Two Parallel PageRank Algorithms via Improving Forward Push

Qi Zhang^a, Rongxia Tang^a, Zhengan Yao^a, Jun Liang^{b,*}

^aDepartment of Mathematics, Sun Yat-sen University, China ^bSchool of Software, South China Normal University, China

Abstract

Initially used to rank web pages, PageRank has now been applied in many fields. With the growing scale of graph, accelerating PageRank computing is urged and designing parallel algorithm is a feasible solution. In this paper, two parallel PageRank algorithms IFP1 and IFP2 are proposed via improving the state-of-the-art Personalized PageRank algorithm, i.e., Forward Push. Theoretical analysis indicates that, IFP1 can take advantage of the DAG structure of the graph, where the dangling vertices improves the convergence rate and the unreferenced vertices decreases the computation amount. As an improvement of IFP1, IFP2 pushes mass to the dangling vertices only once but rather many times, and thus decreases the computation amount further. Experiments on six data sets illustrate that both IFP1 and IFP2 outperform Power method, where IFP2 with 38 parallelism can be at most 50 times as fast as the Power method.

Keywords: PageRank, Forward Push, Parallel

1. Introduction

S.Brin and L.Page[1, 2] proposed PageRank while dealing with the problem of ranking web pages retrieved by Google. PageRank measures the importance of web pages according to the network structure rather than contents. Generally, one web page has higher PageRank value if it is linked by more web pages, or the web pages with a link to it are with higher PageRank value themselves. Nowadays, PageRank's application goes far more beyond

^{*}Corresponding author

Internet[3]. We can find it in many fields such as social network analysis, chemistry, molecular biology, sports[4] and social sciences[5].

In the past decades, plenty of PageRank algorithms were proposed and among which, Power method is the most basic one. Some improvements based on Power method were presented. Kamvar[6] proposed the adaptive method which excluded the vertices had converged from the computation immediately. Haveliwala[7] and Kamvar[8] proposed the extrapolation method which focused on the second largest eigenvalue and made the best use of the previous iteration result. Kamvar[9] proposed the block method by utilizing the network's block structure. Gang[10] proposed the POWER-ARNOLDI method via introducing Arnoldi-type algorithm into PageRank computation.

Monte Carlo(MC) method [11] is another important PageRank algorithm. MC method simulates random walks on graph and approximates PageRank value by the probability that random walks terminate at each vertex. Some improvements based on MC method for undirected graph were proposed. Sarma[12] accelerated the walk by stitching short paths into a long one. Luo[13, 14] proposed Radar Push and thus obtained a result with lower variance.

In addition, distributed PageRank algorithms such as Sankaralingam[15], Zhu[16], Stergiou[17], Ishii[18] and [19, 20, 21, 22, 23] were proposed. [24, 25, 26] took advantage of the DAG structure by linear algebra. [27, 28] tried computing PageRank on GPU.

Algorithms mentioned above have respective advantages. With the explosive growth in the scale of graph, accelerating PageRank computing is urged. Parallelizing existing algorithms such as Power method[29] is a feasible solution, however, there stand barriers as below.

- (1) Algorithms based on the Power method converge slowly and can be partially parallel only, since the dependency between different iterations can not be eliminated.
- (2) Algorithms based on the MC method perform poorly on real application, since large memory space and bandwidth are required.

Forward Push, which is the state-of-the-art Personalized PageRank(PPR) algorithm, has been attracting more attention recently. With initial mass distribution \boldsymbol{p} , each vertex reserves 1-c proportion of the mass receiving from its source vertices, then evenly pushes the remaining c proportion to its target vertices. The mass each vertex obtains in the ending is just the PPR value. Since each vertex executes reserving and pushing operations independently,

Forward Push is easy to parallelizing. Designing parallel PageRank algorithm via improving Forward Push is an natural idea. We can obtain the correct PageRank vector on strongly connected graph, however, problems arise when there exist dangling vertices in the graph. Specifically, based on the random walk model, PageRank requires random walk arriving at dangling vertex randomly jump to any vertex, while Forward Push terminates this walk. The slight difference leads that, utilizing Forward Push for PageRank computation forcibly and counting PPR vector as PageRank vector simply, we can either terminate pushing operation at dangling vertices and thus obtain wrong result, or continue pushing mass at dangling vertices to every vertex and thus generate plenty of computation.

In this paper, we propose parallel PageRank algorithms via improving Forward Push which addressed the problems mentioned above. The contributions are as follows.

- (1) We reveal that PageRank vector is essentially the probability distribution of mass and can be obtained by Forward Push which terminates pushing mass at dangling vertices.
- (2) We propose two parallel PageRank algorithms, IFP1 and IFP2, via improving Forward Push. Compared with the Power method, both IFP1 and IFP2 can take advantage of DAG structure of the graph, has higher convergence rate and generates less computation amount.
- (3) Experimental results on six data sets demonstrate that both IFP1 and IFP2 outperform the Power method, where IFP2 with 38 parallelism can be at most 50 times as fast as.

The remaining of this paper are as follows. In section 2, we introduce PageRank and Forward Push. In section 3, we firstly give the solution of computing PageRank via Forward Push which terminates pushing mass at dangling vertices, then present IFP1 and corresponding theoretical analysis, and propose IFP2 at last. Some numerical experiments are performed in section 4. We summarize this paper in section 5.

2. Preliminary

In this section, PageRank and Forward Push will be briefly introduced. Then the problem arose by dangling vertices when computing PageRank by Forward Push will be detailed.

