SIGMOD 2017 Self-archiving

Distributed Algorithms on Exact Personalized PageRank

Tao Guo[1] Xin Cao[2] Gao Cong[1] Jiaheng Lu[3] Xuemin Lin[2]

[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]School of Computer Science and Engineering, The University of New South Wales, Australia
[3]Department of Computer Science, University of Helsinki, Finland
tguo001@e.ntu.edu.sg, xin.cao@unsw.edu.au, gaocong@ntu.edu.sg, jiahenglu@gmail.com, lxue@cse.unsw.edu.au

ABSTRACT

As one of the most well known graph computation problems, *Personalized PageRank* is an effective approach for computing the similarity score between two nodes, and it has been widely used in various applications, such as link prediction and recommendation. Due to the high computational cost and space cost of computing the exact Personalized PageRank Vector (PPV), most existing studies compute PPV approximately. In this paper, we propose novel and efficient distributed algorithms that compute PPV exactly based on graph partitioning on a general coordinator-based share-nothing distributed computing platform. Our algorithms takes three aspects into account: the load balance, the communication cost, and the computation cost of each machine. The proposed algorithms only require one time of communication between each machine and the coordinator at query time. The communication cost is bounded, and the work load on each machine is balanced. Comprehensive experiments conducted on five real datasets demonstrate the efficiency and the scalability of our proposed methods.

1. INTRODUCTION

Measuring the similarities between nodes is a fundamental graph computation problem. Many random surfer model based measures have been proposed to capture the node-to-node proximities [2, 9, 24, 25, 30]. The *Personalized PageRank* (PPR) [25] is one of the most widely used measures and has gained extensive attention because of its effectiveness and theoretical properties. It has been utilized in various fields of applications, such as web search [8, 11], community detection [3, 21], link prediction [4], anomaly detection [43], and recommendation [22, 27].

Different from the *PageRank* model [38], PPR allows users to specify a set of preference nodes $\mathcal{P}$. The result of PPR w.r.t. $\mathcal{P}$ is called the Personalized PageRank Vector (PPV), which is denoted by $\mathbf{r}_{\mathcal{P}}$. Note that different preference vectors will yield different PPVs, and thus PPV needs to be computed in an online manner, which is different from PageRank. The PPR model can be simulated by numerous "random surfers." Initially, these surfers are distributed to the nodes in $\mathcal{P}$ equally. Next, in each step, a random surfer either jumps to a random outgoing neighbor with a probability of $1-\alpha$, or teleports back to a node u in $\mathcal{P}$ according to the specified preference of u with a probability of α (α is called the *teleport probability*). The procedure is performed repeatedly until it converges to a steady state, i.e, the number of surfers on each node does not change after each iteration. The final distribution of the random surfers on the nodes represents the PPV of $\mathcal{P}$.

The PPR model can also be interpreted as a linear system. We denote $\mathbf{u}_{\mathcal{P}}$ as the preference vector w.r.t. $\mathcal{P}$ and A as the normalized adjacency matrix of the graph. PPV $\mathbf{r}_{\mathcal{P}}$ can be computed as:

$$\mathbf{r}_{\mathcal{P}} = (1-\alpha)A^T\mathbf{r}_{\mathcal{P}} + \alpha\mathbf{u}_{\mathcal{P}}. \tag{1}$$

Hence, $\mathbf{r}_{\mathcal{P}}$ can be computed using the power iteration method, i.e., $\mathbf{r}_{\mathcal{P}}^{k+1} = (1-\alpha)A^T\mathbf{r}_{\mathcal{P}}^{k} + \alpha\mathbf{u}_{\mathcal{P}}$, where $\mathbf{r}_{\mathcal{P}}^{k}$ represents the vector computed in the k^{th} iteration, and $\mathbf{r}_{\mathcal{P}}^{0}$ is set to a uniform distribution vector. Finally, $\mathbf{r}_{\mathcal{P}}$ converges to the PPV of $\mathcal{P}$.

A. Challenges. Although computing PPVs has been extensively studied since the problem is first proposed in the work [25], it is still a very challenging task to compute exact PPVs efficiently for different online applications. One straightforward method is to adopt the power iteration approach by following Equation 1, which is however computationally expensive, and it is not practical for the online applications of PPR.

In the work [25], the linearity property of PPV is studied for computing PPVs exactly. If we pre-compute the PPV for each node, at query time, given a node set $\mathcal{P}$, according to the property, the PPV of $\mathcal{P}$ can be constructed using the pre-computed PPVs of the nodes in $\mathcal{P}$. Unfortunately, this method is impractical because of both the expensive pre-computation time and the offline storage requirements (where $O(|V|^2)$ space is needed). In order to reduce the complexity and space cost, the work [25] selects some important nodes as the "hub nodes", and the user preference nodes can only be from the hub set, thus losing the generality. Due to the hardness of computing the exact PPVs, most of existing studies (e.g., [5, 14, 44, 49]) focus on computing approximate PPVs. These methods sacrifice the accuracy to accelerate PPV computation.

Facing the challenges of computing the exact PPV for any arbitrary user preference node set, a natural question raised is how to perform the computation in parallel on multiple machines to gain efficiency. However, it is known that graph algorithms often exhibit poor locality and incur expensive network communication cost [34]. Therefore, it is a challenging task to design an efficient and scalable distributed algorithm for computing PPVs that has low communication cost and is able to achieve load balance.

The power iteration method can be implemented on the general distributed graph processing platforms to compute PPVs in a distributed way. However, this would not be suitable because of the unavoidable high communication cost. In the power iteration method, computing the vector in the current iteration requires the vector of the previous iteration. Thus, no matter how we distribute the computation, one machine always has to ask for some data from the other machines in each iteration before convergence, thus incurring expensive network communication cost. For example, the graph processing engine Pregel [36] is based on the general bulk synchronous parallel (BSP) model [45]. In each iteration of the BSP execution, Pregel applies a user-defined function on each vertex in parallel. The communications between vertexes are performed with message passing interfaces. The block-centric system Blogel [47] distributes subgraphs to machines as blocks, and messages between blocks are transmitted over the network. Both Pregel and Blogel need many rounds of communications to compute PPVs, which incurs expensive communication cost, thus suffering from low efficiency.

B. Our Proposal. In this work, we design novel distributed algorithms utilizing the graph partitioning for computing exact PPVs, which can be implemented on a general coordinator-based share-nothing distributed computing platform. We take into account three aspects in designing our algorithms: the load balance, the communication cost, and the computation cost on each machine. A salient feature of the proposed algorithms is that each machine only needs to communicate with the coordinator machine once at query time, and we prove that the communication cost of our method is bounded. The main idea of our algorithms are outlined as below.
Observation 1: We can separate the graph into disjoint subgraphs of similar size to distribute the PPV computation.

Based on this observation, we propose the *graph partition based algorithm*, denoted by GPA. We prove that the benefit from the balanced disjoint graph partitioning is two-fold: first, this guarantees a total space cost of $O((|V|-|H|)^2/m+2|V||H|+|H|^2))$, where V represents the nodes of the graph, m is the number of subraphs, and H represents the set of hub nodes separating the subgraphs. Note that $|H|$ is much smaller than $|V|$ (see more analysis in Appendix E), and thus this space cost is significantly smaller than applying the method [25] directly. Second, this enables us to compute PPVs in parallel on separate machines without incurring communication cost between machines. At query time, based on the pre-computed values, each machine constructs part of the PPV, and only communicates with the coordinator once (i.e., sending part of the PPV to the coordinator). If we use n machines, the communication cost of GPA is $O(n|V|)$.
Observation 2: We notice that if we treat each subgraph as an individual graph, we can use GPA to compute the "local" PPV of each node w.r.t. the subgraph itself, and these local PPVs can be used to construct the final "global" PPV. This motivates us to further partition each subgraph and we get a tree-like graph hierarchy. Then, we can recursively apply GPA along the hierarchy of subgraphs. We prove that the pre-computation space cost can be further reduced and bounded by utilizing the graph hierarchy.
Observation 3: Since the sizes of subgraphs in different levels of the hierarchy are different, simply distributing the subgraphs to machines cannot achieve load balance. We design a method to distribute the PPV computation evenly based on hub nodes partitioning to solve this problem.

Based on the aforementioned observations, we propose the *hierarchical graph partition based algorithm*, denoted by HGPA, which greatly reduces the space cost, and we prove that the communication cost of HGPA is $O(n|V|)$ as well.

C. Contributions. To the best of our knowledge, this is the first work that is able to compute exact PPVs efficiently in a distributed manner with reasonable space cost. In this paper, we propose novel distributed algorithms based on graph partitioning, and the salient features of new algorithms can be summarized as follows:

- **Efficiency**: Our proposed algorithm HGPA is $10 \sim 100$ times faster than the power iteration approach running on general distributed graph processing platforms Pregel+ [48] and Blogel [47], and can meet the efficiency need of online applications. The experimental study also shows that HGPA is faster than the power iteration and one state-of-the-art approximate PPV computation algorithm [49] even under the centralized setting.
- **Accuracy**: Our proposed algorithms, namely GPA and HGPA, are able to obtain the same result as that of the work [25], and thus have guaranteed exactness (Theorems 1 and 3).
- **Load Balance**: Our distributed algorithms GPA and HGPA are load balanced and HGPA scales very well with both the size of datasets and the number of machines. As shown in the experiments, the runtime can be reduced nearly by half if we double the number of machines.
- **Low Communication Cost**: Our algorithms only require one time of communication between each machine and the coordinator, and no communication between any two machines is needed for pre-computation and query processing. As shown in the experiments, our method only costs about 1.5 MB network communication to compute a PPV on a graph containing 3M nodes using 10 machines.

The rest of the paper is organized as follows. Section 2 provides the preliminary of this work. Sections 3 and 4 present the distributed algorithms GPA and HGPA, respectively. Section 5 explains our distributed pre-computation. We perform comprehensive experiments to evaluate the efficiency and scalability of our methods on 5 real datasets, and the results are reported in Section 6. Section 7 summarizes the related work on computing personalized PageRank. Finally, Section 8 offers conclusions and future research directions.

2. PRELIMINARY

Table 1: Definition of main symbols.

Symbol	Definition
$\mathcal{P}$	User preference node set
$\mathbf{r}_u$	PPV of a single preference node u
α	Teleport(Restart) probability, $0 < \alpha < 1$
H	Hub node set
$t : u \rightsquigarrow v$	A random tour from node u to v
$\mathbb{P}(t)$	Weight of tour t
$\mathcal{L}(t)$	Length of tour t
$\mathbf{r}_u^H$	PPV of tours from u passing at least one node in H
$\mathbf{p}_u^H(\mathbf{P}_u^H)$	(Adjusted) Partial vector $w.r.t.$ node u and hub nodes H
$\mathbf{s}_u^H(\mathbf{S}_u^H)$	(Adjusted) Hubs skeleton vector $w.r.t.$ node u and hub nodes H
$\mathbf{x}_u$	Basic vector with zero filled except $x_u(u) = 1$
G_m^i	A hierarchical partitioned subgraph in the m^{th} level
$H(G_m^i)$	Hub nodes of subgraph G_m^i

2.1 PPV Decomposition

As proved by Jeh and Widom [25], the PPV score $\mathbf{r}_u(v)$ is equal to the corresponding *Inverse P-distance* from a query node u to v, which is measured by all the weighted tours from u to v:

$$\mathbf{r}_u(v) = \sum_{t:u \rightsquigarrow v} \mathbb{P}(t), \qquad (2)$$

where each tour $t : u \rightsquigarrow v$ represents a random surfer that consists of a sequence of edges starting from u and ending at v. Note that it is allowed to teleport back to u, and thus there may exist cycles in the tour. The weight of a tour $\mathbb{P}(t)$ is the probability that this tour walks from u to v along with t:

$$\mathbb{P}(t) = \alpha(1-\alpha)^{\mathcal{L}(t)} \prod_{i=1}^{\mathcal{L}(t)} \frac{1}{|Out(w_i)|},$$

where nodes $w_1(i.e., u), w_2, \cdots, w_{\mathcal{L}(t)}(i.e., v)$ comprise the path of length $\mathcal{L}(t)$, and $|Out(w_i)|$ is the outdegree of node w_i. Although the concept of inverse P-distance intuitively explains the distribution of random walk, it is impossible to sum up all the tours to obtain the PPV values. The reason is that there may exist loops in the tours, and thus the number of tours could be infinite.

