arXiv:2302.03245v1 [cs.DS] 7 Feb 2023

通过改进前向推送实现的两种并行 PageRank 算法

Qi Zhang[a]、Rongxia Tang[a]、Zhengan Yao[a]、Jun Liang[b,*]

[a]数学系，中山大学，中国[b]软件学院，华南师范大学，中国

摘要

PageRank 最初用于对网页进行排名，如今已应用于许多领域。随着图规模不断扩大，加速 PageRank 计算的需求日益迫切，而设计并行算法是一种可行的解决方案。本文通过改进当前最先进的个性化 PageRank 算法——前向推送，提出了两种并行 PageRank 算法 IFP1 和 IFP2。理论分析表明，IFP1 能够利用图的有向无环图结构，其中悬挂顶点可以提高收敛速度，而未被引用的顶点能够减少计算量。作为 IFP1 的改进，IFP2 仅向悬挂顶点推送一次质量，而不是多次推送，从而进一步减少计算量。在六个数据集上的实验表明，IFP1 和 IFP2 均优于幂法；其中，具有 38 并行度的 IFP2 最多可达到幂法速度的 50 倍。

关键词：PageRank、前向推送、并行

1. 引言

S. Brin 和 L. Page[1, 2] 在处理 Google 检索网页的排名问题时提出了 PageRank。PageRank 根据网络结构而非内容来衡量网页的重要性。通常，如果一个网页被更多网页链接，或者链接到它的网页自身具有更高的 PageRank 值，那么它的 PageRank 值就更高。如今，PageRank 的应用早已超越

*通讯作者

互联网[3]。我们可以在许多领域中看到它的应用，例如社会网络分析、化学、分子生物学、体育[4] 和社会科学[5]。

过去几十年来，人们提出了大量 PageRank 算法，其中幂法是最基础的一种。基于幂法，人们提出了一些改进方法。Kamvar[6] 提出了自适应方法，可立即将已收敛的顶点排除在计算之外。Haveliwala[7] 和 Kamvar[8]提出了外推法，该方法着眼于第二大特征值，并充分利用前一次迭代的结果。Kamvar[9] 利用网络的块结构提出了分块法。Gang[10] 将 Arnoldi 型算法引入 PageRank 计算，提出了 POWER-ARNOLDI 方法。

蒙特卡洛（MC）方法 [11] 是另一种重要的 PageRank 算法。MC 方法模拟图上的随机游走，并根据随机游走终止于各顶点的概率来近似 PageRank 值。针对无向图，人们提出了一些基于 MC 方法的改进。Sarma[12] 通过将短路径拼接成一条长路径来加速游走。Luo[13, 14] 提出了 Radar Push，从而获得了方差更低的结果。

此外，还提出了分布式 PageRank 算法，例如 Sankaralingam[15],Zhu [16], Stergiou[17], Ishii[18] and [19, 20, 21, 22, 23] 。 [24,25, 26]通过线性代数利用了有向无环图结构。 [27, 28] 尝试在图形处理器上计算 PageRank。

上述算法各有优势。随着图规模爆炸式增长，加速 PageRank 计算已迫在眉睫。并行化现有算法（如幂法）[29]是一种可行的解决方案，但仍面临以下障碍。

（1）基于幂法的算法收敛缓慢，且只能实现部分并行，因为不同迭代之间的依赖关系无法消除。（2）基于 MC 方法的算法在实际应用中表现不佳，因为它们需要大量内存空间和带宽。

前向推送是当前最先进的个性化 PageRank（个性化 PageRank）算法，近年来受到了越来越多的关注。在初始质量分布为 $\boldsymbol{p}$ 的情况下，每个顶点会保留从其源顶点接收到的质量的 $1-c$ 比例，然后将剩余的 c 比例均匀推送给其目标顶点。最终每个顶点获得的质量正是其个性化 PageRank 值。由于每个顶点都独立执行保留和推送操作，

前向推送易于并行化。通过改进前向推送来设计并行 PageRank 算法是一个自然的想法。我们可以在强连通图上获得正确的 PageRank 向量；然而，当图中存在悬挂顶点时，问题就会出现。具体而言，根据随机游走模型，PageRank 要求到达悬挂顶点的随机游走随机跳转到任意顶点，而前向推送会终止这次游走。这一细微差异意味着，如果强行利用前向推送进行 PageRank 计算，并简单地将个性化PageRank向量视为 PageRank 向量，那么我们要么在悬挂顶点处终止推送操作，从而得到错误结果；要么将质量继续推送到每个顶点，从而产生大量计算。

本文通过改进前向推送，提出了并行 PageRank 算法，以解决上述问题。本文的贡献如下。

（1）我们揭示了 PageRank 向量本质上是质量的概率分布，可以通过在悬挂顶点处终止推送质量的前向推送来获得。（2）我们通过改进前向推送，提出了两种并行 PageRank 算法 IFP1 和 IFP2。与幂法相比，IFP1 和 IFP2 都能利用图的有向无环图结构，具有更高的收敛速度，并产生更少的计算量。（3）在六个数据集上的实验结果表明，IFP1 和 IFP2 的性能均优于幂法，其中具有 38 并行度的 IFP2 最多可实现 50 倍的加速。

本文其余部分安排如下。第 2 节介绍 PageRank 和前向推送。第 3 节首先给出通过在悬挂顶点处终止推送质量的前向推送来计算 PageRank 的方法，然后介绍 IFP1 及其相应的理论分析，最后提出 IFP2。第 4 节进行数值实验。第 5 节对本文进行总结。