2.1. PageRank

Given graph G(V, E), where $V = \{v_1, v_2, \dots, v_n\}$, $E = \{(v_i, v_j) : i, j = 1, 2, \dots, n\}$ and |E| = m. Denote by $\mathbf{A} = (a_{ij})_{n \times n}$ the adjacency matrix, where a_{ij} is 1 if $(v_j, v_i) \in E$ and 0 else. Denote by $\mathbf{P} = (p_{ij})_{n \times n}$ the probability transition matrix, where

$$p_{ij} = \begin{cases} a_{ij} / \sum_{i=1}^{n} a_{ij}, & \text{if } \sum_{i=1}^{n} a_{ij} \neq 0, \\ 0, & \text{else.} \end{cases}$$

We call vertex without in-link the unreferenced vertex, vertex without outlink the dangling vertex. Delete unreferenced (dangling) vertex, the newly generating unreferenced (dangling) vertex is called the weak unreferenced (dangling) vertex. Let $\mathbf{d} = (d_1, d_2, ..., d_n)^T$, where d_i is 1 if v_i is dangling vertex and 0 else. Let $\mathbf{p} = (p_1, p_2, \cdots, p_n)^T$ denote the n-dimensional probability distribution. Let $\mathbf{P}' = \mathbf{P} + \mathbf{p}\mathbf{d}^T$ and $\mathbf{P}'' = c\mathbf{P}' + (1-c)\mathbf{p}\mathbf{e}^T$, where $c \in (0,1)$ is damping factor and $\mathbf{e} = (1,1,...,1)^T$. Denote by π the PageRank vector. According to [30], let $\mathbf{p} = \frac{e}{n}$, one of PageRank's definitions is

$$\pi = \mathbf{P}'' \pi, \pi_i > 0, \sum_{i=1}^n \pi_i = 1.$$
 (1)

PageRank can be interpreted from many perspectives. Based on random walk model, a random walk starts at random vertex, with probability c walks according to the graph, and with probability 1-c terminates, then π is the probability distribution of random walk terminating at each vertex. When $p \neq \frac{e}{n}$, PageRank converts to Personalized PageRank(PPR), and p is called the personalized vector. PageRank is a special case of PPR.

It should be noted that, the graph which rules the random walk is actually the graph corresponding to P' but rather the original one. The difference between these two is mainly on the dangling vertices, specifically, the former artificially links each dangling vertex to every vertices while the latter does nothing. As a consequence, when a random walk arriving at the dangling vertices, the former requires with probability c randomly choosing a target vertex and continuing the random walk, while the latter means terminating. Graph corresponding to P' contains more extra edges. From the perspective of computing, the existing of dangling vertices always generates more computation when walking according to P', and that is why directly utilizing Forward Push in computing PageRank is costly.

2.2. Forward Push

Forward Push is the state-of-the-art PPR algorithm. As described in Algorithm 1, with initial mass distribution p, each vertex does the following two: (1)reserve 1-c proportion of mass it has received; (2)evenly push the remaining c proportion to its target vertices. While Forward Push running, the pushing mass, i.e., the mass needing to push to the target vertices decreases and the reserved mass increases. The algorithm finishes when there's no vertex holds more than the pre-defined ξ pushing mass. The reserved mass of each vertex is just its PPR value.

Algorithm 1 Forward Push

```
Input:
 1: G(V, E): The graph;
2: c:The damping factor;
3: p:The personalized vector;
4: \xi:The tolerance of error.
Output:

 π:Personalized PageRank vector.

7: Each vertex v_i maintains a data structure \langle \overline{\pi}_i, h_i \rangle.
8: Initially set \overline{\pi}_i = 0, h_i = p_i.
9:
10: while There exists vertex v_i satisfying h_i > \xi do
11.
         \overline{\pi}_i + = (1-c)h_i;
         for v_j \in D(v_i) do
12:
                                                                                         \triangleright [D(v_i)] are target vertices of v_i.
             h_j + = \frac{ch_i}{deg(v_i)};
13:
14:
          end for
15:
         h_i = 0;
16: end while
17: \pi_i = \overline{\pi}_i.
```

Forward Push is convenient to parallelizing since each vertex executes reserving and pushing operation independently. Previous works seldom detailed how to address mass on the dangling vertices. As a special case of PPR, PageRank can be obtained by Forward Push when the graph is strongly connected. However, for graph containing dangling vertices, Forward Push can either

- (1) terminate pushing the mass on dangling vertices, and thus obtain an incorrect result;
- (2) continue pushing mass on dangling vertices evenly to every vertex, and thus generate plenty of computation.

Moreover, in real computing environment, if the scale of graph is large enough, the mass each vertex getting from the dangling vertices will be so small that the result may loss precision. Graphs abstracted from reality always contain large proportion of dangling vertices. To design parallel PageRank algorithm based on Forward Push, the problem arose by dangling vertices needs to be addressed firstly.

3. Algorithm and convergence analysis

In this section, we firstly demonstrate that PageRank vector is essentially a distribution of mass, and it is feasible to compute PageRank via Forward Push that needs not push mass on dangling vertices. Then we propose the parallel PageRank algorithm IFP1 by improving Forward Push and present theoretical analysis as well. At last, we propose IFP2, the improvement of IFP1.