In order to compute $\mathbf{r}_u$, it is divided into two types of vectors w.r.t. a set of hub nodes H, i.e., partial vectors and hubs skeleton vectors.

Definition 1: Partial vector: Given a node u, its *partial vector* $\mathbf{p}_u^H$ is defined as a vector of random walk results computed using tours passing through no hub nodes. That means, given a node v,

$$\mathbf{p}_u^H(v) = \sum_{\forall w \in t \& w \neq u, w \notin H} \mathbb{P}(t)$$

Definition 2: Hubs skeleton vector: Given a node u, its *hubs skeleton vector* $\mathbf{s}_u^H$ is defined as a vector of $|H|$ dimensions composed of all hub nodes' PPV values w.r.t. u. Given a hub node h, $\mathbf{s}_u^H(h) = \mathbf{r}_u(h)$.

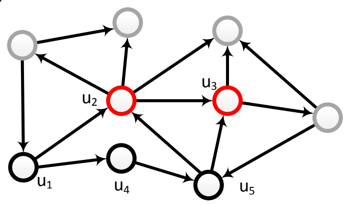


Figure 1: Random Surfer Example

If a tour passes no hub node, it contributes to the partial vector. If a tour stops at a hub node, it contributes to the skeleton vector. For ease of understanding, we explain the meaning of the two types of vectors with the graph shown in Figure 1, where two nodes u_2 and u_3 are selected as hub nodes. We next illustrate the partial and skeleton vectors with node u_1 as an example.

The partial vector $\mathbf{p}_{u_1}^H$ consists of two tours t_1 and t_2, where $t_1 = u_1 \rightsquigarrow u_4$ and $t_2 = u_1 \rightsquigarrow u_4 \rightsquigarrow u_5$. Note that all other tours are blocked by either u_2 or u_3 (hub nodes), and thus any gray node in this example is not reachable from u_1. We have $\mathbf{p}_{u_1}^H(u_4) = \mathbb{P}(t_1)$ and $\mathbf{p}_{u_1}^H(u_5) = \mathbb{P}(t_2)$.

For the skeleton vector $\mathbf{s}_{u_1}^H$, we only consider the tours stopping at a hub node. Different from the partial vector, the tour can contain one or more hub nodes. From u_1 to hub node u_2, there exist two tours $t_3 = u_1 \rightsquigarrow u_2$ and $t_4 = u_1 \rightsquigarrow u_4 \rightsquigarrow u_5 \rightsquigarrow u_2$. From u_1 to u_3 there exist three tours $t_5 = u_1 \rightsquigarrow u_2 \rightsquigarrow u_3$, $t_6 = u_1 \rightsquigarrow u_4 \rightsquigarrow u_5 \rightsquigarrow u_3$, and $t_7 = u_1 \rightsquigarrow u_4 \rightsquigarrow u_5 \rightsquigarrow u_2 \rightsquigarrow u_3$. Therefore, $\mathbf{s}_{u_1}^H(u_2) = \mathbb{P}(t_3) + \mathbb{P}(t_4)$ and $\mathbf{s}_{u_1}^H(u_3) = \mathbb{P}(t_5) + \mathbb{P}(t_6) + \mathbb{P}(t_7)$.

Note that because of cycles in the tours, it is not feasible to compute all possible tours, which may be infinite. This simple example is only used to illustrate the intuitive meaning of the two types of vectors.

2.2 PPV Construction

The PPV of u (i.e., $\mathbf{r}_u$) can be constructed by the partial vectors and hubs skeleton vectors. We use $\mathbf{r}_u^H$ to denote the vector of random walk results computed using tours passing though at least one hub node. That is, give a node v, $\mathbf{r}_u^H(v) = \sum_{t: u \rightsquigarrow H \rightsquigarrow v} \mathbb{P}(t)$. We can combine $\mathbf{r}_u^H$ and the partial vector of u to obtain the full PPV of u: $\mathbf{r}_u^H + \mathbf{p}_u^H = \mathbf{r}_u$. As proved by Jeh and Widom [25], $\mathbf{r}_u^H$ can be computed according to the following equation:

$$\mathbf{r}_u^H = \frac{1}{\alpha} \sum_{h \in H} (\mathbf{s}_u^H(h) - \alpha f_u(h)) \cdot (\mathbf{p}_h^H - \alpha \mathbf{x}_h), \quad (3)$$

where $f_u(h) = 1$ if $u = h$, and 0 otherwise. $f_u(h)$ is used to deal with the special case when u is a hub node. $\mathbf{x}_h$ is a vector that has value 1 at h and 0 everywhere else. It is used to deal with the special case when h is the target node.

In order to make the equations more clear, we define the *adjusted partial vector* for each hub node h:

$$\mathbf{P}_h^H = \mathbf{p}_h^H - \alpha \mathbf{x}_u,$$

and the *adjusted hubs skeleton vector* $\mathbf{S}_u^H$ for each node u, where:

$$\mathbf{S}_u^H(h) = \mathbf{s}_u^H(h) - \alpha f_u(h).$$

After the partial vectors and skeleton vectors have been pre-computed, the exact PPV of a given query node u can be constructed by:

$$\mathbf{r}_u = \frac{1}{\alpha} \sum_{h \in H} \mathbf{S}_u^H(h) \cdot \mathbf{P}_h^H + \mathbf{p}_u^H \quad (4)$$

In order to reduce the complexity, the work [25] only considers the construction of PPV for hub nodes. According to Equation 4, we need to pre-compute the partial vectors of all hub nodes, which requires space cost $O((|V| - |H|) \cdot |H|)$ in the worst case, and the skeleton vectors of all hub nodes, which requires $O(|H|^2)$ space.

2.3 A Brute-force Extension

It is too restricted to consider the hub set from only preference nodes as done in work [25]. In fact, it can be extended to compute the PPV for any given query node, as indicated by Equation 4. However, it incurs huge space cost as explained below: first, pre-computing the partial vectors for non-hub nodes requires worst case space cost $O((|V| - |H|)^2)$, which happens when every node can reach every other node without passing any hub node; second, pre-computing the partial vectors for hub nodes requires $O((|V| - |H|)|H|)$ space in the worst case; third, pre-computing the skeleton vectors for all nodes requires space cost $O(|V| \cdot |H|)$. Thus, the total pre-computation space cost is $O(|V|^2)$, which is equivalent to pre-computing the PPVs of all nodes. In practice, the vectors may be sparse and the space cost is usually not that large, but it is still not applicable for large graphs. We denote this algorithm by PPV-JW.

3. ALGORITHM GPA

Although it is usually difficult to efficiently parallelize graph algorithms [34], we propose the *graph partition based algorithm*, denoted by GPA, to distribute the PPV computation. We present the algorithm details in Section 3.1. We adopt a general coordinator-based share-nothing distributed computing platform. For the convenience of presentation, we call each machine handling some subgraphs as "machine" and the machine aggregating the final result as "coordinator." In addition, in Section 3.2 we prove that GPA reduces the space cost significantly compared with the method PPV-JW presented in Section 2.3.

3.1 Distributed PPV Construction

Graph partition is commonly used in parallel computing [23], but it is always challenging to decouple the computation dependencies. We aim to distribute the PPV computation to multiple machines and each machine can independently compute part of the result.We use a balanced graph partition algorithm (METIS [26]) to divide the graph into m disjoint subgraphs. Figure 2 shows an example, where the graph is partitioned into two disjoint subgraphs G_1 and G_2. The bridging nodes between subgraphs form the hub nodes. In the example, u_1 and u_2 are selected as the hub nodes.

After the graph is partitioned into m subgraphs, we distribute the subgraphs to multiple machines evenly. The pre-computed partial vector and skeleton vector of each node is stored in the machine where the node resides.

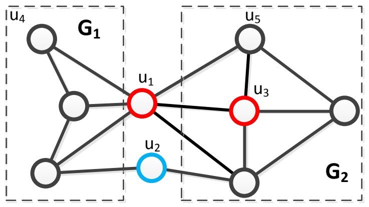


Figure 2: Example of graph partition and hub nodes

Recall that the PPV construction of the method [25] is based on Equation 4. Now the pre-computed vectors are stored on n machines M_1, ..., M_n, and we can distribute the computation according to the following equation:

$$\mathbf{r}_u = \frac{1}{\alpha}\sum_{i=1}^{n}\sum_{h\in H(M_i)} \mathbf{S}_u^H(h)\cdot \mathbf{P}_h^H + \mathbf{p}_u^H, \tag{5}$$

where $H(M_i)$ denotes the set of hub nodes assigned to M_i.

Assume that the partial and skeleton vectors have already been pre-computed and stored. We introduce the details of this step in Section 5. Equation 5 indicates that we can do the distributed PPV construction at query time as follows: after a query node u is given, the coordinator first detects the machine M_u that stores the partial vector of u. Then, M_u computes the following vector: $\mathbf{v}_u = \frac{1}{\alpha}\sum_{h\in H(M_u)} \mathbf{S}_u^H(h)\cdot \mathbf{P}_h^H + \mathbf{p}_u^H$. Simultaneously, each of the other machines M_j ($1 \le j \le n, j \ne u$) computes a vector $\mathbf{v}_j = \frac{1}{\alpha}\sum_{h\in H(M_j)} \mathbf{S}_u^H(h)\cdot \mathbf{P}_h^H$. The coordinator receives the vectors computed from all machines, and computes the final PPV as $\mathbf{r}_u = \sum_{i=1}^{n}\mathbf{v}_i$. Therefore, at query time, each machine communicates with the coordinator exactly once.

Theorem 1: GPA can obtain the same results as computed by the work proposed by Jeh and Widom [25].

PROOF. $\sum_{i=1}^{n}\sum_{h\in H(M_i)}$ is equal to $\sum_{h\in H}$, and thus Equation 5 can compute the same vector as Equation 4. □

Communication Cost. Each machine needs to compute a vector of size $|V|$, and then sends it to the coordinator. Thus, if n machines are employed in GPA, the total communication cost of GPA is $O(n|V|)$.

Time Complexity. According to Equation 5, to compute a node u's PPV, we need to fetch the partial vector of each hub node h reachable from node u (i.e., $\mathbf{P}_h^H$), the skeleton vector of u (i.e., $\mathbf{S}_u^H$), and the partial vector of u (i.e., $\mathbf{P}_u^H$). The partial vector of each hub node h gets its weight $\mathbf{S}_u^H(h)$ from the skeleton vector of u, and they are then aggregated together with $\mathbf{P}_u^H$. Therefore, we need to read at most $O(|H|)$ vectors of dimension $|V|$ and $O(|H|)$ vectors.

Assume that we use n machines and the hub nodes are distributed evenly to these machines. In the worst case, on each machine we only need to sum up $O(|H|/n)$ vectors, and on the coordinator we need to sum up n vectors. Thus, the complexity of GPA is $O(\frac{1}{n}|H||V| + n|V|)$.