2. 预备知识

本节将简要介绍 PageRank 和前向推送。随后将详细说明使用前向推送计算 PageRank 时由悬挂顶点引发的问题。

2.1. PageRank

给定图 $G(V, E)$，其中 $V = \{v_1, v_2, \cdots, v_n\}$, $E = \{(v_i, v_j) : i, j = 1, 2, \cdots, n\}$ 和 $|E| = m$。记 $\boldsymbol{A} = (a_{ij})_{n\times n}$ 为邻接矩阵，其中若 $(v_j, v_i) \in E$，则 a_{ij} 为 1，否则为 0。记 $\boldsymbol{P} = (p_{ij})_{n\times n}$ 为概率转移矩阵，其中

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

我们将没有入链的顶点称为未引用顶点，将没有出链的顶点称为悬挂顶点。删除未引用（悬挂）顶点后，新产生的未引用（悬挂）顶点称为弱未引用（悬挂）顶点。令 $\boldsymbol{d} = (d_1, d_2, ..., d_n)^T$，其中 d_i 在 v_i 是悬挂顶点时为 1，否则为 0。令 $\boldsymbol{p} = (p_1, p_2, \cdots, p_n)^T$ 表示 n 维概率分布。令 $\boldsymbol{P}' = \boldsymbol{P} + \boldsymbol{p}\boldsymbol{d}^T$ 和 $\boldsymbol{P}'' = c\boldsymbol{P}' + (1 - c)\boldsymbol{p}\boldsymbol{e}^T$，其中 $c \in (0, 1)$ 是阻尼因子，且 $\boldsymbol{e} = (1, 1, ..., 1)^T$。用 $\boldsymbol{\pi}$ 表示 PageRank 向量。根据[30], , 令 $\boldsymbol{p} = \frac{e}{n}$，PageRank 的一种定义为 –

$$\boldsymbol{\pi} = \boldsymbol{P}''\boldsymbol{\pi}, \pi_i > 0, \sum_{i=1}^{n} \pi_i = 1. \tag{1}$$

PageRank 可以从多个角度进行解释。基于随机游走模型，随机游走从一个随机顶点开始，以 c 的概率沿图进行游走，并以 $1 - c$ 的概率终止；此时，$\boldsymbol{\pi}$ 表示随机游走在各个顶点终止的概率分布。当 $\boldsymbol{p} \neq \frac{e}{n}$ 时，PageRank 转化为个性化 PageRank（PPR），而 $\boldsymbol{p}$ 被称为个性化向量。PageRank 是 PPR 的一个特例。

需要注意的是，控制随机游走的图实际上是与 $\boldsymbol{P}'$ 对应的图，而不是原始图。这两者的区别主要在于悬挂顶点，具体而言，前者人为地将每个悬挂顶点连接到所有顶点，而后者则不作处理。因此，当随机游走到达悬挂顶点时，前者要求以 c 的概率随机选择一个目标顶点并继续随机游走，而后者则意味着终止。与 $\boldsymbol{P}'$ 对应的图包含更多额外的边。从计算的角度来看，按照 $\boldsymbol{P}'$ 进行游走时，悬挂顶点总会带来更多计算，这就是直接利用前向推送计算 PageRank 成本高昂的原因。

2.2. 前向推送

前向推送是最先进的个性化 PageRank 算法。如算法 **1** 所述，在初始质量分布为 $\boldsymbol{p}$ 时，每个顶点执行以下两项操作：（**1**）保留其收到的质量中的 $1-c$ 比例；（2）将剩余的 c 比例平均推送给其目标顶点。在前向推送运行过程中，推送质量（即需要推送给目标顶点的质量）逐渐减少，而保留质量逐渐增加。当没有顶点持有超过预定义 ξ 的推送质量时，算法结束。每个顶点的保留质量就是其个性化 PageRank 值。

算法 1 前向推送

输入：
1: $G(V, E)$:图;
2: c: 阻尼因子;
3: $\boldsymbol{p}$: 个性化向量;
4: ξ: 误差容限。

输出：
5: $\boldsymbol{\pi}$: 个性化 PageRank 向量。
6:
7: 每个顶点 v_i 都维护一个数据结构 $\langle \pi_i, h_i \rangle$。
8: 初始设置 $\pi_i = 0$、 $h_i = p_i$。
9:
10: **当** 存在满足 $h_i > \xi$ 的顶点 v_i **执行**
11: $\pi_i += (1-c)h_i$;
12: **对于** $v_j \in D(v_i)$ **执行** ▷ [$D(v_i)$ 是 v_i 的目标顶点。]
13: $h_j += \frac{ch_i}{deg(v_i)}$;
14: **结束循环**
15: $h_i = 0$;
16: </style}结束循环
17: $\pi_i = \pi_i$。

前向推送便于并行化，因为每个顶点都能独立执行预留和推送操作。以往的研究很少详细说明如何处理悬挂顶点上的质量。作为个性化 PageRank 的一个特例，当图强连通时，可以通过前向推送得到 PageRank。然而，对于包含悬挂顶点的图，前向推送可能会

（**1**）停止推送悬挂顶点上的质量，从而得到错误结果；（2）继续将悬挂顶点上的质量均匀地推送到每个顶点，从而产生大量计算。

此外，在真实的计算环境中，如果图的规模足够大，每个顶点从悬挂顶点获得的质量会

小到可能导致结果精度损失。从现实世界抽象出的图总是包含很大比例的悬挂顶点。要设计基于前向推送的并行 PageRank 算法，首先需要解决悬挂顶点带来的问题。

3. 算法与收敛性分析

本节首先说明 PageRank 向量本质上是质量分布，并且可以通过前向推送计算 PageRank，而无需在悬挂顶点上推送质量。然后，我们通过改进前向推送提出并行 PageRank 算法 IFP1，并给出相应的理论分析。最后，我们提出 IFP1 的改进版本 IFP2。