3.1. Computing PageRank vector via Forward Push

By expanding Formula (1), we have

$$(\mathbf{I} - c\mathbf{P}')\boldsymbol{\pi} = (1 - c)\boldsymbol{p}. \tag{2}$$

Since $\rho(c\mathbf{P}') < 1$, $(\mathbf{I} - c\mathbf{P}')^{-1} = \sum_{r=0}^{\infty} (c\mathbf{P}')^r$, it follows that

$$\boldsymbol{\pi} = (1 - c) \sum_{r=0}^{\infty} (c \boldsymbol{P}')^r \boldsymbol{p}. \tag{3}$$

Formula (3) is just the algebraic form of Forward Push, the masses on dangling vertices are pushed to every vertex according to p.

By expanding (2), we have

$$(\boldsymbol{I} - c\boldsymbol{P})\boldsymbol{\pi} = (c\boldsymbol{d^T}\boldsymbol{\pi} + 1 - c)\boldsymbol{p}.$$

Since $\mathbf{I} - c\mathbf{P}$ is invertible, it follows that

$$\boldsymbol{\pi} = (c\boldsymbol{d^T}\boldsymbol{\pi} + 1 - c)(\boldsymbol{I} - c\boldsymbol{P})^{-1}\boldsymbol{p}.$$

Let $\gamma = c\mathbf{d}^T\boldsymbol{\pi} + 1 - c$, it follows that

$$\boldsymbol{\pi} = \gamma (\boldsymbol{I} - c\boldsymbol{P})^{-1} \boldsymbol{p}.$$

Since $1 = e^T \boldsymbol{\pi} = \gamma e^T (\boldsymbol{I} - c\boldsymbol{P})^{-1} \boldsymbol{p}$, we have

$$\gamma = \frac{1}{e^T (I - cP)^{-1} p}.$$

Since $\pi_i = \gamma e_i^T (I - cP)^{-1} p$, where e_i is *n*-dimensional vector with the i_{th} element is 1 and 0 others, it follows that

$$\pi_i = \frac{e_i^T (I - cP)^{-1} p}{e^T (I - cP)^{-1} p}.$$

Since $(\boldsymbol{I} - c\boldsymbol{P})^{-1} = \sum_{r=0}^{\infty} (c\boldsymbol{P})^r$, we have

$$\pi_i = \frac{\boldsymbol{e_i^T} \sum_{r=0}^{\infty} (c\boldsymbol{P})^r \boldsymbol{p}}{\boldsymbol{e^T} \sum_{r=0}^{\infty} (c\boldsymbol{P})^r \boldsymbol{p}}.$$
 (4)

 $\sum_{r=0}^{\infty} (c\mathbf{P})^r \mathbf{p}$ is the algebraic form of Forward Push as well, it is different from Formula (3) on two aspects: (1)it terminates pushing mass on the dangling vertices; (2) it reserves 100% but not 1-c proportion of the mass.

Formula (4) demonstrates that PageRank vector is essentially a distribution of reserving mass. The proportion each vertex reserves has no effect on the final result, however, if each vertex reserves 100% proportion of mass, there's no need for dangling vertices executing reserving operation. Moreover, it is \boldsymbol{P} but rather \boldsymbol{P}' that ruled the process of Forward Push, the mass on dangling vertices need not to be pushed any more. Formula (4) implies a solution of addressing the problem arose by dangling vertices.

3.2. IFP1

The remaining issue is parallelizing. Restricted by the computing resource, assigning exclusive thread for every vertex is infeasible. We can generate some threads and assign vertices to them. Then IFP1 is proposed as Algorithm 2.

While IFP1 running, thread j circularly scans S_j , any vertex v_i satisfying $h_i > \xi$ will be processed. The mass received from its source vertices are reserved by 100% proportion, and then pushed to its target vertices by c proportion. On the whole, the reserved mass increases and pushing mass decreases. The reserved mass of unreferenced vertices and weak unreferenced vertices stay steadily after several iterations, i.e., these vertices get converged.

Algorithm 2 IFP1

```
Input:
1: K:The number of threads;
2: \xi:The lower bound of mass.
Output:
3: \pi:PageRank vector.
4: Each vertex v_i maintains a data structure \langle \overline{\pi}_i, h_i \rangle.
5: Assign non-dangling vertices to K threads, denote by S_i the set of vertices belonging to thread j.
6: Initially set \overline{\pi}_i = 0, h_i = 1.
7: Invoke K Calculate \pi following \pi_i = \frac{\overline{\pi}_i + n_i}{\sum\limits_{i=1}^n (\overline{\pi}_i + h_i)}
7: Invoke K Calculations and Management; > [The K Calculations and Management do in parallel.]
                                        \frac{\overline{\pi}_i + \overline{h}_i}{n} while the Management terminates.
10: function CALCULATION(j)
11:
         while 1 do
12:
             for v_i \in S_i do
13:
                 if h_i > \xi then
14:
                      \overline{\pi}_i = \overline{\pi}_i + h_i;
15:
                      for u \in D(v_i) do
                                                                            \triangleright [D(v_i)] is the set of target vertices of v_i.
                         h_u = h_u + \frac{ch_i}{deg(v_i)};
16:
17:
                      end for
18:
                      h_i = 0;
19:
                  end if
20:
              end for
21:
          end while
22: end function
23:
24: function Management
25:
         while There exists non-dangling vertex satisfying h_i > \xi do
26:
          end while
27:
          Terminate all the K Calculations.
28: end function
```

If there exists no non-dangling vertex holds more than the predefined ξ pushing mass, IFP1 gets converged. IFP1 is similar to Forward Push except the following:

- (1) IFP1 is parallel while Forward Push is serial;
- (2) IFP1 processes non-dangling vertices only while Forward Push addresses all of them;
- (3) IFP1 reverses 100% proportion of the mass while Forward Push reserves 1-c proportion.