3.2 Space Cost of GPA

We proceed to show that the space cost of GPA is reduced significantly compared with PPV-JW, the extension of the method [25] as presented in Section 2.3. We use an example as shown in Figure 2 to briefly explain the reason. In the work [25], nodes with high PageRank values are chosen as hub nodes, since most random walks have high probability to visit these nodes. As a result, nodes u_1 and u_3 are selected to be the hub nodes. The support of vector $\mathbf{p}_{u_4}^H$ (the partial vector of u_4) can be as large as the size of the graph, since there exists a tour from u_4 to each other node in the graph. If we select the nodes u_1 and u_2 as the hub nodes, they are able to partition the graph into two disjoint subgraphs. In this way, every tour from node u_4 to a node in the other part has to pass either u_1 or u_2, and thus the tours between different subgraphs are blocked by the hub nodes. The support of $\mathbf{p}_{u_4}^H$ is reduced to the size of the subgraph containing u_4.

As illustrated in the example, after the graph partition, we select the bridging nodes between subgraphs as the hub nodes. The random walks that represent partial vectors are restricted within each individual subgraph by the hub nodes. Thus, the support of the partial vector of non-hub node $\mathbf{p}_u^H$ is reduced from $|V| - |H|$ to the size (the number of nodes) of the subgraph containing u. In PPV-JW, the space cost of partial vectors of non-hub nodes, i.e., $O((|V| - |H|)^2)$, is the major cost. In GPA, if we assume that the graph is partitioned into m subgraphs of equal size, the size of each subgraph is $O((|V| - |H|)/m)$, and the space cost is $O((|V| - |H|)^2/m)$. In the worst case, the space cost of storing the partial vectors of the hub nodes is $O((|V|-|H|)|H|)$, and storing the skeleton vectors of non-hub nodes costs $O(|V||H|)$. In conclusion, the total space cost of GPA is $O((|V| - |H|)^2/m + 2|V||H| - |H|^2)$, based on the balanced graph partition.

Note that for most graphs, the number of hub nodes that can divide the graphs into different components is always much smaller than the total number of nodes, i.e., $|H| \ll |V|$. Therefore, the space cost of GPA is much smaller than $O(|V|^2)$, the cost of PPV-JW to compute PPV for an arbitrary node in a centralized setting.

4. ALGORITHM HGPA

In GPA, the computation of partial vectors is restricted to each individual subgraph, and the total space cost of all machines is consequently reduced compared with that required by the centralized extension of the method [25]. To further reduce the space cost and achieve better load balance and efficiency, we propose a new approach based on a hierarchy of subgraphs. This algorithm is inspired by the observation that, the computation of a node u's partial vector is equivalent to the computation of the "local" PPV of u w.r.t. the subgraph that contains u.

We first introduce how we can compute the partial vectors using the way of computing "local" PPVs in Section 4.1. Based on this property, we partition the graph into a hierarchy of subgraphs as presented in Section 4.2. Then, we introduce how to get PPV utilizing the graph hierarchy in Section 4.3, and we design the distributed PPV computation method to achieve load balance in Section 4.4. We prove that the space cost benefits from the hierarchical graph partitioning in Section 4.5.

4.1 Partial Vector vs Local PPV

In GPA, we partition the graph into disjoint subgraphs. Recall that as presented in Section 2 PPV can be computed by the random surfers following all possible paths, and the partial vector of a node is the result computed using random surfers passing no hub node. Therefore, the computation of the partial vector of a node is only related to the subgraph containing the node. This motivates us to

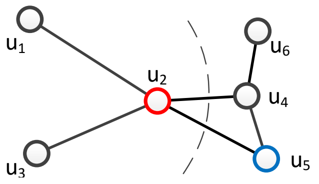


Figure 3: Full Graph G

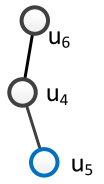


Figure 4: Subgraph SG

think whether it is possible to use the local PPV of a node w.r.t. a subgraph to compute the partial vector of this node. Consider the toy graph in Figure 3, which is separated by hub node u_2. Figure 4 shows the isolated subgraph SG. Suppose the query node is u_5, we would like to know whether the partial vector $\mathbf{p}_{u_5}^{H}$ in G is equal to the local PPV of u_5 in SG, i.e., $\mathbf{r}_{u_5}[SG]$.

Recall that the computation of $\mathbb{P}(t)$ requires the out-degrees of nodes in the tour t. The out-degree of u_5 is 2 in G but it is 1 in subgraph SG, and obviously the probability of a random surfer walking from u_5 to u_4 is different in the two graphs. Therefore, $\mathbf{p}_{u_5}^{H} \neq \mathbf{r}_{u_5}[SG]$. In order to solve this problem, we introduce the following definition.

Definition 3: Virtual subgraph: After partitioning a graph into smaller subgraphs, for each subgraph SG, we create a virtual node VN, and for each edge that connects a hub node and a node u in SG, we create an edge between u and VN. We call the graph composed of subgraph SG and its virtual node as well as the edges connecting them the **virtual subgraph**, which is denoted by $\widetilde{SG}$.

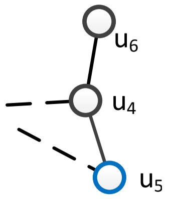


Figure 5: Virtual Subgraph $\widetilde{SG}$

Figure 5 shows the corresponding virtual subgraph of SG as shown in Figure 4. Using the concept of the virtual subgraph, we have the following theorem.

Theorem 2: Given a graph G, a set of hub nodes H, and a node u in a subgraph SG, the partial vector of u, i.e., $\mathbf{p}_{u}^{H}$ is equivalent to u's PPV vector w.r.t. the virtual subgraph of SG, i.e., $\mathbf{r}_{u}[\widetilde{SG}]$.

PROOF. Given any node v in G, the value of $\mathbf{p}_{u}^{H}(v)$ can be computed using the tours from u to v without passing through any hub node in H. If $v \notin SG$, $\mathbf{p}_{u}^{H}(v) = 0$. Otherwise, $\mathbf{p}_{u}^{H}(v) = \sum_{t:u \rightsquigarrow v} \mathbb{P}(t)$ where $t : u \rightsquigarrow v$ represents a tour from u to v within SG.

According to Equation 2, $\mathbf{r}_{u}[\widetilde{SG}](v)$ can also be computed by $\sum_{t:u \rightsquigarrow v} \mathbb{P}(t)$. Note that all tours are within $\widetilde{SG}$, the virtual node in $\widetilde{SG}$ has no outgoing edge. Hence, for each tour t, the value of $\mathbb{P}(t)$ is the same for computing both $\mathbf{p}_{u}^{H}(v)$ and $\mathbf{r}_{u}[\widetilde{SG}](v)$, and this means that $\mathbf{p}_{u}^{H}(v)$ is equal to $\mathbf{r}_{u}[\widetilde{SG}](v)$. □

Using virtual subgraph $\widetilde{SG}$, we guarantee that the partial vector equals to the local PPV in the virtual subgraph, i.e., $\mathbf{p}_{u_5}^{H} = \mathbf{r}_{u_5}[\widetilde{SG}]$. For the simplification of presentation, we use "subgraph" to indicate "virtual subgraph" in the rest of the chapter.

4.2 Hierarchical Graph Partitioning

Based on Theorem 2, we can obtain the partial vector for a node by computing the local PPV in the subgraph containing the node. To compute the local PPV for a subgraph, we can recursively apply GPA within the subgraph, i.e., we further partition the subgraph into lower level subgraphs, and for each lower level subgraph we apply Theorem 2, and the procedure can be repeated until we hit a specified level. To realize the idea, we recursively partition the whole graph from top to down into a hierarchy. For ease of presentation, we partition the graph into a hierarchy of two-way partitions. As shown in Figure 6, the root of the hierarchy is G itself. Generally, in the m^{th} $(0 \le m \le l)$ level, G is partitioned into 2^m disjoint subgraphs, and we denote a subgraph in the m^{th} level by G_m^i, where $0 \le i < 2^m$. In this hierarchy, given a subgraph G_m^i, its parent subgraph is $G_{m-1}^{\lfloor \frac{i}{2} \rfloor}$, and its two child subgraphs are G_{m+1}^{2i} and G_{m+1}^{2i+1}. We denote the hub nodes separating G_{m+1}^{2i} and G_{m+1}^{2i+1} by $H(G_m^i)$.

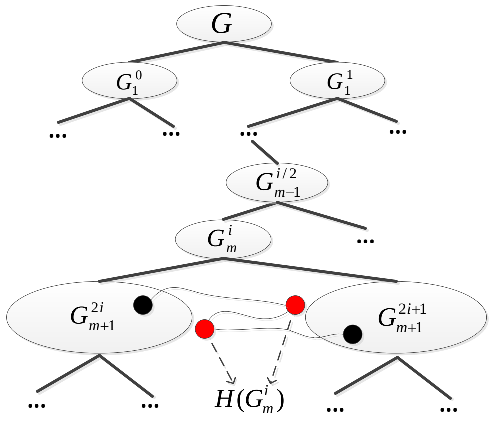


Figure 6: Hierarchy of the original graph

Figure 7 exemplifies the hierarchy and the hub nodes of each level obtained using the multilevel 2-way partitioning method. First, G is partitioned into G_1^0 and G_1^1. At this level, edges (u_2, u_4) and (u_2, u_5) are hub edges, and we can get the hub node set $H(G) = \{u_2\}$. In the next level, G_1^0 is partitioned into G_2^0 and G_2^1 with no hub node; G_1^1 is partitioned into G_2^2 and G_2^3 with hub node set $H(G_1^1) = \{u_4\}$. Note that once a node is selected as hub node, this node and all the related edges will be omitted in the next level and not appear in any of the subgraphs. For example, u_2 is contained in $H(G)$ but in neither G_2^0 nor G_2^1.

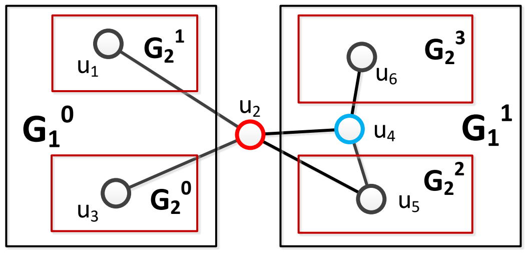


Figure 7: Example of the hierarchy

We use the two-way partitioning to reduce the number of hub nodes. We employ the 2-way partitioning algorithm [26] to recursively perform the partitioning from top to down, until we reach a level such that no edges exist within the same subgraph. Using the two-way partitioning, minimizing the number of hub nodes is identical to the minimum vertex cover problem in a bipartite graph. This problem is proved to be solvable in polynomial time by Kőnig's theorem [33]. We use the algorithm [33] to select the minimum hub nodes from the returned hub edges. Our proposed techniques are still applicable if we adopt multiple-way partitioning. We compare the effects of different partitioning strategies in Section 6.

4.3 PPV Construction on Hierarchy

We proceed to explain the procedure of PPV computation based on the hierarchy. Consider the hub nodes set $\mathcal{H}_0$ in the first level of the hierarchy, which separates G into two subgraphs G_1^0 and G_1^1. Given a query node $u \in G_1^1$, according to Equation 4, to obtain $\mathbf{r}_u$ in GPA, we need the partial vector of node u, i.e., $\mathbf{p}_u^{\mathcal{H}_0}$, the partial vectors of hub nodes, i.e., $\mathbf{p}_h^{\mathcal{H}_0}(h \in \mathcal{H}_0)$, and the skeleton vectors $\mathbf{s}_u^{\mathcal{H}_0}$.

$\mathbf{p}_h^{\mathcal{H}_0}$ and $\mathbf{s}_u^{\mathcal{H}_0}$ can be computed similarly as in GPA. According to Theorem 2, the partial vector of u ($\mathbf{p}_u^{\mathcal{H}_0}$) is identical to u's PPV vector w.r.t. the virtual subgraph of G_1^1. Thus, we can compute the local PPV of u w.r.t. $\widetilde{G_1^1}$ as $\mathbf{p}_u^{\mathcal{H}_0}$. That is, we construct $\mathbf{p}_u^{\mathcal{H}_0}$ by using the local skeleton vector of lower leve hub nodes in $\widetilde{G_1^1}$ and the local partial vector of u in $\widetilde{G_1^1}$, which can be further computed based on lower level subgraphs, and thus the procedure can be repeated until we reach the leaf-level subgraphs.