3.1. 通过前向推送计算 *PageRank* 向量

展开公式 (1)，可得

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

由于 $\rho(c\boldsymbol{P}^{'}) < 1$，$(\boldsymbol{I} - c\boldsymbol{P}^{'})^{-1} = \sum_{r=0}^{\infty}(c\boldsymbol{P}^{'})^{r}$，因此

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

公式 (3) 正是前向推送的代数形式，悬挂顶点上的质量会根据 $\boldsymbol{p}$推送到每个顶点。

展开式 (2)，可得

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

由于 $\boldsymbol{I} - c\boldsymbol{P}$ 可逆，因此可得

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

令 $\gamma = c\boldsymbol{d}^{\boldsymbol{T}}\boldsymbol{\pi} + 1 - c$，可得

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

由于 $1 = \boldsymbol{e}^{\boldsymbol{T}}\boldsymbol{\pi} = \gamma\boldsymbol{e}^{\boldsymbol{T}}(\boldsymbol{I} - c\boldsymbol{P})^{-1}\boldsymbol{p}$，我们有

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

由于 $\pi_i = \gamma \boldsymbol{e}_{\boldsymbol{i}}^{\boldsymbol{T}}(\boldsymbol{I} - c\boldsymbol{P})^{-1}\boldsymbol{p}$，其中 $\boldsymbol{e}_i$ 是 n 维向量，第 i_{th} 个元素为 1，其余元素均为 0，因此

$$\pi_i = \frac{\boldsymbol{e}_{\boldsymbol{i}}^{\boldsymbol{T}}(\boldsymbol{I}-c\boldsymbol{P})^{-1}\boldsymbol{p}}{\boldsymbol{e}^{\boldsymbol{T}}(\boldsymbol{I}-c\boldsymbol{P})^{-1}\boldsymbol{p}}.$$

由于 $(\boldsymbol{I} - c\boldsymbol{P})^{-1} = \sum_{r=0}^{\infty}(c\boldsymbol{P})^r$，我们有

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

$\sum_{r=0}^{\infty}(c\boldsymbol{P})^r\boldsymbol{p}$ 同样是前向推送的代数形式，但它与公式（3）在两个方面有所不同：（1）它会在悬挂顶点处终止推送质量；（2）它保留 100% 的质量，而非 $1-c$ 比例的质量。

公式（4）表明，PageRank 向量本质上是保留质量的一种分配。每个顶点保留的比例不会影响最终结果；然而，如果每个顶点都保留 100% 的质量，悬挂顶点便无需执行预留操作。此外，决定前向推送过程的是 $\boldsymbol{P}$ 而非 $\boldsymbol{P}'$，悬挂顶点上的质量无需再推送。公式（4）为解决悬挂顶点引发的问题提供了一种方案。

3.2. IFP1

剩下的问题是如何并行化。受计算资源限制，为每个顶点分配一个专属线程并不可行。我们可以生成若干线程，并将顶点分配给这些线程。由此提出 IFP1，如算法 2 所示。

IFP1 运行期间，线程 j 循环扫描 S_j，任何满足 $h_i > \xi$ 的顶点 v_i 都会被处理。从其源顶点接收的质量会按 100% 的比例予以保留，然后按 c 的比例推送到其目标顶点。总体而言，保留质量增加，推送质量减少。经过几次迭代后，未被引用的顶点和弱无引用顶点的保留质量保持稳定，即这些顶点已收敛。

算法 2IFP1

输入： 1

： K**:线程数；** **2:** ξ**:质量下界。输出：** **3:** π**:PageRank 向量。** **4: 每个顶点** v_i **维护一个数据结构** $\langle \pi_i, h_i \rangle$。**5: 将非悬挂顶点分配给** K **个线程，用** S_j **表示属于线程** j **的顶点集合。** **6: 初始设置** $\pi_i = 0$、$h_i = 1$。**7: 调用** K **计算与管理；▷ [计算与管理** K **并行执行。]8: 在管理过程终止期间，按如下方式计算** π **：** $\pi_i = {}_{n}{}^{\pi_i + h_i}{\sum (\pi_i + h_i)_{i=1}}$**9:10: 函数** CALCULATION**(**j**) 11: 当 1 时执行 12: 对每个** $v_i \in S_j$ **执行 13: 如果** $h_i > \xi$ **，则 14:** $\pi_i = \pi_i + h_i$**；15: 对每个** $u \in D(v_i)$**执行▷ [**$D(v_i)$ **是** v_i**的目标顶点集合。]16:** $h_u = h_u + {}_{deg}{}^{ch_i}(v_i)$**；17: 结束循环 18:** $h_i = 0$**；19: 结束条件 20: 结束循环 21: 结束循环 22: 结束函数 23:24: 函数** MANAGEMENT**25: 当存在满足**$h_i > \xi$ **的非悬挂顶点时 26: 结束循环 27: 终止所有** K **计算。28: 结束函数**

如果不存在持有超过预定义 ξ 推送质量的非悬挂顶点，则 IFP1 收敛。除了以下几点，IFP1 与前向推送相似：

(1) IFP1 是并行的，而前向推送是串行的；(2) IFP1 仅处理非悬挂顶点，而前向推送处理所有顶点；(3) IFP1 反转 100% 的质量，而前向推送保留$1 - c$ 比例。