It should be noted that, in multi-thread environment, the addition on h_i must be atomic and thus data race may occur.

3.3. Algorithm analysis

We analyse IFP1 from three aspects, the convergence rate, error and computation amount.

3.3.1. Convergence rate

Convergence rate relates to iteration rounds, however, as a parallel algorithm IFP1 in fact has no iteration. We define one iteration of IFP1 as that the threads finish scanning through the whole vertices. Since whether processes the dangling vertices, the proportion of reserving mass and the parallelism have no effect on the convergence rate, for convenience, we assume that (1)IFP1 addresses all of the vertices; (2)the proportion of reserving mass is still 1-c; (3)the parallelism is 1.

Denote by $\overline{\pi}^{I}(t)$ the mass having been reserved at the beginning of the t_{th} iteration, by $\overline{\pi}^{R}(t)$ the mass needing to push at the beginning of the t_{th} iteration, then it follows that

$$||\overline{\boldsymbol{\pi}}^{I}(t) + \overline{\boldsymbol{\pi}}^{R}(t)||_{1} \leq n,$$

where $\overline{\pi}^I(0) = \mathbf{0}$, $\overline{\pi}^R(0) = \mathbf{e}$, $\lim_{t \to \infty} \frac{\overline{\pi}^I(t)}{||\overline{\pi}^I(t)||_1} = \mathbf{\pi}$, $\lim_{t \to \infty} \overline{\pi}^R(t) = \mathbf{0}$, $||\overline{\pi}^I(t)||_1$ increases monotonously on t, and $||\overline{\pi}^R(t)||_1$ decreases monotonously on t. The convergence rate can be measured by $\frac{||\overline{\pi}(t)||_1}{||\overline{\pi}(t+1)||_1}$. At the beginning of the t_{th} iteration, some vertices having been converged

At the beginning of the t_{th} iteration, some vertices having been converged and denote them by $V_U(t)$. Denote by V_D the set of dangling vertices, it follows that

$$||\overline{\pi}^{R}(t)||_{1} = \sum_{v_{i} \in V_{II}(t)} \overline{\pi}_{i}^{R}(t) + \sum_{v_{i} \in V_{1}(t)} \overline{\pi}_{i}^{R}(t) + \sum_{v_{i} \in V_{2}(t)} \overline{\pi}_{i}^{R}(t),$$

where
$$V_1(t) = V_D - V_U(t)$$
 and $V_2(t) = V - V_U(t) - V_D$.

During the t_{th} iteration, the reserved mass of $V_U(t)$ stay constant; the mass needing to push of $V_1(t)$ decrease to 0; the sum of mass needing to push of $V_2(t)$ decrease by 1-c proportion, thus it follows that

$$||\overline{\boldsymbol{\pi}}^{R}(t+1)||_{1} = \sum_{v_i \in V_{II}(t)} \overline{\pi}_i^{R}(t) + c \sum_{v_i \in V_2(t)} \overline{\pi}_i^{R}(t).$$

Vertices belonging to $V_U(t)$ have been converged, we have

$$\sum_{v_i \in V_U(t)} \overline{\pi}_i^R(t) < |V_U(t)|\xi.$$

Assuming that ξ is sufficiently small that $|V_U(t)|\xi \to 0$, then

$$\frac{||\overline{\pi}^R(t+1)||_1}{||\overline{\pi}^R(t)||_1} = \frac{c\sum_{v_i \in V_2(t)} \overline{\pi}^R_i(t)}{\sum_{v_i \in V_1(t)} \overline{\pi}^R_i(t) + \sum_{v_i \in V_2(t)} \overline{\pi}^R_i(t)}.$$

Let
$$\alpha(t) = \frac{\sum_{v_i \in V_2(t)} \overline{\pi}_i^R(t)}{\sum_{v_i \in V_1(t)} \overline{\pi}_i^R(t) + \sum_{v_i \in V_2(t)} \overline{\pi}_i^R(t)}$$
, then

$$\frac{||\overline{\pi}^R(t+1)||_1}{||\overline{\pi}^R(t)||_1} = c\alpha(t).$$

Since $V_1(t) \cup V_2 = V - V_U(t)$, we have

$$\alpha(t) = \frac{\sum_{v_i \in V - V_U(t) - V_D} \overline{\pi}_i^R(t)}{\sum_{v_i \in V - V_U(t)} \overline{\pi}_i^R(t)}.$$

 $\alpha(t)$ is the proportion of pushing mass on non-dangling vertices among the total pushing mass at the beginning of the t_{th} iteration.