Assume that all partial and skeleton vectors have been processed and stored (see more details in Section 5). Formally, the construction at query time can be described by the following equation:

$$\mathbf{r}_u = \sum_{m=0}^{l-1} \frac{1}{\alpha} \sum_{h \in H(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}] + \mathbf{r}_u[G_l^{(u)}], \quad (6)$$

where $G_m^{(u)}$ denote the subgraph in the m^{th} level that contains u, $\mathbf{S}_u^H[G_m^{(u)}]$ is the adjusted skeleton vector of u w.r.t. the subgraph $G_m^{(u)}$, and $\mathbf{P}_h^H[G_m^{(u)}]$ is the adjusted partial vector of a hub node w.r.t $G_m^{(u)}$. Note that the subgraphs we visit are $G_l^{(u)}$,$G_{l-1}^{(u)}$,$G_{l-2}^{(u)}$, $\cdots$, G_0^0, and thus the number of graphs visited during this procedure is exactly l.

Theorem 3: The vector computed by Equation 6 is exactly the same as that computed by GPA.

PROOF. u's local partial vector at level m can be computed as:

$$\mathbf{r}_u[G_m^{(u)}] = \frac{1}{\alpha} \sum_{h \in H(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}] + \mathbf{r}_u[G_{m-1}^{(u)}].$$

Therefore, Equation 6 can be written as:

$$\begin{aligned}
\mathbf{r}_u = & \sum_{m=0}^{l-2} \frac{1}{\alpha} \sum_{h \in H(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}] \\
& + \frac{1}{\alpha} \sum_{h \in H(G_{l-1}^{(u)})} \mathbf{S}_u^H[G_{l-1}^{(u)}](h) \cdot \mathbf{P}_h^H[G_{l-1}^{(u)}] + \mathbf{r}_u[G_l^{(u)}] \\
= & \sum_{m=0}^{l-2} \frac{1}{\alpha} \sum_{h \in H(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}] + \mathbf{r}_u[G_{l-1}^{(u)}] \\
= & \cdots \\
= & \frac{1}{\alpha} \sum_{h \in H(G_0^{(u)})} \mathbf{S}_u^H[G_0^{(u)}](h) \cdot \mathbf{P}_h^H[G_0^{(u)}] + \mathbf{r}_u[G_1^{(u)}]
\end{aligned}$$

Note that $G_0^{(u)}$ is the whole graph, and $G_1^{(u)}$ can be viewed as a subgraph in GPA, and thus the result of Equation 6 is exactly the same as Equation 5. □

4.4 Distributed PPV Computation

Based on the graph hierarchy, we can compute the PPV of a graph from the pre-computation results w.r.t. its hub nodes and the PPV of its child graphs. However it is still challenging to design a distributed algorithm, especially when the graph is large and the pre-computation vectors are too large to save in memory.

To address these challenges, we propose HGPA, the hub-distributed hierarchical graph partition based algorithm, which is load balanced and scalable because the computation in each level can be evenly distributed. We use one coordinator machine to collect the results from other machines for HGPA. We note that the computation of $\mathbf{r}_u$ relies on all hub nodes in all levels as shown in Equation 6. This inspires us to divide the hub node set of each subgraph in each level into disjoint components to distribute the computation. Specifically, given a subgraph SG, we divide $H(SG)$ into s disjoint subsets equally $H^1(SG), H^2(SG), \cdots, H^s(SG)$, where $H(SG) = \cup_{i=1}^s H^i(SG)$. We do this on each subgraph in each level. As a result, the computation of $\mathbf{r}_u$ can be interpreted as Equation 7 based on the balanced hub nodes partition.

$$\mathbf{r}_u = \sum_{m=0}^{l-1} \frac{1}{\alpha} \sum_{i=1}^{s} \sum_{h \in H^i(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}] + \mathbf{r}_u[G_l^{(u)}] \quad (7)$$

It is obvious that the first component of Equation 7 can compute the same results as the first component in Equation 6.

Based on Equation 7, for each subgraph in each level, we divide its hub node set evenly into s disjoint subsets and store them in s machines. We also distribute the leaf level subgraphs evenly to s machines. Each machine only maintains the partial and skeleton vectors of nodes stored on it. Given a query node u, the i^{th} machine computes a vector using the pre-computation results stored on it, i.e., $\sum_{h \in H^i(G_m^{(u)})} \mathbf{S}_u^H[G_m^{(u)}](h) \cdot \mathbf{P}_h^H[G_m^{(u)}]$, and then sends the vector to the coordinator. The partial vectors of all non-hub nodes w.r.t. their leaf level subgraphs (e.g., $\mathbf{r}_u[G_l^{(u)}]$) are also distributed to s machines evenly. The coordinator sums up all the vectors received from the machines to construct the PPV. Figure 8 illustrates the idea of HGPA.

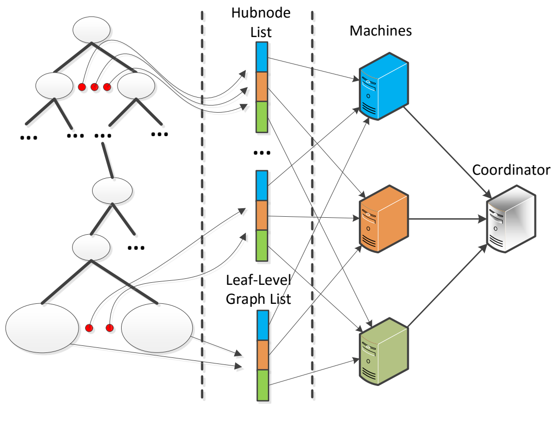


Figure 8: HGPA Architecture

It is obvious that algorithm HGPA is load balanced. The computation on each machine is presented by Algorithm 1.

Theorem 4: If the hub node set of each subgraph is divided into n disjoint subsets, the communication cost can be bounded by $O(n|V|)$.

PROOF. In HGPA, each machine computes a vector of size at

Algorithm 1: HGPA processor(q, i, α)

```
1 $\vec{ppv} \leftarrow \vec{0}$;
2 foreach subgraph $Gs$ do
3     foreach hubnode $h$ in $H^i(Gs)$ do
4         foreach non-zero entry $\mathbf{p}_h^H[Gs](k)$ do
5             $\vec{ppv}(k) \leftarrow \vec{ppv}(k) + \mathbf{S}_q^H[Gs](h) \cdot \mathbf{P}_h^H[Gs](k)/\alpha$;
6 if $q$ is not a hub node and $\mathbf{P}_q^H[G_l^q]$ on machine $i$ then
7     foreach non-zero entry $\mathbf{P}_q^H(k)$ do
8         $\vec{ppv}(k) \leftarrow \vec{ppv}(k) + \mathbf{P}_q^H[G_l^q](k)$;
9 send $\vec{ppv}$ to coordinator;
```

most $|V|$, and sends it to the coordinator. Hence, the total communication cost is $O(n|V|)$. □

4.5 Space Cost of HGPA

We assume that the graph is partitioned in balance in each level. Let l denote the level of the graph hierarchy, and there are 2^l leaf level subgraphs. We have the following theorem that shows the space complexity of HGPA.

Theorem 5: The space cost of HGPA is $O((|V| - |H|)^2/2^l + \sum_{i=0}^{l} |H_i|(|V| - |H|)/2^i + (|V| - |H|) \sum_{i=0}^{l} |H_i^{max}|)$, where H_i is the set of hub nodes in the i^{th} level, and H_i^{max} is the maximum number of hub nodes in a subgraph in the i^{th} level.

PROOF. In the leaf level, we compute and store the PPV of non-hub nodes w.r.t. each subgraph, and the size of the local PPV is $(|V| - |H|)/2^l$. Therefore, the total space cost of storing PPVs for leaf level subgraphs is $O((|V| - |H|)^2/2^l))$.

In the i^{th} level, there are 2^i subgraphs. For each subgraph G_i^j, we need to compute the partial vectors of the hub nodes in it. Hence, the space cost of storing the partial vectors of hub nodes for the level is $O(|H_i|(|V| - |H|)/2^i)$. For all levels, the total space cost of this part is $O(\sum_{i=0}^{l} |H_i|(|V| - |H|)/2^i)$.

We also need to compute the skeleton vectors of the non-hub nodes in G_i^j, and the space cost for G_i^j is $O((|V| - |H|)/2^i |H_i(G_i^j)|)$, where $H_i(G_i^j)$ is the set of hub nodes in G_i^j. In the i^{th} level, in total the space cost is $O((|V| - |H|)/2^i \sum_{j=0}^{2^i} |H_i(G_i^j)|) < O((|V| - |H|)/2^i \sum_{j=0}^{2^i} |H_i^{max}|) = O((|V| - |H|)|H_i^{max}|)$. Thus, for all levels, the total cost of this part is $O((|V| - |H|) \sum_{i=0}^{l} |H_i^{max}|)$. □

To compare the space cost of GPA with HGPA in an intuitive way, we consider that in GPA the graph is partitioned as the leaf level of the graph hierarchy in HGPA, i.e., the leaf level subgraphs in HGPA are the subgraphs in GPA. We can conclude that HGPA has smaller space cost than GPA. The two algorithms have the same space cost of storing the partial vectors of non-hub nodes. However, HGPA has smaller space cost of storing the partial vectors of hub nodes and the skeleton vectors of non-hub nodes than that in GPA, because $O(|V||H|) > O(\sum_{i=0}^{l} |H_i|(|V| - |H|)/2^i)$ and $O((|V| - |H|)|H|) > O((|V| - |H|) \sum_{i=0}^{l} |H_i^{max}|)$.

5. DISTRIBUTED PRE-COMPUTATION

We proceed to present how to pre-compute the partial and the skeleton vectors in a distributed manner for our algorithms.

5.1 Distributed Partial Vectors Computation

We adopt the *selective expansion algorithm* [25] as introduced in Section E.1 to compute the partial vectors for all nodes. As shown in Equation 9, the iteration requires the information of the graph structure. We keep a copy of the graph structure on each machine. As a result, no communication is required in the pre-computation of partial vectors. The computation can be done for each node separately, and each machine only needs to handle the nodes assigned to it.

In GPA, each machine only computes the partial vector $\mathbf{p}_u^H$ if node u is assigned to it. When computing the partial vectors for a non-hub node, only the structure of the subgraph containing the node is required, because the computation can be restricted by the subgraph.

In HGPA, the partial vectors of nodes are computed by doing iterations w.r.t. the subgraphs, rather than w.r.t. the whole graph as in GPA, thus reducing the space and time complexity required during the iteration for a node. Given a node u assigned to machine M, if u is a non-hub node, M computes its partial vector w.r.t. the leaf-level subgraph containing u. Otherwise, if u is a hub node at level m, M computes its partial vector w.r.t. the subgraph at level m containing u.

5.2 Distributed Skeleton Vectors Computation

We improve the *dynamic programming algorithm* [25] as introduced in Section E.2 in order to distribute the skeleton vectors computation. As shown in Equation 10, for a given node u, two vectors $\mathbf{D}_k[u]$ and $\mathbf{E}_k[u]$ are used to do the iteration, and finally $\mathbf{D}_k[u]$ would converges to $\mathbf{s}_u^H$, the skeleton vector of u. However, this algorithms incurs huge space cost, and the intermediate vectors could likely be larger than available main memory, and it is suggested to be implemented as a disk-based version [25]. The problem of the original method is that, in each step, the update of $\mathbf{D}_{k+1}[u]$ relies on $\mathbf{D}_k[v]$, if there exists an edge (u, v). Thus, the skeleton vectors of u cannot be computed without computing the skeleton vectors of other nodes. In addition, the computation of a skeleton vector of a node needs to consider all hub nodes, and we have to maintain the value of $\mathbf{s}_u^H(h)$ for each node u in memory. As a result, the original algorithm can not work in parallel and consumes huge memory.