应注意，在多线程环境中，对 h_i的加法必须是原子操作，因此可能发生数据竞争。

3.3. 算法分析

我们从收敛速度、误差和计算量三个方面分析 IFP1。

3.3.1. 收敛速度

收敛速度与迭代轮次有关，但作为一种并行算法，IFP1 实际上没有迭代。我们将 IFP1 的一次迭代定义为线程完成对所有顶点的扫描。由于是否处理悬挂顶点、保留质量的比例以及并行度都不会影响收敛速度，为方便起见，我们假设：(1) IFP1 处理所有顶点；(2)保留质量的比例仍为 $1-c$；(3)并行度为 1。

用 $\boldsymbol{\pi}^I(t)$ 表示第 t_{th} 次迭代开始时已保留的质量，用 $\boldsymbol{\pi}^R(t)$ 表示第 t_{th} 次迭代开始时需要推送的质量，则可得

$$||\overline{\boldsymbol{\pi}}^I(t) + \overline{\boldsymbol{\pi}}^R(t)||_1 \leq n,$$

其中 $\boldsymbol{\pi}^I(0) = 0$, $\boldsymbol{\pi}^R(0) = e$, $\lim_{t\to\infty} \frac{\boldsymbol{\pi}^I(t)}{||\boldsymbol{\pi}^I(t)||_1} = \boldsymbol{\pi}$, $\lim_{t\to\infty} \boldsymbol{\pi}^R(t) = 0$, $||\boldsymbol{\pi}^I(t)||_1$ 在 t 上单调递增，而 $||\boldsymbol{\pi}^R(t)||_1$ 在 t 上单调递减。收敛速度可由 $\frac{||\boldsymbol{\pi}^t||_1}{||\boldsymbol{\pi}^{t+1}||_1}$ 衡量。

在第 t_{th} 次迭代开始时，某些顶点已经收敛，将它们记为 $V_U(t)$。用 V_D 表示悬挂顶点的集合，则可得

$$||\overline{\boldsymbol{\pi}}^R(t)||_1 = \sum_{v_i\in V_U(t)} \overline{\pi}_i^R(t) + \sum_{v_i\in V_1(t)} \overline{\pi}_i^R(t) + \sum_{v_i\in V_2(t)} \overline{\pi}_i^R(t),$$

其中 $V_1(t) = V_D - V_U(t)$ 且 $V_2(t) = V - V_U(t) - V_D$。

在 t_{th} 迭代期间，$V_U(t)$ 的保留质量保持不变；$V_1(t)$ 中需要推送的质量降至 0；$V_2(t)$ 中需要推送的质量总和减少了 $1-c$ 比例，因此可得

$$||\overline{\boldsymbol{\pi}}^R(t+1)||_1 = \sum_{v_i\in V_U(t)} \overline{\pi}_i^R(t) + c \sum_{v_i\in V_2(t)} \overline{\pi}_i^R(t).$$

属于 $V_U(t)$ 的顶点已收敛，我们有

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

假设 ξ 足够小，使得 $|V_U(t)|\xi \to 0$，那么

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

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

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

由于 $V_1(t) \cup V_2 = V - V_U(t)$，我们有

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

$\alpha(t)$表示在 t_{th} 次迭代开始时，非悬挂顶点上的推送质量占总推送质量的比例。

一般而言， $\alpha(t)$与悬挂顶点的比例呈负相关。IFP1运行期间，未收敛顶点的数量会减少，随后未收敛顶点中悬挂顶点的比例会增加，因此 $\alpha(t)$会减少。令 $\alpha = \max\limits_{t \geq 1} \{\alpha(t)\}$ 和 $\lambda = \alpha c$，则

$$\min_{t \geq 1} \left\{ \frac{||\overline{\boldsymbol{\pi}}^R(t)||_1}{||\overline{\boldsymbol{\pi}}^R(t+1)||_1} \right\} = \frac{1}{\max\limits_{t \geq 1}\{\alpha(t)\}c} = \frac{1}{\alpha c} = \lambda^{-1}.$$

λ^{-1} 可视为IFP1的收敛速度。一般而言，图中悬挂顶点的比例越高，IFP1的收敛速度就越快。在预定义阈值 ξ下，IFP1收敛所需的迭代次数可估算为

$$||\overline{\boldsymbol{\pi}}^R(T)||_1 = n \prod_{t=0}^{T-1} c\alpha(t) = \sum_{v_i \in V_1(t)} \overline{\pi}_i^R(T) + \sum_{v_i \in V_2(t)} \overline{\pi}_i^R(T).$$

由于当 $||\boldsymbol{\pi}_R(t)||_\infty < \xi$ 且 t_{th} 次迭代期间 $V_1(t)$上的质量降至0时，IFP1即收敛，因此一个充分条件是

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

因此

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

一般而言， $|V - V_U(T) - V_D| = O(n)$，我们有

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

3.3.2. 误差

由于图结构复杂，无法估算每个顶点的误差。我们讨论预定义阈值 ξ 对整体相对误差 $ERR(\xi)$的影响。假设 $||\underline{\boldsymbol{\pi}}^R(t)||_1 = n\lambda^t$，则

$$||\overline{\boldsymbol{\pi}}^I(T+t) - \overline{\boldsymbol{\pi}}^I(T)||_1 \leq n|\lambda^T - \lambda^{T+t}| = (1-\lambda^t)n\lambda^T.$$

设 $t \to \infty$，则可得

$$ERR(\xi) = \lim_{t\to\infty} \frac{||\boldsymbol{\pi}^I(T+t) - \boldsymbol{\pi}^I(T)||_1}{||\boldsymbol{\pi}^I(T+t)||_1} < \lim_{t\to\infty} \frac{|\lambda^T - \lambda^{T+t}|}{1-\lambda^{T+t}} = \xi. \qquad (6)$$