Generally, $\alpha(t)$ negatively correlates with the proportion of dangling vertices. While IFP1 running, the amount of non-converged vertices decreases, then the proportion of dangling vertices among non-converged vertices increases, and thus $\alpha(t)$ decreases. Let $\alpha = \max_{t \geq 1} {\{\alpha(t)\}}$ and $\lambda = \alpha c$, then

$$\min_{t \ge 1} \left\{ \frac{||\overline{\pi}^R(t)||_1}{||\overline{\pi}^R(t+1)||_1} \right\} = \frac{1}{\max_{t > 1} \{\alpha(t)\}c} = \frac{1}{\alpha c} = \lambda^{-1}.$$

 λ^{-1} can be viewed as IFP1's convergence rate. Generally, the larger proportion of dangling vertices a graph has, the higher convergence rate IFP1 is. The iterations IFP1 takes to get converged under the predefined threshold ξ can be estimated by

$$||\overline{\pi}^{R}(T)||_{1} = n \prod_{t=0}^{T-1} c\alpha(t) = \sum_{v_{i} \in V_{1}(t)} \overline{\pi}_{i}^{R}(T) + \sum_{v_{i} \in V_{2}(t)} \overline{\pi}_{i}^{R}(T).$$

Since IFP1 gets converged when $||\boldsymbol{\pi}_R(t)||_{\infty} < \xi$ and mass on $V_1(t)$ decrease to 0 during the t_{th} iteration, one of the sufficient conditions is

$$n \prod_{t=0}^{T-1} c\alpha(t) \le n(\lambda)^T < |V - V_U(t) - V_D|\xi,$$

and thus

$$T > \log_{\lambda} \xi + \log_{\lambda} \frac{|V - V_U(T) - V_D|}{n}.$$

Generally, $|V - V_U(T) - V_D| = O(n)$, we have

$$T = O(\log_{\lambda} \xi). \tag{5}$$

3.3.2. Error

It is infeasible to estimate the error of each vertex since the complication of graph structure. We discuss the effect of pre-defined threshold ξ on relative error $ERR(\xi)$ from the whole. Assuming $||\overline{\pi}^R(t)||_1 = n\lambda^t$, then

$$||\overline{\boldsymbol{\pi}}^{I}(T+t) - \overline{\boldsymbol{\pi}}^{I}(T)||_{1} \le n|\lambda^{T} - \lambda^{T+t}| = (1-\lambda^{t})n\lambda^{T}.$$

Let $t \to \infty$, then we can obtain that

$$ERR(\xi) = \lim_{t \to \infty} \frac{||\boldsymbol{\pi}^{I}(T+t) - \boldsymbol{\pi}^{I}(T)||_{1}}{||\boldsymbol{\pi}^{I}(T+t)||_{1}} < \lim_{t \to \infty} \frac{|\lambda^{T} - \lambda^{T+t}|}{1 - \lambda^{T+t}} = \xi.$$
 (6)

Formula (6) can be viewed as an estimation of relative error under the predefined threshold ξ . It is to be noted that the error analysis mentioned above is rough, it considers no any factor of graph structure.

3.3.3. Computation amount

The computation of IFP1 mainly consists by additions and productions. Denote by m(t) the computation amount of the t_{th} iteration, and by M the total computation amount for IFP1 getting converged. Since only vertices belonging to $V-V_U(t)$ generate arithmetic operations during the t_{th} iteration, it follows that

$$m(t) = \sum_{v_i \in V - V_U(t)} (deg(v_i) + 1).$$

Let $\beta(t) = \frac{m(t+1)}{m(t)}$. $\beta(t)$ indicates the decreasing rate of computation amount. Unreferenced vertices and weak unreferenced vertices gradually exit computing while IFP1 running, thus $\beta(t) \leq 1$ and

$$m + n = m(0) \ge m(1) \ge m(2) \ge \dots \ge m(T).$$

Let $\beta = \max_{1 \le t \le T} \{\beta(t)\}$. Assuming that $\beta < 1$, i.e., there exist vertices getting converged at each iteration, we have

$$M = \sum_{t=0}^{T} m(t) = \sum_{t=0}^{T} \sum_{v_i \in V - V_U(t)} (deg(v_i) + 1) < (m+n) \frac{1-\beta^T}{1-\beta},$$

where $T = O(\log_{\lambda} \xi)$.

As a summary, it is clear that IFP1 could take advantage of DAG structure, where the dangling vertices improve the convergence rate and the unreferenced vertices lower the computation amount. Compared with Power method, on graph containing DAG structure, IFP1 requires less iterations to get converged and generates less computations at each iteration, thus is faster.

3.4. IFP2

IFP1 requires no the dangling vertices executing reserving operation, however, that pushing mass to dangling vertex from its source vertices is still needed and sometimes will be executed many times. Specifically, denote by v_d a dangling vertex and by $S(v_d)$ the set of v_d 's source vertices, each time v_i satisfies $h_i > \xi$, the pushing operation from v_i to v_d will be executed. On the other hand, we have

$$\sum_{t=0}^{T} \overline{\pi}_d^R(t) = \sum_{v_i \in S(v_d)} \frac{c}{\deg(v_i)} \sum_{t=0}^{T} \overline{\pi}_i^R(t),$$

where $\sum_{t=0}^{T} \overline{\pi}_{i}^{R}(t)$ is the reserved mass of v_{i} . That implies the mass on dangling vertices is completely determined by their source vertices. If we do not push mass to the the dangling vertices from their source vertices initially, but execute these operations after all the non-dangling vertices getting converged, then only once pushing operation is sufficient. Motivated by this, IFP2, an improvement of IFP1, is proposed as Algorithm 3.

Different from IFP1, IFP2 needs to address all of the vertices, both non-dangling vertices and dangling vertices are assigned to the K threads. IFP2 contains two phases, where phase1 addresses the non-dangling vertices and phase2 addresses the dangling vertices. Phase1 is similar to IFP1 except that no mass is pushed to the dangling vertices. While phase1 finished, all of the non-dangling vertices get converged, phase2 starts and pushes mass to the dangling vertices. Compared with IFP1, phase2 executes only once and thus some pushing operation corresponding to dangling vertices are saved. Specifically, during the t_{th} iteration, there are $\sum_{v \in V_D - V_U(t)} |S(v)| \text{ mass push-}$

Specifically, during the t_{th} iteration, there are $\sum_{v \in V_D - V_U(t)} |S(v)|$ mass pushing operations saved, thus the total saved operations is $\sum_{t=0}^{T} \sum_{v \in V_D - V_U(t)} |S(v)|$. Generally, the higher proportion of edges corresponding to dangling vertices the graph contains, the less computation IFP2 generates.