To distribute the computation and reduce memory cost, we consider computing the skeleton vector score w.r.t. a single hub node each time, i.e., $\mathbf{D}_k[u](h)$ for each node u. Our algorithm can be explained by Equation 8. Initially, $\mathbf{F}_0 = \mathbf{0}$. It is easy to show that after convergence, $\mathbf{F}_k(u)$ is equal to $\mathbf{D}_k[u](h)$, which is $\mathbf{s}_u^H(h)$.

$$\mathbf{F}_{k+1}(u) = (1 - \alpha) \sum_{v \in Out(u)} \frac{\mathbf{F}_k(v)}{|Out(u)|} + \alpha \mathbf{x}_h(u) \qquad (8)$$

Theorem 6: $\mathbf{F}_k(u)$ is equal to $\mathbf{s}_u^H(h)$ when the iteration converges.

PROOF. We prove the correctness of Equation 8 by demonstrating that $\mathbf{F}_k(u)$ is equal to $\mathbf{D}_k[u](h)$ of Equation 10 in every steps. In the first round, $\mathbf{F}_1(u) = \alpha \mathbf{x}_h(u)$ and $\mathbf{D}_1[u](h) = \alpha \mathbf{x}_u(h)$, which are identical. In step $k+1$, $\mathbf{F}_{k+1}(u) = (1-\alpha) \sum_{v \in Out(u)} \frac{\mathbf{F}_k(v)}{|Out(u)|} + \alpha \mathbf{x}_h(u)$, meanwhile we have $\mathbf{D}_{k+1}[u](h) = (1 - \alpha) \sum_{v \in Out(u)} \frac{\mathbf{D}_k[v](h)}{|Out(u)|} + \alpha \mathbf{x}_u(h)$, and we can conclude that they are identical in each step. □

The space complexity of the improved skeleton computation method is only $O(|V|)$. To obtain the full vector skeleton vector for a node u, we need to run this algorithm for each $h \in H$ ($|H|$ times in total).

In GPA, given a hub node h, we compute $\mathbf{s}_u^H(h)$ w.r.t. the whole graph for each node u according to Equation 8. In HGPA, given a hub node h at level m, we first identify the subgraph at level m where h resides ($G_m^{(h)}$), and we compute $\mathbf{s}_u^H[G_m^{(h)}](h)$ w.r.t. the subgraph $G_m^{(h)}$ for each node u in $G_m^{(h)}$. Note that based on Equation 8 there is no dependency among machines, and each machine can do the computation independently for the nodes assigned to it, and thus there is no network communication required.

6. EXPERIMENTS

6.1 Experimental Setup

Datasets We conduct experiments on five public real-life network datasets:

Email[1]. This dataset is generated using email data from a large European research institution. This graph comprises 265,214 nodes and 420,045 edges.

Web[2]. This dataset is generated using web pages from the Google programming contest in 2002. This graph comprises 875,713 nodes and 5,105,039 edges.

Youtube.[3] This graph is generated from the video and user information from the well-known video sharing website Youtube, and it comprises 1,134,890 nodes and 2,987,624 edges.

PLD.[4] This dataset is extracted using hyperlink pages from the Web corpus released by the Common Crawl Foundation in 2014. As the whole dataset is too large, we extract a sample graph comprising 3,000,000 nodes and 18,185,350 edges. We also report the results on the full graph in Appendix B, which contains 101M nodes and 1.94B edges and is denoted by PLD_full.

Meetup. This dataset is a social graph crawled from Meetup[5].We collect various sizes of events to build the graphs of different sizes for studying scalability of our algorithm. The detail of this dataset is shown in Section 6.2.7.

Algorithms. We evaluate the performance of the distributed algorithm HGPA (Section 4.4). We also compare with the algorithm GPA (Section 3) and the distributed PPV computation using power iteration based on Pregel+ [48] and Blogel [47]. Note that there is no better baseline for distributed PPV computation. We evaluate the performance of these algorithms from the following aspects: the space cost, the efficiency of computing the PPV for a query node, and the communication cost during the online PPV computing. We also compare HGPA with power iteration and a state-of-the-art approximate PPV computing method [49] in a centralized setting.

Accuracy Metric. To show the exactness of our proposed algorithms, we compare with *power iteration* method using average L_1-norm and L_∞-norm metrics, which are also used to evaluate PageRank algorithm performance [7]. Given the PPV vectors $\mathbf{r}_u$ and $\bar{\mathbf{r}}_u$ computed by different algorithms, the average L_1-norm is defined as $L_1^{avg}(\mathbf{r}_u, \bar{\mathbf{r}}_u) = \Sigma_{v \in V} |\mathbf{r}_u(v) - \bar{\mathbf{r}}_u(v)|/|V|$ and the L_∞-norm is defined as $L_\infty(\mathbf{r}_u, \bar{\mathbf{r}}_u) = \max_{v \in V} |\mathbf{r}_u(v) - \bar{\mathbf{r}}_u(v)|$.

Query Generation. We randomly choose 1000 nodes as query nodes for each graph, and report the average performance over all queries. In all experiments, we only focus on single node queries.

Parameters. By default, we set the number of machines as 6. We use the two-way hierarchical partition method [26] for HGPA. We set $\epsilon = 10^{-4}$ for all algorithms evaluated in the experiment, following the previous work [25]. We set teleport probability $\alpha = 0.15$ for all experiments, as it is widely used in previous work.

Setup. All algorithms were implemented in C++ complied with GCC 4.8.2 and run on Linux. The experiments are conducted in a cluster consisting of 10 machines, each machine with a 2.70GHz CPU and 64GB of main memory. The machines are interconnected by a 100MB TP-LINK switch. All the pre-computations are performed using 4 threads on each machine. For each experiment, we run our algorithm 100 times and report the average.

[1]http://snap.stanford.edu/data/email-EuAll.html
[2]http://snap.stanford.edu/data/web-Google.html
[3]http://snap.stanford.edu/data/com-Youtube.html
[4]http://webdatacommons.org/hyperlinkgraph
[5]http://www.meetup.com

6.2 Experimental Results

6.2.1 *Number of Hub nodes in* HGPA

For HGPA, we perform the hierarchical partitioning until no edges exist within each subgraph. This is because further partitioning cannot gain more improvement. We show the effect of partitioning levels in Section 6.2.4. We partition Email into 5 levels (yielding 2^5 = 32 leaf-level subgraphs), Web into 12 levels (4,096 leaf-level subgraphs), and both Youtube and PLD are partitioned into 15 levels (32,768 leaf-level subgraphs).

The pre-computation space and time cost of HGPA depends on the number of hub nodes in each level. We list the number of hub nodes in each level obtained by multi-way hierarchical partitioning of the four datasets in Tables 2–5, where the original graph is at level 0 and the leaf subgraphs are in the maximum level. It can be observed that the number of hub nodes is always much smaller than the total number of nodes in all datasets.

Table 2: Hub Nodes in Each Level on Email

Level	0	1	2	3	4
Hub Number	1,208	84	34	16	18

Table 3: Hub Nodes in Each Level on Web

Level	0	1	2	3	4	5
Hub Number	6,763	2,108	1,150	694	464	3,738
Level	6	7	8	9	10	11
Hub Number	6,918	6,919	12,561	6,795	5,867	15,115

Table 4: Hub Nodes in Each Level on Youtube

Level	0	1	2	3	4
Hub Number	30,932	9,055	8,966	5,287	4,041
Level	5	6	7	8	9
Hub Number	2,431	1,748	1,222	875	925
Level	10	11	12	13	14
Hub Number	979	1,140	1,593	2,577	4,099

Table 5: Number of Nodes in Each Level on PLD

Level	0	1	2	3	4
Hub Number	42,101	23,829	20,276	11,827	10,908
Level	5	6	7	8	9
Hub Number	16,583	12,967	9,235	6,046	5,135
Level	10	11	12	13	14
Hub Number	11,119	16,438	6,892	8,927	7,830

6.2.2 *Comparison of* GPA *and* HGPA

This experiment is to compare the performances of GPA and HGPA using default parameters. We report the maximum runtime across all machines as the overall query processing time. For the space cost, we report the maximum space used overall all machines. The pre-computation time is evaluated as the maximum time across all machines. The communication cost is reported as the size of all the data received by the coordinator during the query processing.

Figure 9 shows the results of the comparison between GPA and HGPA on Web. We observe from the results that GPA runs a bit slower than HGPA, because HGPA is more load balanced. The maximum space cost and offline pre-computation time of HGPA are better than that of GPA, and this is consistent with the theoretical analysis in Section 4.5. The theoretical communication cost of HGPA and GPA are the same, while we can observe from the

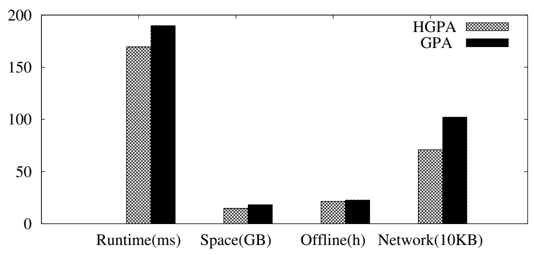


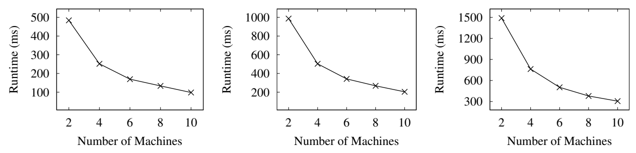
Figure 9: Algorithms Comparison on Web

figure that in practice HGPA spends less network cost than GPA. Similar results are observed on the other datasets, and are thus not reported.

Since HGPA outperforms GPA in terms of all the aspects concerned, we omit the results of GPA in the subsequent experiments.

6.2.3 Effects of Number of Machines

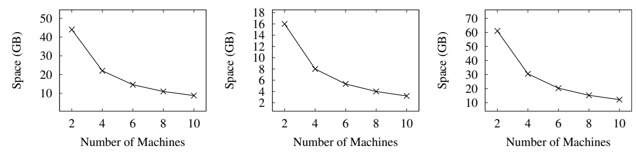
This experiment is to study the effect of machines number on HGPA.



(a) Web (b) Youtube (c) PLD

Figure 10: HGPA Runtime

Figures 10(a)–10(c) show the runtime of HGPA on Web, Youtube, and PLD when we vary the number of machines, respectively. We observe that the query processing time drops significantly as the number of machines increases. When we double the number of machines the runtime is nearly reduced by half. The reason is that the computation is evenly distributed to multiple machines in HGPA, and thus the algorithm is highly load-balanced.



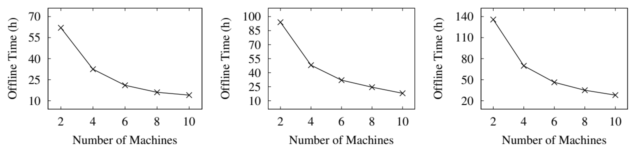
(a) Web (b) Youtube (c) PLD

Figure 11: HGPA Space Cost

Figures 11(a)–11(c) show the space cost of HGPA. Note that each machine only stores the pre-computed vectors of nodes assigned to it. We report the maximum space cost over all machines. It is observed that the result is as expected, where the maximum space cost is reduced when the number of machines increases. There is no redundant information shared between different machines.

The pre-computation time is shown in Figures 12(a)–12(c). Each machine only needs to do the pre-computation for the nodes stored on it. HGPA is load balanced, and thus the space costs of pre-computation is nearly linear to the number of machines used.