公式（6）可视为在预定义阈值 ξ下对相对误差的估算。需要指出的是，上述误差分析较为粗略，未考虑图结构的任何因素。

3.3.3. 计算量

IFP1 的计算主要由加法和乘法组成。用 $m(t)$表示第 t_{th} 次迭代的计算量，用 M 表示 IFP1 收敛时的总计算量。由于在第 t_{th} 次迭代中，只有属于 $V - V_U(t)$的顶点会产生算术运算，因此

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

设 $\beta(t) = \frac{m(t+1)}{m(t)}$。$\beta(t)$表示计算量的下降率。随着 IFP1 的运行，未被引用的顶点和弱无引用顶点逐渐退出计算，因此 $\beta(t) \leq 1$ 且

$$m + n = m(0) \geq m(1) \geq m(2) \geq \cdots \geq m(T).$$

令 $\beta = \max\limits_{1\leq t\leq T}\{\beta(t)\}$。假设 $\beta < 1$，即每次迭代都有顶点收敛，则有

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

其中 $T = O(\log_\lambda \xi)$。

总而言之，显然，IFP1 能够利用有向无环图结构：悬挂顶点可以提高收敛速度，而未被引用的顶点则能减少计算量。与幂法相比，在包含有向无环图结构的图上，IFP1 只需更少的迭代即可收敛，并且每次迭代产生的计算量也更少，因此速度更快。

3.4. IFP2

IFP1 无需悬挂顶点执行预留操作，但仍需要将质量从其源顶点推送到悬挂顶点，有时还会执行多次。具体而言，记 v_d 为一个悬挂顶点，记 $S(v_d)$ 为 v_d 的源顶点集合。每当 v_i 满足 $h_i > \xi$ 时，就会执行从 v_i 到 v_d 的推送操作。另一方面，我们有

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

其中 $\sum_{t=0}^{T} \pi_i^R(t)$ 是 v_i 的保留质量。这意味着，悬挂顶点上的质量完全由其源顶点决定。如果一开始不从源顶点向悬挂顶点推送质量，而是在所有非悬挂顶点收敛后再执行这些操作，那么只需执行一次推送操作即可。受此启发，本文提出了 IFP1 的改进算法 IFP2，即算法 3。

与 IFP1 不同，IFP2 需要处理所有顶点，非悬挂顶点和悬挂顶点都会分配给 K 个线程。IFP2 包含两个阶段：阶段 1 处理非悬挂顶点，阶段 2 处理悬挂顶点。阶段 1 与 IFP1 类似，但不会向悬挂顶点推送质量。阶段 1 结束时，所有非悬挂顶点均已收敛，随后阶段 2 开始向悬挂顶点推送质量。与 IFP1 相比，阶段 2 只执行一次，因此省去了与悬挂顶点对应的一些推送操作。具体而言，在第 t_{th} 次迭代中，可省去 $\sum_{v \in V_D - V_U(t)} |S(v)|$ 次质量推送操作，因此总共省去 $\sum_{t=0}^{T} \sum_{v \in V_D - V_U(t)} |S(v)|$ 次操作。一般来说，图中与悬挂顶点对应的边所占比例越高，IFP2 所需的计算量就越少。

需要指出的是，在阶段 1 运行之前，IFP2 需要进行一些预处理，将悬挂顶点对应的边设为无效。这些工作可能会增加时间消耗，但借助适当的数据结构，预处理的开销最多为 $o(\sum_{v \in V_D} |S(v)|)$。例-

以下实验将表明，无效化操作几乎不会增加额外的 CPU 时间消耗。

算法 3IFP2

输入：1

：K：线程数；2：ξ：质量下界。输出：3：$\boldsymbol{\pi}$：PageRank 向量。4：5：预处理图数据，并使悬挂顶点与其源顶点之间的无效边失效。6：每个顶点 v_i 维护一个数据结构 $\langle \pi_i, h_i \rangle$。7：将顶点分配给 K 个线程，分别用 S_j^1 和 S_j^2 表示属于线程 j 的非悬挂顶点集合和悬挂顶点集合。8：初始设置 $\pi_i = 0$、$h_i = 1$。9：调用 K 计算与管理；▷ [计算与管理并行执行。]10：在管理终止时，计算 π 如下所示。$\sum \pi_{ii=1}$11：12：函数 CALCULATION（j）13：当 1 时执行 14：如果 $CTRL$，则▷ [$CTRL$ 是布尔值，初始为 true。]15：阶段 1：16：对于 $v_i \in S_j^1$，执行 17：如果 $h_i > \xi$，则 18：$\pi_i = \pi_i + h_i$；19：对于 $u \in D\ (v_i)$，执行▷ [$D\ (v_i)$ 是 v_i的目标顶点集合。]20：$h_u = h_u + \frac{ch_i}{deg}\ (v_i)$；21：结束循环 22：$h_i = 0$；23：结束 if 24：结束循环 25：否则 26：阶段 2：27：如果 STS，则▷ [STS是布尔值，初始为 true。]28：对于 $v_i \in S_j^2$，执行 29：$\pi_i = h_i$；30：对于 $u \in S\ (v_i)$，执行▷ [$S\ (v_i)$ 是 v_i的源顶点集合。]31：$\pi_i = \pi_i + \frac{c\pi_u}{deg}\ (u)$；32：结束循环 33：$h_i = 0$；34：结束循环 35：$\neg STS = false$；36：结束 if 37：结束 if 38：结束循环 39：结束函数 40：41：函数 MANAGEMENT42：当存在满足 $h_i > \xi$ 的非悬挂顶点时，执行 43：结束循环 44：$CTRL = false$；45：当存在满足 $h_i > \xi$ 的悬挂顶点时，执行 46：结束循环 47：终止所有 K 计算。48：结束函数