It should be noted that, IFP2 needs some preprocessing, before phase1 running, edges corresponding to dangling vertices should be invalidated. These work may increase the time consumption, however, based on appropriate data structure, the preprocessing cost is at most $o(\sum_{v \in V_D} |S(v)|)$. Experiments in the following will show that, the invaliding operations almost add no extra CPU time consumption.

Algorithm 3 IFP2

```
Input:
1: K:The number of threads;
2: \xi:The lower bound of mass.
Output:
3: \pi:PageRank vector.
4:
5: Preprocess the graph data and invalid edges between dangling vertices and their source vertices.
6: Each vertex v_i maintains a data structure \langle \overline{\pi}_i, h_i \rangle.
7: Assign vertices to K threads, denote by S_j^1 and S_j^2 the set of non-dangling vertices and dangling
    vertices belonging to thread j respectively.
8: Initially set \overline{\pi}_i = 0, h_i = 1.
9: Invoke K Calculations and Management; > [The K Calculations and Management do in parallel.]
10: Calculate \pi following \pi_i = \frac{\overline{\pi_i}}{\sum\limits_{i=1}^{n} \overline{\pi_i}} while the Management terminates.
12: function CALCULATION(j)
13:
         while 1 do
14:
             if CTRL then
                                                                                    \triangleright [CTRL is bool and initially true.]
15:
                 Phase1:
                 for v_i \in S_j^1 do
if h_i > \xi then
16:
17:
                         \overline{\pi}_i = \overline{\pi}_i + h_i;
for u \in D(v_i) do
18:
19:
                                                                           \triangleright [D(v_i)] is the set of target vertices of v_i.
                             h_u = h_u + \frac{ch_i}{deg(v_i)};
20:
21:
                          end for
22:
                          h_i = 0;
23:
                      end if
24:
                 end for
25:
             else
26:
                 Phase2:
27:
                 if STS then
                                                                                      \triangleright [STS is bool and initially true.]
                     for v_i \in S_j^2 do
\overline{\pi}_i = h_i;
for u \in S(v_i) do
28:
29:
30:
                                                                            \triangleright [S(v_i)] is the set of source vertices of v_i.]
                             \overline{\pi}_i = \overline{\pi}_i + \frac{c\overline{\pi}_u}{deg(u)};
31:
32:
                          end for
33:
                          h_i = 0;
34:
                      end for
                     STS = false;
35:
36:
                 end if
37:
              end if
38:
         end while
39: end function
40:
41: function Management
42:
         while There exists non-dangling vertex satisfying h_i > \xi do
43:
         end while
44:
         CTRL = false;
45:
         while There exists dangling vertex satisfying h_i > \xi do
46:
47:
         Terminate all the K Calculations.
48: end function
```

4. Experiment

In this section, both IFP1 and IFP2 are demonstrated experimentally. Firstly, we introduce the computing environment and indexes to be used. Then, the convergence of IFP1 and IFP2 are illustrated, and the comparison with Power method as well. At last, the results of IFP2 with different threads is elaborated.

4.1. Experiment Setting

All algorithms of this experiment are implemented with C++ on serves with Intel(R) Xeon(R) Silver 4210R CPU 2.40GHz 40 processors and 190GB memory. The operation system is Ubuntu 18.04.5 LTS. Six data sets are illustrated in table 1, where n, m, n_d, m_d and $deg = \frac{m}{n}$ represent the number of vertices, the number of edges, the number of dangling vertices, the number of edges corresponding to the dangling vertices and the average degree respectively. The CPU time consumption T and max relative error $ERR = \max_{v_i \in V} \frac{|\pi_i - \pi_i|}{\pi_i}$ are taken to estimate the algorithms, where The true PageRank value π_i is obtained by Power method at the 210_{th} iteration. The damping factor c = 0.85.

Data Sets	n	m	n_d	m_d	deg
web-Stanford	281903	2312497	172	410	8.21
Stanford-Berkeley	683446	7583376	68062	994368	11.1
web-Google	916428	5105039	176974	325725	5.57
in-2004	1382908	16917053	282306	1006484	12.23
soc-Live $Journal1$	4847571	68993773	539119	2092984	14.23
uk-2002	18520343	298113762	2760973	16029357	16.10

Table 1: Data Sets

4.2. Convergence

The relation of ERR, T and ξ on six data sets is illustrated in Figure 1.

(1) The blue lines show that ERR has a positive linear relation with ξ , which is consistent with Formula (6). The red lines show that T has a negative exponential relation with ξ , which is consistent with Formula (5). That red lines marked with triangles are lower than red lines marked with squares show that IFP2 is faster than IFP1 on all six data sets.

(2) The blue lines show that ERR scarcely changes when $\xi < 10^{-15}$. It seems both IFP1 and IFP2 have limitation in precision. We believe it is not a algorithm flaw, but caused by the insufficiency of C++'s DOUBLE data type, whose significant digit number is 15. $h_i < 10^{-15}$ can not change $\overline{\pi}_i$, thus ERR stays constantly.