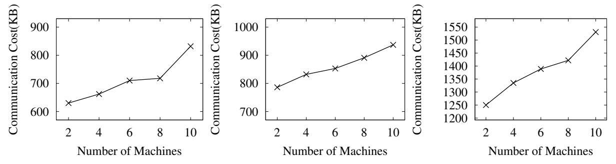
Figures 13(a)–13(c) show the communication cost incurs at query time of HGPA. We notice even for the largest dataset working on 10 machines HGPA only has less than 2MB network cost. It can be observed that the communication cost increases as more machines are employed, which is consistent with our analysis in Theorem 4.



(a) Web (b) Youtube (c) PLD

Figure 12: HGPA Pre-Computation Time

It is obvious since more machines communicate with the coordinator.



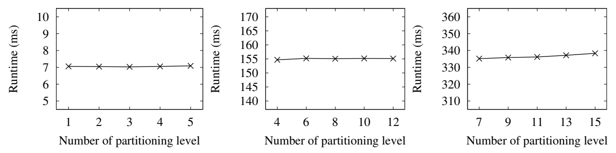
(a) Web (b) Youtube (c) PLD

Figure 13: HGPA Communication Cost

6.2.4 Effect of Partitioning Levels

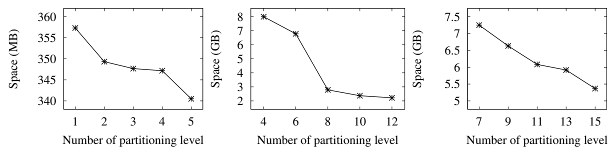
This set of experiments is to study the effect of the level of the hierarchy of subgraphs on the performance of HGPA. Figures 14(a)–14(c) show the runtime for computing PPVs for query nodes on Email, Web, and Youtube, respectively. Figures 15 and 16 show the space and time cost of pre-computation on the three datasets.

The space and time of pre-computation drop significantly as we increase the number of levels of the graph hierarchy. As the number of levels increases, the number of subgraphs in the hierarchy increases exponentially, and thus the size of the subgraphs in leaf-level decreases greatly. However, when in a certain level there exists few or no edges within each subgraph, further partitioning is unnecessary because it cannot reduce space cost any more. According to Equation 7, using more levels needs more computation to construct the PPV, and thus causes slightly longer query processing time, which can be observed in Figure 14. The communication cost is almost not affected by the number of partition levels, and thus is not reported.



(a) Email (b) Web (c) Youtube

Figure 14: HGPA Runtime



(a) Email (b) Web (c) Youtube

Figure 15: HGPA Space Cost

6.2.5 Effect of Multi-way Partitioning

This experiment is to study the effect of the partition strategy. We take Web as an example in this section, and we observe the similar results on the other datasets. We partition Web into 2,4,8,16 and

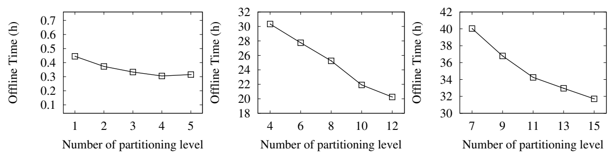

(a) Email (b) Web (c) Youtube

Figure 16: HGPA Pre-Computation Time

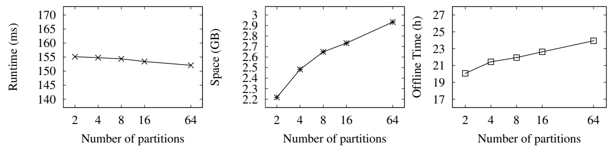

(a) Runtime (b) Space Cost (c) Offline Time

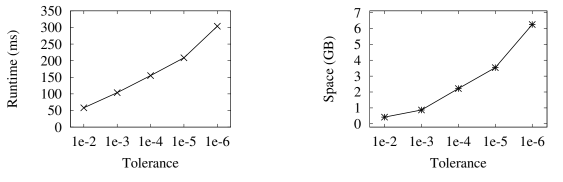
Figure 17: Web (HGPA)

64 subgraphs in each level and we study the time and space cost for both pre-computation and query processing.

Figure 17(a) shows the query processing time. We observe that the runtime slightly decreases when the number of partitions in each level increases. Figures 17(b) and 17(c) show the space and time cost of the pre-computation. We can see that with more partitions in each level, the cost of pre-computation increases greatly. We choose the 2-way partitioning strategy by default in this paper, as the runtime is close to that of more partitions and its pre-computation space and time cost is the smallest. The partition strategy almost does not affect the communication cost, and thus the we do not report it.

6.2.6 *Effect of Tolerance* ϵ

This experiment is to study the effect of tolerance ϵ on the performance of HGPA. For the offline pre-computation, the tolerance decides when the iteration terminates. Again, we take Web as an example in this section, and we observe similar results on the other datasets. Figures 18(a)–18(d) show the query processing time, the space and time of pre-computation, and the communication cost of HGPA on Web. We observe that all the four measures increase as we take a smaller tolerance. With a high accuracy, more results of small values are generated, and thus it costs more time for pre-computing the vectors and more space for storing them. At query time, a high accuracy makes the vectors pre-computed have larger size, and thus both PPV construction time and the communication cost increase.


(a) Runtime (b) Space Cost

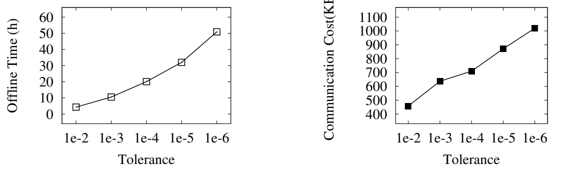

(c) Offline Time (d) Communication Cost

Figure 18: Web (HGPA)

We also study the exactness of HGPA when we vary the tolerance ϵ. We take the power iteration method as baseline, treating its PPV result as exact. For each query, we compare the PPV computed by HGPA and the power iteration method under the same tolerance. We report the average L_1 and L_∞ of the difference of two vectors. The results on datasets Email and Web are shown in Figures 19(a) and 19(b). It can be observed from the figures that, as ϵ decreases both measures on the differences of two vectors become smaller, which is as expected. The ℓ-norms are nearly in the same order of magnitude with the tolerance. This means we can always obtain a more exact PPV result by setting a smaller tolerance.

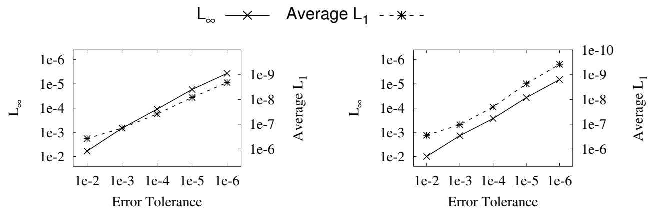

(a) Email (b) Web

Figure 19: ℓ norm Accuracy

6.2.7 *Scalability*

Table 6: Graph sizes for scalability study (Meetup)

Graph ID	# Nodes	# Edges
M1	997,304	82,966,338
M2	1,197,009	107,393,088
M3	1,396,054	129,774,158
M4	1,596,455	163,320,390
M5	1,796,226	194,083,414

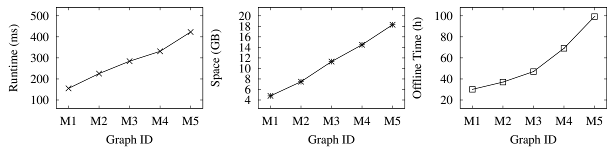

(a) Runtime (b) Space Cost (c) Offline Time

Figure 20: Scalability (Meetup)

This experiment is to study the scalability of HGPA with the size of graphs. The difficulty is that there does not exist a group of graphs of different sizes but with similar properties. To this end, we build graphs of different sizes by taking different number of meetup events. The sizes of these graphs are listed in Table 6. In this experiment, we fix the number of machines used to be 10 for all graphs.

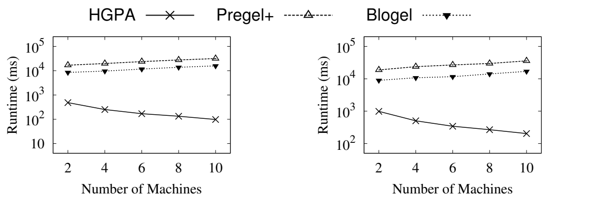
Figure 20(a) shows the runtime of computing PPVs for query nodes. We observe that the query processing time increases almost linearly with the size of graphs. Figures 20(b) and 20(c) report the space cost and time of pre-computation of HGPA for graphs of different sizes. We can see that both the space cost and time increase almost linearly as we increase the size of graphs.

6.2.8 *Comparison with General Distributed Graph Processing Systems*

Baseline. To the best of our knowledge, there exists no work for distributed exact PPV computation. Many distributed graph computation platforms [47, 48] take the PageRank (PR) problem as a basic graph computing application. In these platforms, the PR problem is usually solved by the *power iteration* method. Since the personalized PageRank is derived from the PR problem [38], we can compute PPV by implementing the power iteration method as well on these platforms, which are used as baselines.

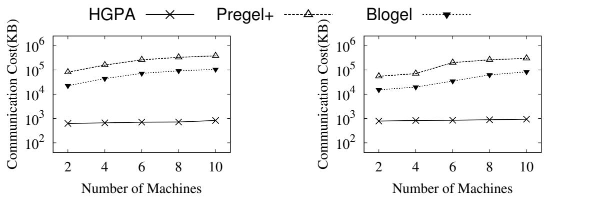
We compare HGPA with the power iteration method implemented on Pregel+ [48] and Blogel [47], which are well-known open-source distributed graph computation platforms (we denote the two algorithms by Pregel+ and Blogel, respectively). It is shown [48] that Pregel+ outperforms other Pregel systems such as Giraph [19] and GPS [41]. The follow-up work Blogel [47] breaks the bottlenecks of vertex-centric models such as Pregel. We compare the runtime and communication cost of HGPA with Pregel+ and Blogel under the same tolerance, and the result is shown in Figures 21(a)– 22(b).

It can be seen that our algorithm is faster than Pregel+ and Blogel by orders of magnitude on Web and Youtube, and HGPA outperforms Pregel+ by at least two orders of magnitude in terms of communication cost. We observe that the runtime and communication cost of Pregel+ and Blogel increase when the number of machines increases. The reason is as follows: Pregel+ is designed based on the general bulk synchronous parallel (BSP) model. It sends massages from vertex to vertex in each iteration of the BSP step, and when the vertices are on different machines it needs a lot of communications between machines. As the number of machines increases, the number of massages increases which costs more communication time. We observe the same phenomenon on Blogel. However it always outperforms Pregel+ in terms of the runtime and communication cost. This is because Blogel is based on the block-centric model, and it sends massages from block to block. Our proposed algorithm HGPA significantly outperforms the algorithms implemented on Pregel+ and Blogel for computing PPVs. As can be observed from the figures, the runtime of HGPA decreases significantly as the number of machines increases. This is achieved by avoiding the huge communication costs, and thus is more suitable for the online PPV applications.



(a) Web (b) Youtube

Figure 21: Runtime



(a) Web (b) Youtube

Figure 22: Communication Cost

6.2.9 Performance under Centralized Setting

HGPA can also be implemented on a single machine. We compare the runtime of HGPA in a centralized setting with the power iteration method, under the same error tolerance, and the result is shown in Figure 23. It can be seen that our algorithm is at least 3.5 times faster than the power iteration method. On Email and Web the speedup is much more significant. It demonstrates that HGPA can achieve comparable performance in terms of runtime as the algorithm proposed in the work [35] on a single machine.