4. 实验

本节通过实验展示 IFP1 和 IFP2。首先介绍所用的计算环境和指标，然后展示 IFP1 和 IFP2 的收敛情况，并与幂法进行比较。最后阐述 IFP2 在使用不同线程数时的实验结果。

4.1. 实验设置

本实验中的所有算法均使用 C++ 在配备 Intel(R) Xeon(R) Silver 4210R CPU 2.40GHz 处理器（40 个处理器）和 190GB 内存的服务器上实现。操作系统为 Ubuntu 18.04.5 LTS。表 1 展示了六个数据集，其中 n、m、n_d、m_d 和 $deg = \frac{m}{n}$ 分别表示顶点数、边数、悬挂顶点数、与悬挂顶点对应的边数以及平均度。我们采用 CPU 时间消耗 T 和最大相对误差 $ERR = \max_{v_i \in V} \frac{|\pi_i - \pi_i|}{\pi_i}$ 来评估算法，其中真实的 PageRank 值 π_i 由幂法在第 210_{th} 次迭代时求得。阻尼因子为 $c = 0.85$。

数据集	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-LiveJournal1	4847571	68993773	539119	2092984	14.23
uk-2002	18520343	298113762	2760973	16029357	16.10

表 1：数据集

4.2. 收敛

六个数据集上 ERR、T 和 ξ 之间的关系如图 1 所示 .

(1) 蓝色线条表明，ERR 与 ξ呈正线性关系，这与公式（6）一致。红色线条表明，T 与 ξ呈负指数关系，这与公式（5）一致。标有三角形的红色线条低于标有方块的红色线条，这表明在全部六个数据集上，IFP2 的速度都快于 IFP1。

(2) 蓝色线条表明，当 $\xi < 10^{-15}$时， ERR 几乎没有变化。看来 IFP1 和 IFP2 在精度方面都有局限。我们认为这并非算法缺陷，而是由于 C++的 DOUBLE 数据类型精度不足，其有效位数为 15。 $h_i < 10^{-15}$无法改变 π_i，因此 ERR 保持不变。

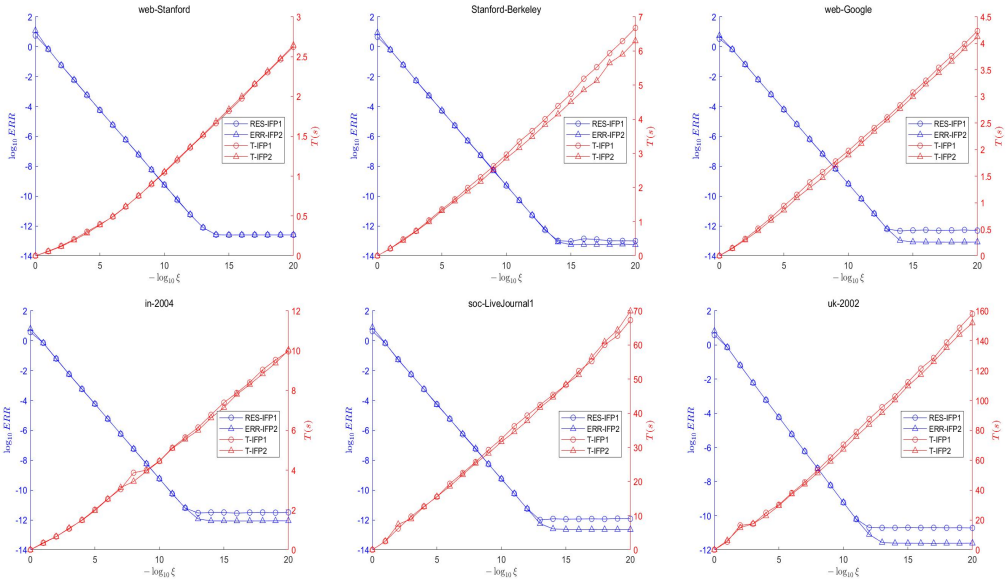

图 1： ξ 与 RES 和 T的对比

4.3. 与其他算法的比较

我们将 IFP1 和 IFP2 与幂法（SPI） [1, 2] 以及并行幂法（MPI）[27] 进行比较。MPI、IFP1 和 IFP2 均采用 38 的并行度执行。 ERR 与 T 之间的关系如图 2 所示。预处理的 CPU 时间消耗见表 2。表 3 展示了 T 在 $ERR < 0.001$时的 CPU 时间消耗。

（1）图2中，洋红色、红色和蓝色曲线均低于绿色曲线，这表明 MPI、IFP1 和 IFP2 的速度都快于 SPI，并行化是加速 PageRank 计算的有效方案。（2）图2中，洋红色和红色曲线均低于绿色曲线，这表明 IFP1 和 IFP2 的性能都优于 MPI。由于 IFP1、IFP2 和 MPI 采用相同的并行度执行，我们认为这种优势源于 IFP1 和 IFP2 更高的收敛速度。

（3）对于 MPI 和 IFP1，预处理是将顶点分配给线程；对于 IFP2，预处理还包括分配顶点以及使与悬挂顶点对应的边失效。MPI 会将所有顶点分配给线程，因此耗时更多。使边失效的时间消耗非常少，即使对于 uk-2002，也只需 0.155 秒即可完成。（4）表 3 显示，当 $ERR < 0.001$ 时，采用 38 并行度的 IFP2 最多可比 SPI 快 50 倍，最多可比 MPI 快 3 倍。

