. Karypis and V. Kumar. METIS – serial graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. [Online; accessed 22-July-2020].
[15] R. E. Korf. Multi-way number partitioning. In *AAAI*, 2009.
[16] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *USENIX OSDI*, 2012.
[17] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna. GPOP: a cache and memory-efficient framework for graph processing over partitions. In *ACM PPoPP*, 2019.
[18] K. Lakhotia, R. Kannan, and V. Prasanna. Accelerating PageRank using partition-centric processing. In *USENIX ATC*, 2018.
[19] Z. Liu, P. Zhou, Z. Li, and M. Li. Think like a graph: real-time traffic estimation at city-scale. *IEEE Transactions on Mobile Computing*, 18(10):2446–2459, 2019.
[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
[21] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *USENIX ATC*, 2017.
[22] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.
[23] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *USENIX HotOS*, 2015.
[24] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. FrogWild!: fast PageRank approximations on graph engines. *Proceedings of the VLDB Endowment*, 8(8):874–885, 2015.
[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999.
[26] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.
[27] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti. Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *IEEE/ACM Micro*, 2019.
[28] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti. PageRank acceleration for large graphs with scalable hardware and two-step SpMV. In *IEEE HPEC*, 2018.
[29] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM PPoPP*, 2013.
[30] L. Wang, L. Zhuang, J. Chen, H. Cui, F. Lv, Y. Liu, and X. Feng. Lazygraph: lazy data coherency for replicas in distributed graph-parallel computation. *ACM SIGPLAN Notices*, 53(1):276–289, 2018.
[31] H. Zhao, X. Xu, Y. Song, D. L. Lee, Z. Chen, and H. Gao. Ranking users in social networks with higher-order structures. In *AAAI*, 2018.