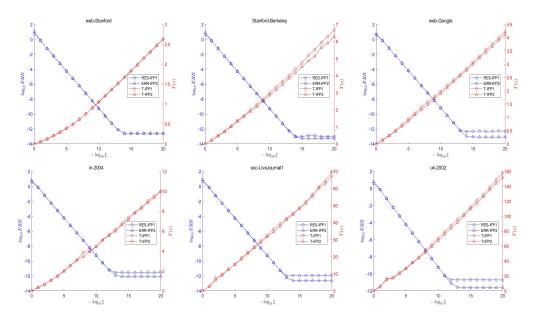


Figure 1: ξ Versus RES and T

4.3. Comparison with other algorithms

We compare IFP1 and IFP2 with Power method (SPI) [1, 2] and parallel Power method (MPI)[27]. Both MPI, IFP1 and IFP2 are executed with 38 parallelism. The relation of ERR and T is illustrated in Figures 2. The CPU time consumption of preprocessing is illustrated in Table 2. Table 3 illustrates the CPU time consumption T when ERR < 0.001.

- (1) The magenta, red and blue lines are lower than green lines in Figure 2, it shows that both MPI, IFP1 and IFP2 are faster than SPI, parallelizing is an effective solution to accelerate PageRank computing.
- (2) The magenta lines and red lines are lower than green lines in Figure 2, it shows that both IFP1 and IFP2 outperform MPI. Since IFP1, IFP2 and MPI are executed with the same parallelism, we believe the advantages owe to the higher convergence rate of IFP1 and IFP2.

- (3) For MPI and IFP1, the preprocessing is assigning vertices to threads, for IFP2, the preprocessing includes assigning and invalidating the edges corresponding to the dangling vertices. MPI assigns all of the vertices to threads, thus consumes more time. The time consumption of invalidating is so few that it can be done in 0.155s even for uk-2002.
- (4) Table 3 shows that IFP2 with 38 parallelism can be at most 50 times as fast as SPI, and at most 3 times faster than MPI, when ERR < 0.001.

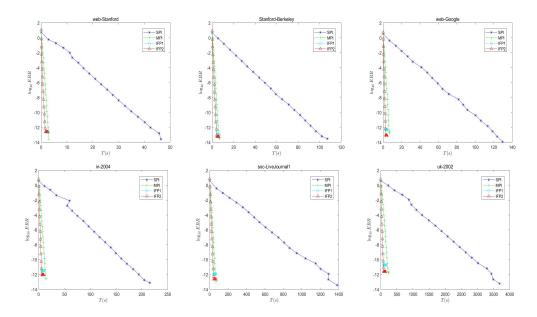


Figure 2: T Versus ERR

MPI	IFP1	IFP2
0.014	0.012	0.012
0.041	0.030	0.031
0.043	0.033	0.036
0.070	0.046	0.054
0.236	0.178	0.194
0.926	0.629	0.784
	0.014 0.041 0.043 0.070 0.236	0.014 0.012 0.041 0.030 0.043 0.033 0.070 0.046 0.236 0.178

Table 2: The time consumption of preprocessing

Data sets	SPI	MPI	IFP1	IFP2
web-Stanford	14.445	0.949	0.303	0.285
Stanford-Berkeley	27.525	1.709	1.038	0.995
web-Google	32.182	1.936	0.718	0.671
in-2004	64.665	4.728	1.497	1.481
soc-LiveJournal1	423.183	21.415	12.702	12.617
uk-2002	1108.711	85.455	24.823	22.479

Table 3: The time consumption when ERR < 0.001

4.4. Performance under different parallelism

We execute IFP2 with parallelism 4, 8, 16, 32 and 38. The relation of ERR and T with different parallelism are illustrated in Figure 3. With the increasing of parallelism, the CPU time consumption T decreases. Generally, the more the parallelism the faster the computing. IFP2 with 16 parallelism is faster than MPI with 38 parallelism, which proves IFP2 has higher convergence rate than Power method again.

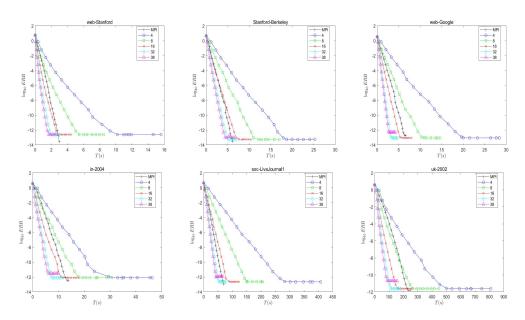


Figure 3: T Versus ERR under parallelism 4,8,16,32 and 38

5. Conclusion

PageRank is a basic problem of graph computation and parallelization is a feasible solution of accelerating computing. In this paper, we firstly reveal that PageRank vector is essentially the mass distribution, and base on which two parallel PageRank algorithms IFP1 and IFP2 are proposed. The most prominent feature of IFP1 is that it can make full use of the DAG structure, generally, the larger proportion of dangling vertices, the higher convergence rate, the larger proportion of unreferenced vertices and weak unreferenced vertices, the less computation amount. IFP2 pushes mass to the dangling vertices only once, thus, the more edges corresponding to the dangling vertices, the more computation decreases. Experiments on six data sets demonstrate that both IFP1 and IFP2 are faster than the Power method. However, every coin has two sides, on graph containing no DAG structure such as undirected graph, neither IFP1 nor IFP2 can outperform the Power method since the atomic operation is costly. In the future, PageRank on dynamic graph can be studied.