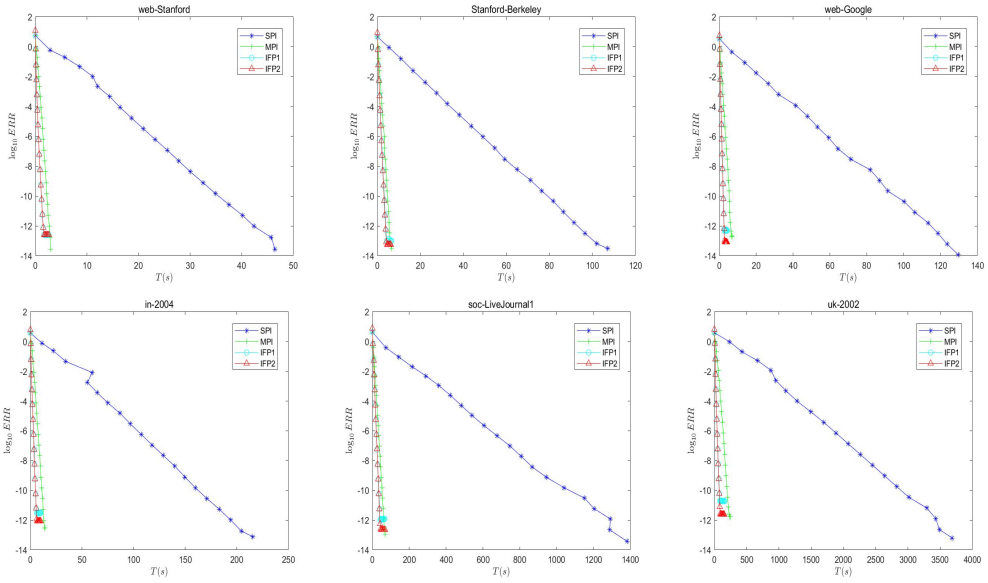

图2： T 与 ERR的比较

数据集	MPI	IFP1	IFP2
web-Stanford	0.014	0.012	0.012
Stanford-Berkeley	0.041	0.030	0.031
web-Google	0.043	0.033	0.036
in-2004	0.070	0.046	0.054
soc-LiveJournal1	0.236	0.178	0.194
uk-2002	0.926	0.629	0.784

表 2：预处理的时间消耗

数据集	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**

表 3： $ERR < 0.001$时的时间消耗

4.4 不同并行度下的性能

我们以 4、8、16、32 和 38 的并行度运行 IFP2。图3展示了不同并行度下 ERR 与 T 的关系。随着并行度提高，CPU 时间消耗 T 会减少。通常，并行度越高，计算速度越快。并行度为 16 的 IFP2 比并行度为 38 的 MPI 更快，这再次证明 IFP2 的收敛速度高于幂法。

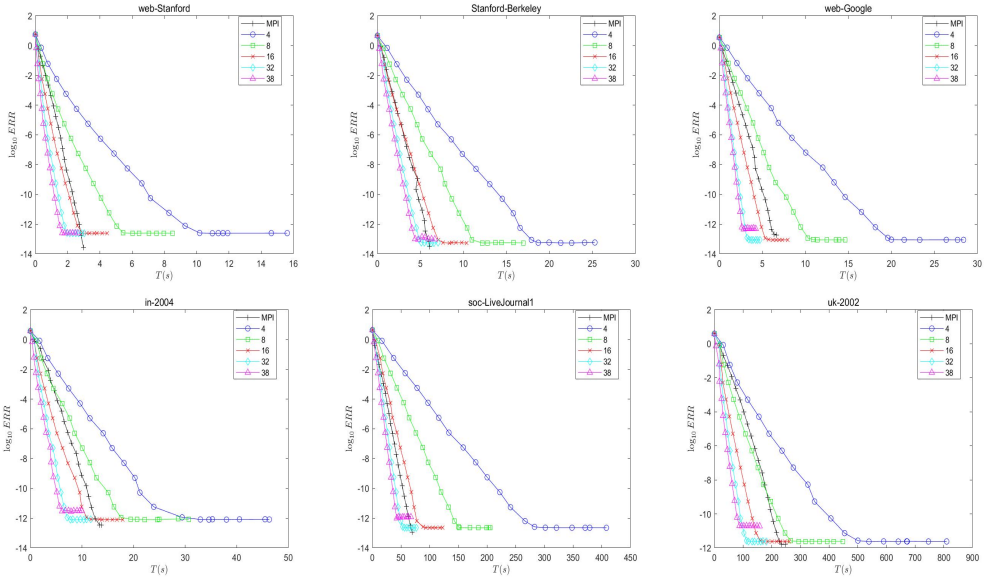

图3：并行度为 4、8、16、32 和 38 时， T 与 ERR 的对比

5. 结论

PageRank 是图计算中的一个基本问题，而并行化是加速计算的一种可行方案。本文首先揭示 PageRank 向量本质上是质量分布，并基于此提出了两种并行 PageRank 算法：IFP1 和 IFP2。IFP1 最突出的特点是能够充分利用有向无环图结构。一般而言，悬挂顶点所占比例越大，收敛速度越快；未被引用的顶点和弱无引用顶点所占比例越大，计算量越少。IFP2 只向悬挂顶点推送一次质量，因此，与悬挂顶点对应的边越多，计算量减少得越多。在六个数据集上进行的实验表明，IFP1 和 IFP2 都比幂法更快。然而，事物都有两面性；对于不含有向无环图结构的图（例如无向图），由于原子操作代价高昂，IFP1 和 IFP2 都无法胜过幂法。未来可以研究动态图上的 PageRank。