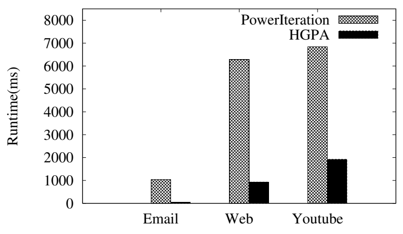
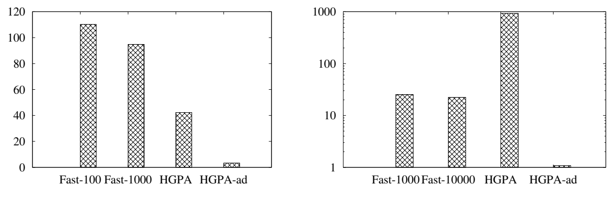


Figure 23: Compared to Power Iteration

We also compare our work with the state-of-the-art approximate PPV computation method *FastPPV* [49]. In *FastPPV*, the PPV scores less than 10^{-4} are discarded. It is shown in [11, 12] that removing the small values only sacrifices little accuracy. To compare with the approximate method, we also discard the offline scores that are less than 10^{-4}, and this adapted method is denoted by HGPA_ad. Since in *FastPPV* the number of hub nodes is a parameter that affects the trade-off between accuracy and runtime, we compare with this algorithm using different numbers of hub nodes.

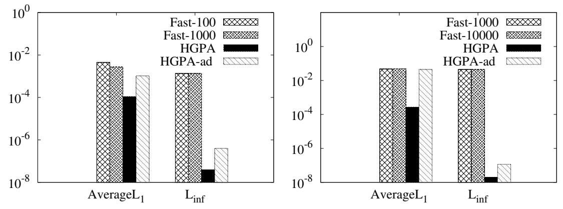


(a) Email (b) Web

Figure 24: Runtime (ms)

Figure 24 shows the runtime on Email and Web datasets. We use *Fast_h* to denote the method of *FastPPV* using h hub nodes. We notice that our exact method HGPA is faster than *FastPPV* on small dataset and slower than *FastPPV* on the large one. By discarding tiny PPV scores, we notice that the adapted method HGPA_ad runs faster than *FastPPV* by orders of magnitude on both datasets. Note that in this set of experiments our algorithm is implemented on a single machine. On the distributed computing platforms, our algorithm will be much faster.

We next evaluate the accuracy of the four algorithms and use the result computed by power iteration as exact. We use average L_1 and L_∞ as we used in Section 6.2.6. Figure 25 show the ℓ norm accuracy comparison on Email and Web datasets. On all measures, it is not surprised to observe that HGPA is much better than *FastPPV* since the computing model of our algorithm is exact. We notice that our approximate HGPA_ad also consistently outperforms *FastPPV* in terms of accuracy.



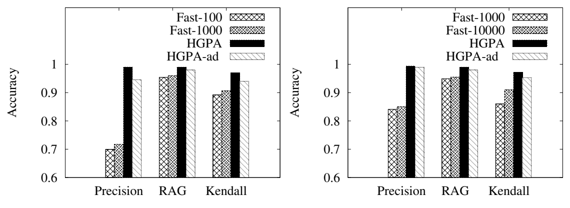
(a) Email (b) Web

Figure 25: ℓ norm Accuracy

In summary, our proposed distributed algorithms GPA and HGPA have low communication cost and are load-balanced with acceptable pre-computation and space cost. HGPA consistently outperforms GPA, and it outperforms the power iteration method implemented on general graph processing platforms such as Pregel+ and Blogel by orders of magnitude. In the centralized setting, HGPA can achieve comparable query time with the exact method [35] and the approximate method [49]. Its adapted version HGPA_ad is able to outperform the approximate method [49] in terms of both efficiency and accuracy.

6.2.10 *Exact vs Approximate*

This experiment extends the study of Section 6.2.9 to show the advantage of exact PPVs compared to approximate PPVs. We use two accuracy metrics, i.e., Precision and *Kendall's* τ (by following the work [11, 49]), to compare the top-100 nodes obtained from each algorithm with the result of the power iteration method. In a nutshell, Precision is based on the value of top-k PPV results, and Kendall is based on the percentage of the correct node pair order. The detailed definition can be referred to [11]. Figure 26 shows the results on Email and Web. We observe that on both metrics HGPA performs much better than FastPPV [49], and even the adaptive algorithm HGPA_ad achieves nearly full score. This indicates that about 30% of the top-100 nodes returned by FastPPV are wrong, and about 10% node pairs are ordered incorrectly.



(a) Email (b) Web

Figure 26: Accuracy

7. RELATED WORK

The personalized PageRank(PPR) has been widely used in various fields of applications, such as community detection [3,21], link prediction [4], and recommendation systems [22,27].

Exact PPV computation. PPR was first proposed by Jeh and Widom [25]. It usually needs to be computed *in an online manner, and thus has a high requirement for efficiency.* A straightforward way of computing the Personalized PageRank Vector for a given set of nodes is the power iteration method, which is prohibitively expensive in time and is not suitable even for offline scenarios.

Due to the hardness of computing the exact PPV, Jeh and Widom propose to limit the preference nodes in a subset of a specified hub node set [25]. Therefore, this approach cannot be used to compute the exact PPV for arbitrary preference node set. We will introduce more about the algorithm in Section 2.

Maehara et al. [35] propose an iteration-based method that computes exact PPVs by exploiting graph structures. They decompose a graph into a core and a small tree-width graph. The two components are processed differently, and the PPV is constructed using the processed results. Unfortunately, this approach is not able to compute exact PPVs in a distributed manner. According to the experimental study, this method is about five times faster than the power iteration method. As shown in the experimental study, our distributed algorithm HGPA is faster than the power iteration method implemented on general graph processing engines by orders of magnitude.

Approximate methods for PPV. Most of existing studies compute PPVs approximately to trade for efficiency. Some proposals (e.g., [6, 14]) utilize the Monte Carlo simulation methods, while some other studies (e.g., [44]) utilize the matrix factorization. Zhu et al. [49] propose an approximate method based on the concept of the inverse P-distance [25]. They first partition the tours into different tour sets according to their importance. Then, they design an algorithm to aggregate the contribution of tours from the most important ones to less important ones. In the experimental study, we show that our exact algorithm HGPA is able to achieve similar query time to the approximate algorithm [49] while obtaining much better accuracy, and an adapted version of HGPA outperforms the approximate algorithm [49] in terms of both efficiency and accuracy.

Top-k and node-to-node search for PPV. Some approaches [16–18,46] aim to find a small part of the PPV. That is, for a query node, they only identify its top-k relevant nodes and omit the other nodes. Lofgren et al. [31,32] study how to estimate the node-to-node PPV score. Given a query node u, a target node v and threshold δ, they estimate whether $\mathbf{r}_u(v) > \delta$ is true. All these methods cannot be used for computing the whole PPV w.r.t. a given query node set. However, finding top-k or estimating node-to-node PPV value is insufficient for many applications (e.g., [4,8,11]) which require the PPV scores of all nodes.

Distributed PPV computation. There also exist studies on distributed computation of approximate PPVs. In particular, Bahmani et al. [5] proposes a distributed algorithm based on MapReduce utilizing Monte Carlo simulation, which has no guaranteed error bound. The general graph processing engines, such as Pregel [36], Pregel+ [48] and Blogel [47], can be used for various distributed graph processing. As shown in the work [36], the power iteration method can be implemented on Pregel to compute PageRank, and thus PPVs. However, using these engines always induces multiple rounds of communications between machines; therefore, the communication cost is large and the query processing is slow, which make them impractical for applications of PPVs that have a high requirement on efficiency. In contrast, our proposed algorithms only require the communication between the machines and the coordinator once at query time. As shown in the experimental study, our algorithm significantly outperforms the PPV computation based on Pregel+ [48] and Blogel [47].

8. CONCLUSION

In this paper, we propose novel and efficient distributed algorithms that are able to compute the exact PPV for all nodes. The proposed algorithms can be implemented on a general coordinator-based share-nothing distributed computing platform. The processors only need to communicate with the coordinator once at query time in our algorithms. We first develop the algorithm GPA that works based on subgraphs. To further improve the performance, we propose HGPA based on a hierarchy of subgraphs, which has smaller space cost and better load balance and efficiency than GPA. The experimental study shows that HGPA has excellent performance in terms of efficiency, space cost, communication cost, and scalability. Our algorithm HGPA outperforms the PPV computation using power iteration based on two distributed graph processing systems by orders of magnitude. Further, an adapted version of HGPA outperforms the state-of-the-art approximate algorithm [49] in terms of both efficiency and accuracy. In the future we would like to further study how to reduce the number of hub nodes.

9. ACKNOWLEDGMENT

This work was carried out at the Rapid-Rich Object Search(ROSE) Lab at the Nanyang Technological University, Singapore. The ROSE Lab is supported by the National Research Foundation, Singapore, under its Interactive Digital Media(IDM) Strategic Research Programme.

10. REFERENCES

[1] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[2] I. Antonellis, H. G. Molina, and C. C. Chang. Simrank++: Query rewriting through link analysis of the click graph. *PVLDB*, 1(1):408–421, 2008.

[3] H. Avron and L. Horesh. Community detection using time-dependent personalized pagerank. In *ICML*, pages 1795–1803, 2015.

[4] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *WSDM*, pages 635–644, 2011.

[5] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *KDD*, pages 973–984, 2011.

[6] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.

[7] B. Bahmani, R. Kumar, M. Mahdian, and E. Upfal. Pagerank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 24–32. ACM, 2012.

[8] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[9] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[10] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[11] S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW*, pages 571–580, 2007.

[12] S. Chakrabarti, A. Pathak, and M. Gupta. Index design and query processing for graph conductance search. *The VLDB Journal*, 20(3):445–470, 2011.

[13] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[14] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[15] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[16] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5):442–453, 2012.

[17] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka. Efficient ad-hoc search for personalized pagerank. In *ICDM*, pages 445–456, 2013.

[18] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *KDD*, pages 15–23, 2012.

[19] A. Giraph. http://giraph.apache.org/.

[20] D. Gleich, L. Zhukov, and P. Berkhin. Fast parallel pagerank: A linear system approach. *Yahoo! Research Technical Report YRL-2004-038, available via http://research. yahoo. com/publication/YRL-2004-038. pdf*, 13:22, 2004.

[21] D. F. Gleich and C. Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *KDD*, pages 597–605, 2012.

[22] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: the who to follow service at twitter. In *WWW*, pages 505–514, 2013.

[23] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[24] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *KDD*, pages 538–543, 2002.

[25] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[26] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[27] H. Kim and A. El-Saddik. Personalized pagerank vectors for tag recommendations: inside folkrank. In *RecSys*, pages 45–52, 2011.

[28] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008.

[29] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[30] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. *PVLDB*, 1(1):422–433, 2008.

[31] P. Lofgren, S. Banerjee, and A. Goel. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*, pages 163–172, 2016.

[32] P. Lofgren, S. Banerjee, A. Goel, and S. Comandur. FAST-PPR: scaling personalized pagerank estimation for large graphs. In *KDD*, pages 1436–1445, 2014.

[33] L. Lovász and M. D. Plummer. Matching theory. *New York*, 1986.

[34] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[35] T. Maehara, T. Akiba, Y. Iwata, and K. Kawarabayashi. Computing personalized pagerank quickly by exploiting graph structures. *PVLDB*, 7(12):1023–1034, 2014.

[36] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[37] 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.

[38] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Technial Report*, 1999.

[39] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[40] J. X. Parreira, C. Castillo, D. Donato, S. Michel, and G. Weikum. The juxtaposed approximate pagerank method for robust pagerank approximation in a peer-to-peer web search network. *The VLDB Journal*, 17(2):291–313, 2008.

[41] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22.

ACM, 2013.

[42] K. Shin, J. Jung, S. Lee, and U. Kang. Bear: Block elimination approach for random walk with restart on large graphs. In *SIGMOD*, pages 1571–1585, 2015.