References

- [1] S. Brin, L. Page, Reprint of: The anatomy of a large-scale hypertextual web search engine, Computer networks 56 (18) (2012) 3825–3833.
- [2] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web., Tech. rep., Stanford InfoLab (1999).
- [3] D. F. Gleich, Pagerank beyond the web, Siam Review 57 (3) (2015) 321-363.
- [4] S. Brown, A pagerank model for player performance assessment in basketball, soccer and hockey, arXiv preprint arXiv:1704.00583 (2017).
- [5] Ying, Hou, Zhou, Dou, Wang, Shao, The changes of central cultural cities based on the analysis of the agglomeration of literati's footprints in tang and song dynasties, Journal of Geo-information Science 22 (5) (2020) 945–953.
- [6] S. Kamvar, T. Haveliwala, G. Golub, Adaptive methods for the computation of pagerank, Linear Algebra and its Applications 386 (2004) 51–65.

- [7] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, G. Golub, Computing pagerank using power extrapolation, Tech. rep., Stanford (2003).
- [8] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, G. H. Golub, Extrapolation methods for accelerating pagerank computations, in: Proceedings of the 12th international conference on World Wide Web, 2003, pp. 261–270.
- [9] S. Kamvar, T. Haveliwala, C. Manning, G. Golub, Exploiting the block structure of the web for computing pagerank, Tech. rep., Stanford (2003).
- [10] G. Wu, Y. Wei, A power-arnoldi algorithm for computing pagerank, Numerical Linear Algebra with Applications 14 (7) (2007) 521–546.
- [11] K. Avrachenkov, N. Litvak, D. Nemirovsky, N. Osipova, Monte carlo methods in pagerank computation: When one iteration is sufficient, SIAM Journal on Numerical Analysis 45 (2) (2007) 890–904.
- [12] A. D. Sarma, A. R. Molla, G. Pandurangan, E. Upfal, Fast distributed pagerank computation, in: International Conference on Distributed Computing and Networking, Springer, 2013, pp. 11–26.
- [13] S. Luo, Distributed pagerank computation: An improved theoretical study, Proceedings of the AAAI Conference on Artificial Intelligence 33 (2019) 4496–4503.
- [14] S. Luo, Improved communication cost in distributed pagerank computation—a theoretical study, in: International Conference on Machine Learning, PMLR, 2020, pp. 6459–6467.
- [15] K. Sankaralingam, S. Sethumadhavan, J. Browne, Distributed pagerank for p2p systems, in: High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on, 2003, pp. 58–68. doi:10.1109/HPDC.2003.1210016.
- [16] Y. Zhu, S. Ye, X. Li, Distributed pagerank computation based on iterative aggregation-disaggregation methods, in: Proceedings of the 14th ACM international conference on Information and knowledge management, 2005, pp. 578–585.

- [17] S. Stergiou, Scaling pagerank to 100 billion pages, in: Proceedings of The Web Conference 2020, 2020, pp. 2761–2767.
- [18] H. Ishii, R. Tempo, Distributed randomized algorithms for the pagerank computation, IEEE Transactions on Automatic Control 55 (9) (2010) 1987–2002.
- [19] H. Ishii, R. Tempo, E.-W. Bai, A web aggregation approach for distributed randomized pagerank algorithms, IEEE Transactions on automatic control 57 (11) (2012) 2703–2717.
- [20] A. Suzuki, H. Ishii, Efficient pagerank computation via distributed algorithms with web clustering, arXiv preprint arXiv:1907.09979 (2019).
- [21] T. Charalambous, C. N. Hadjicostis, M. G. Rabbat, M. Johansson, Totally asynchronous distributed estimation of eigenvector centrality in digraphs with application to the pagerank problem, in: 2016 IEEE 55th Conference on Decision and Control (CDC), IEEE, 2016, pp. 25–30.
- [22] A. Suzuki, H. Ishii, Distributed randomized algorithms for pagerank based on a novel interpretation, in: 2018 Annual American Control Conference (ACC), IEEE, 2018, pp. 472–477.
- [23] L. Dai, N. M. Freris, Fully distributed pagerank computation with exponential convergence, arXiv preprint arXiv:1705.09927 (2017).
- [24] I. C. F. Ipsen, T. M. Selee, Pagerank computation, with special attention to dangling nodes, SIAM Journal on Matrix Analysis and Applications 29 (4) (2008) 1281–1296. arXiv:https://doi.org/10.1137/060664331. URL https://doi.org/10.1137/060664331
- [25] Y. Lin, X. Shi, Y. Wei, On computing pagerank via lumping the google matrix, Journal of Computational and Applied Mathematics 224 (2) (2009) 702–708.
- [26] Z. Zhu, Q. Peng, Z. Li, X. Guan, O. Muhammad, Fast pagerank computation based on network decomposition and dag structure, IEEE Access 6 (2018) 41760–41770.

- [27] N. T. Duong, Q. A. P. Nguyen, A. T. Nguyen, H.-D. Nguyen, Parallel pagerank computation using gpus, in: Proceedings of the Third Symposium on Information and Communication Technology, 2012, pp. 223–230.
- [28] Lai, The computations of pagerank scores based on gpgpu environment, Ph.D. thesis, SUN YAT-SEN University (2017).
- [29] H. Migallón, V. Migallón, J. Penadés, Parallel two-stage algorithms for solving the pagerank problem, Advances in Engineering Software 125 (2018) 188–199.
- [30] Berkhin, Pavel, A survey on pagerank computing, Internet Mathematics 2 (1) (2005) 73–120.