参考文献

[1] S. Brin, L. Page, 《大型超文本网络搜索引擎的剖析》重印版, Computer Networks 56 (18) (2012) 3825–3833。[2] L. Page, S. Brin, R. Motwani, T. Winograd, 《PageRank 引文排名：为网络建立秩序》, 技术报告，Stanford InfoLab (1999)。[3] D. F. Gleich, 《超越网络的 PageRank》, SIAM Review 57 (3) (2015) 321–363。[4] S. Brown, 《用于评估篮球、足球和冰球运动员表现的 PageRank 模型》, arXiv 预印本 arXiv:1704.00583 (2017)。[5] Ying, Hou, Zhou, Dou, Wang, Shao, 《基于唐宋文人足迹聚集分析的中心文化城市变迁》, 《地球信息科学学报》22 (5) (2020) 945–953。[6] S. Kamvar, T. Haveliwala, G. Golub, 《用于 PageRank 计算的自适应方法》, Linear Algebra and its Applications 386 (2004) 51–65。

[7] T. Haveliwala, S. Kamvar, D. Klein, C. Manning, G. Golub, 《使用幂外推计算 PageRank》，技术报告，斯坦福大学 (2003)。[8] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, G. H. Golub, 《加速 PageRank 计算的外推方法》，载于：第 12 届万维网国际会议论文集，2003 年，第 261–270 页。[9] S. Kamvar, T. Haveliwala, C. Manning, G. Golub, 《利用网络的块结构计算 PageRank》，技术报告，斯坦福大学 (2003)。[10] G. Wu, Y. Wei, 《用于计算 PageRank 的幂-阿诺尔迪算法》，《数值线性代数及其应用》14 (7) (2007) 521–546。[11] K. Avrachenkov, N. Litvak, D. Nemirovsky, N. Osipova, 《PageRank 计算中的蒙特卡罗方法：一次迭代何时足够》，《SIAM 数值分析杂志》45 (2) (2007) 890–904。[12] A. D. Sarma, A. R. Molla, G. Pandurangan, E. Upfal, 《快速分布式 PageRank 计算》，载于：国际分布式计算与网络会议，施普林格，2013 年，第 11–26 页。[13] S. Luo, 《分布式 PageRank 计算：一项改进的理论研究》，AAAI 人工智能会议论文集 33 (2019) 4496–4503。[14]S. Luo, 《降低分布式 PageRank 计算的通信成本——一项理论研究》，载于：国际机器学习会议，机器学习研究论文集，2020 年，第 6459–6467 页。[15] K. Sankaralingam, S. Sethumadhavan, J. Browne, 《面向点对点系统的分布式 PageRank》，载于：高性能分布式计算，2003 年第 12 届电气与电子工程师协会国际研讨会论文集，2003 年，第 58–68 页。doi:10.1109/HPDC.2003.1210016. [16] Y. Zhu, S. Ye, X. Li, 《基于迭代聚合-解聚方法的分布式 PageRank 计算》，载于：第 14 届美国计算机协会信息与知识管理国际会议论文集，2005 年，第 578–585 页。

[17] S. Stergiou，将 PageRank 扩展到 1000 亿个页面，载于：2020年万维网大会论文集，2020年，第 2761–2767 页。[18] H. Ishii、R. Tempo，PageRank 计算的分布式随机算法，《IEEE 自动控制汇刊》55 (9) (2010) 1987–2002。[19] H. Ishii、R. Tempo、E.-W. Bai，一种用于分布式随机 PageRank 算法的网络聚合方法，《IEEE 自动控制汇刊》57 (11) (2012) 2703–2717。[20] A. Suzuki、H. Ishii，通过结合网络聚类的分布式算法高效计算 PageRank，arXiv 预印本 arXiv:1907.09979 (2019)。[21] T. Charalambous、C. N. Hadjicostis、M. G. Rabbat、M. Johansson，有向图中特征向量中心性的全异步分布式估计及其在 PageRank 问题中的应用，载于：2016年IEEE第55届决策与控制会议（CDC），电气与电子工程师协会，2016年，第 25–30 页。[22] A. Suzuki、H. Ishii，基于一种新颖解释的 PageRank 分布式随机算法，载于：2018年美国控制会议（ACC），电气与电子工程师协会，2018年，第 472–477 页。[23] L. Dai 、 N. M. Freris，具有指数收敛速度的完全分布式 PageRank 计算，arXiv 预印本 arXiv:1705.09927 (2017)。[24] I. C. F. Ipsen、T. M. Selee，PageRank 计算，特别关注悬挂节点，《SIAM 矩阵分析与应用杂志》29 (4) (2008) 1281–1296。
arXiv:https://doi.org/10.1137/060664331,
doi:10.1137/060664331。网址： https://doi.org/10.1137/060664331
[25] Y. Lin、X. Shi、Y. Wei，通过聚合谷歌矩阵计算 PageRank，《计算与应用数学杂志》224 (2) (2009) 702–708。[26] Z. Zhu、Q. Peng、Z. Li、X. Guan、O. Muhammad，基于网络分解和有向无环图结构的快速 PageRank 计算，《IEEE Access》6 (2018) 41760–41770。

[27] N. T. Duong、Q. A. P. Nguyen、A. T. Nguyen、H.-D. Nguyen，使用图形处理器的并行 PageRank 计算，载于：第三届信息与通信技术研讨会论文集，2012年，第 223–230 页。[28] Lai，基于通用图形处理器环境的 PageRank 分数计算，博士论文，中山大学 (2017)。[29] H. Migall´on、V. Migall´on、J. Penad´es，求解 PageRank 问题的并行两阶段算法，《工程软件进展》125 (2018) 188–199。[30] Berkhin、Pavel，PageRank 计算综述，《互联网数学》2 (1) (2005) 73–120。