[43] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005.

[44] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006.

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

[46] Y. Wu, R. Jin, and X. Zhang. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *KDD*, pages 1139–1150, 2014.

[47] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[48] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

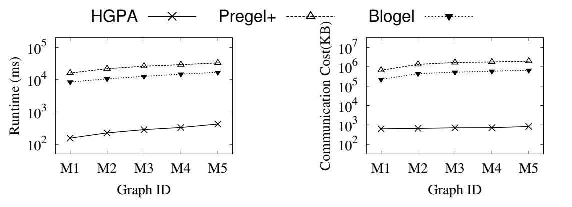
[49] F. Zhu, Y. Fang, K. C.-C. Chang, and J. Ying. Incremental and accuracy-aware personalized pagerank through scheduled approximation. *PVLDB*, 6(6):481–492, 2013.

[50] Y. Zhu, S. Ye, and X. Li. Distributed pagerank computation based on iterative aggregation-disaggregation methods. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 578–585. ACM, 2005.

APPENDIX

A. SCALABILITY STUDY ON GENERAL GRAPH PROCESSING SYSTEMS

This experiment is to study the scalability of the power iteration method on Pregel+ and Blogel. We use the same datasets used in Section 6.2.7 for studying the scalability of HGPA. Figure 27(a) shows the runtime of computing PPV on the two platforms compared to our HGPA method. We observe that the runtime of Pregel+ and Blogel increases linearly with the size of graphs, and HGPA is orders of magnitude faster than them. Figure 27(b) reports the communication cost during the query processing. Similarly, we can observe that the communication cost of Pregel+ and Blogel grows linearly with the graph size. It is because the communication of Pregel+ and Blogel is based on edges, and thus it is linear to the number of edges.

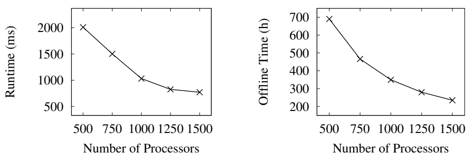


(a) Runtime (b) Communication Cost

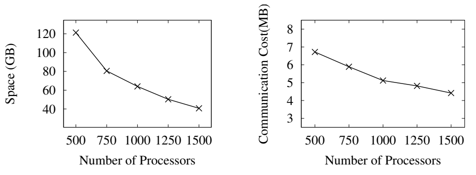
Figure 27: Scalability (Meetup)

B. EXPERIMENTS ON LARGE GRAPH

This experiment is to study the performance of HGPA on the large graph data set PLD_full, which contains 101 millions of nodes and 1.94 billions of edges. To conduct this experiment, we deploy our algorithm on Amazon EC2[6] using up to 24 instances, each of which has 64 processors (1500 processors in total). We set ϵ as 10^{-2} to save the cost of resources, and other settings are same with our other experiments.



(a) Runtime (b) Offline



(c) Space (d) Communication Cost

Figure 28: HGPA Performance on PLD_full

The result is shown in Figure 28 as the number of processors is varied from 500 to 1500. We observe that: although the dataset PLD_full is over 50 times larger than the sampled PLD dataset, our algorithm HGPA is still able to perform well in terms of query time, pre-processing cost, and communication cost. We also observe that although the network communication cost is high, the runtime is under 2000 ms. This indicates that even for large graphs the network communication does not affect the runtime very much, because our algorithm only requires one time of communication between the coordinator and each machine.

C. POWER ITERATION IMPLEMENTATION

In this section, we introduce how the power iteration algorithm is implemented. We present the centralized version, and it can be easily modified to work in parallel on Pregel+ and Blogel. We utilize the adjacency list to implement the power iteration algorithm, and this method is also used in the work [49]. The pseudo code is shown in Algorithm 2.

To improve the efficiency, we use a queue $valuedNodes$ to track the nodes with non-zero value, and only the nodes in $valuedNodes$ and their neighbors are visited. In each iteration, the nodes in the queue either teleport to the query node(line 12) or randomly surf to their out-neighbors(lines 17–21). Note that in lines 14–16, we deal with the dangling nodes (nodes without out-neighbors) by adding an arc to query node q. At last, we check whether the stop condition is fulfilled(lines 25–28).

D. HUB NODE SELECTION

The graph partitioning problem aims to remove minimum number of edges/nodes from a given graph such that the graph becomes disconnected. It is shown that finding such edge separators or vertex separators is NP-hard [10]. The best approximate algorithm of finding the vertex separators has an approximation ratio of $O(log^2|V|)$ [1, 13] with high computational complexity. For planar graphs, there exists $O(\sqrt{(|V|)})$ theoretical bound [29] for the number of vertex separators. For general real-world networks, the number of hub nodes is usually small, e.g., social graphs are often organized into communities with large internal density of edges

[6]http://aws.amazon.com/ec2/

Algorithm 2: Power Iteration

```
input  : Graph $G(V, E)$, query node $q$, error tolerance $\epsilon$
output: Personalized PageRank Vector $\vec{ppv}$
1  $\vec{ppv} \leftarrow \vec{0}$;
2  initialize Queue $valuedNodes$;
3  initialize Array $inQueue$ of size $|V|$ with $False$;
   /* First round iteration */
4  $ppv[q] \leftarrow 1$;
5  push $q$ into $valuedNodes$;
6  $inQueue[q] \leftarrow True$;
7  $converged \leftarrow False$;
8  while not converged do
9      $tmpVec \leftarrow \vec{0}$;
10     $tmpQueue \leftarrow valuedNodes$;
11     foreach $u$ in $valuedNodes$ do
           /* teleport to origin with
              probability $\alpha$ */
12         $tmpVec[q] \leftarrow tmpVec[q] + ppv[u] * \alpha$;
13         $nbs \leftarrow number\ of\ out\ neighbors\ of\ v$;
14         if $nbs = 0$ then
15             $tmpVec[q] \leftarrow tmpVec[q] + ppv[u] * (1 - \alpha)$;
16             Continue;
17         foreach out-neighbor $v$ of $u$ do
18             $tmpVec[q] \leftarrow tmpVec[q] + ppv[u] * (1 - \alpha)/nbs$;
19             if not $inQueue[v]$ then
20                 $inQueue[v] \leftarrow True$;
21                 push $v$ into $tmpQueue$;
22     foreach $u$ in $tmpQueue$ do
23         push $u$ into $valuedNodes$;
       /* Check convergence */
24     $converged \leftarrow True$;
25     foreach $u$ in $valuedNodes$ do
26         if $|ppv[u] - tmpVec[u]| > \epsilon$ then
27             $converged \leftarrow False$;
28         $ppv[u] \leftarrow tmpVec[u]$;
29 return $\vec{ppv}$;
```

and sparse edges between communities [15, 28]. For the matrix-based methods for PPV computation [16,42], they reorder the matrix, which partition the graph into m subgraphs (each of them is a small matrix) and leave the hub nodes in a matrix. The reordering also assumes the number of hub nodes is not large.

Due to the hardness of this problem, most graph partitioning algorithms aim to minimize the number of hub edges approximately. It is shown [26] that the METIS method performs well on minimizing the number of hub edges and balancing the size of subgraphs. We first adopt this partition algorithm to minimize the number of hub edges, and we then try to find the minimum number of hub nodes from these hub edges. The problem of finding the minimum number of hub nodes from hub edges is equivalent to the well-known *Vertex Cover* problem, which is also NP-hard. Specifically, given the hub edges, we choose some endpoints of the hub edges to be hub nodes, such that each edge is covered by at least one hub node. In this way, if we remove all the hub nodes, every hub edges will also be removed and the graph is partitioned into k parts. We use the approximate vertex cover algorithm proposed in the work [39] to approximately find the minimum set of nodes that can cover these hub edges.

It is worth mentioning that though we use approximate algorithm to find hub nodes, our algorithm for PPV computation is still exact only if these nodes can separate the graph. A better way of selecting the hub nodes is an area of future work.

E. PARTIAL & SKELETON VECTORS COMPUTATION

Both the partial and hubs skeleton vectors are computed in iterative ways. Note that these vectors could be rounded to any given error, known as $tolerance$, which is also used in the power iteration algorithm. The precision can be guaranteed, and such algorithms are regarded as exact in the literatures [25, 49]. In this paper, exact PPV means that we achieve the same results as the algorithms proposed by Jeh and Widom [25]. Details about our distributed pre-computation of partial vectors and skeleton vectors will be introduced in Section 5.

E.1 Partial Vector computation

We introduce the *selective expansion algorithm* [25] to compute $\mathbf{p}_u^H$ w.r.t. node u. Two intermediate vectors $\mathbf{D}_k[u]$ and $\mathbf{E}_k[u]$ are maintained, where k represents the k^{th} iteration. Initially, $\mathbf{D}_0[u] = \mathbf{0}$ and $\mathbf{E}_0[u] = \mathbf{x}_u$, where $\mathbf{x}_u$ is a basic vector with zero filled except $\mathbf{x}_u(u) = 1$. Then, Equation 9 illustrates the iteration method, where $Out(v)$ denotes the set of out-going neighbors of v.

$$\begin{aligned}\mathbf{D}_{k+1}[u] &= \mathbf{D}_k[u] + \sum_{v \in V-H} \alpha \mathbf{E}_k[u](v)\mathbf{x}_v \\ \mathbf{E}_{k+1}[u] &= \mathbf{E}_k[u] - \sum_{v \in V-H} \mathbf{E}_k[u](v)\mathbf{x}_v \\ &+ \sum_{v \in V-H} \frac{1-\alpha}{|Out(v)|} \sum_{i=1}^{|Out(v)|} \mathbf{E}_k[u](v)\mathbf{x}_{Out_i(v)}\end{aligned} \tag{9}$$

During the iterations, $\mathbf{D}_k[u]$ would converge to $\mathbf{p}_u^H$, and $\mathbf{E}_k[u]$ would approach $\mathbf{0}$, which represents the difference between $\mathbf{D}_k[u]$ and $\mathbf{p}_u^H$. $\mathbf{D}_k[u]$ is a lower-approximation of $\mathbf{p}_u^H$. The computation is terminated when the value of $\mathbf{E}_k[u]$ approaches $\mathbf{0}$, which means we can obtain $\mathbf{p}_u^H$ to any arbitrary precision. More precisely, we set an error tolerance bound ϵ and terminate the iteration when $\mathbf{E}_k[u](v) \leq \epsilon, \forall u, v \in V$.

E.2 Skeleton Vector computation

We introduce the basic *dynamic programming algorithm* [25] that computes $\mathbf{s}_u^H$ w.r.t. node u. We also maintain two intermediate vectors $\mathbf{D}_k[u]$ and $\mathbf{E}_k[u]$, where initially $\mathbf{D}_0[u] = \mathbf{0}$ and $\mathbf{E}_0[u] = \mathbf{x}_u$. Then, the iteration is done as follows:

$$\begin{aligned}\mathbf{D}_{k+1}[u] &= \frac{1-\alpha}{|Out(u)|} \sum_{i=1}^{|Out(u)|} \mathbf{D}_k[Out_i(u)] + \alpha \mathbf{x}_u \\ \mathbf{E}_{k+1}[u] &= \frac{1-\alpha}{|Out(u)|} \sum_{i=1}^{|Out(u)|} \mathbf{E}_k[Out_i(u)]\end{aligned} \tag{10}$$

During the iterations, $\mathbf{D}_k[u]$ would converge to $\mathbf{s}_u^H$, and $\mathbf{E}_k[u]$ would approach $\mathbf{0}$, which represents the difference between $\mathbf{D}_k[u]$ and $\mathbf{s}_u^H$. We terminate the computation when the value of $\mathbf{E}_k[u]$ approaches 0. That is, we set an error tolerance ϵ, and it is treated as converged when each value in $\mathbf{E}_k$ is less than ϵ.