

并行计算技术课程期末研究报告

学号：2120210492 姓名：肖飞

目录

- 一、研究背景及意义2
- 二、国内外研究历史及现状.....2
- 三、实验原理6
 - 3.1、基本的 PageRank 算法.....6
 - 3.2、dead ends 和 spider trap 节点8
 - 3.3、稀疏矩阵优化与 Block-Stripe Update algorithm9
- 四、研究问题描述11
- 五、并行算法设计与实现11
 - 5.1、串行算法及其并行潜力分析11
 - 5.2、并行算法设计.....12
 - 5.3、数据集说明12
 - 5.4、算法实现.....14
- 六、实验及结果分析16
 - 6.1、性能分析.....16
 - 6.2、结果正确性分析19
- 七、可能的优化改进方法21
- 八、附件22
- 九、参考文献.....22

一、研究背景及意义

随着电子商务、物联网、社交网络、生物信息学等诸多领域的蓬勃发展，大量有价值的数据快速地产生并积累，“大数据”的计算处理成为近年来的研究热点之一。大量的统计结果显示，大数据应用问题普遍具有数据的总量大且增长速度快的特点。截止到 2016 年 3 月，Google 抓取的网页已超过 60 万亿，并以每天超过 450 亿个网页的速度增长 FaceBook 的用户数量已达到 13 亿并保持每秒 5 位用户的增长速度。当数据发生变化时，需要对数据重新处理以更新结果，因此，高效地处理频繁变化的大数据具有重大意义。

PageRank 算法在搜索引擎、信息检索、社交网络挖掘等多个领域得到广泛的应用，具有重要地位。PageRank 算法所处理的互联网络和社交网络数据是典型的频繁变化的大数据集，具有总量大、更新速度快、每次更新的数据占总量比例小的特点。为满足结果的时效性要求，只要数据更新就需要重计算以更新结果。在频繁变化的大数据集上反复地进行 PageRank 计算，会消耗巨大的计算资源，计算成本高。

二、国内外研究历史及现状

随着人们存储和处理数据量呈爆炸式地增长，如何衡量这些数据的重要性非常关键。通过搜索引擎获得有效信息是目前大众最常使用一个途径。PageRank 算法由 Lawrence Page 和 Sergey Brin 提出用于解决搜索引擎在万维网检索中遇到的挑战[1]。具体的挑战为当一个用户的搜索请求到来之后，搜索引擎在海量的网页中可以快速的检索包含了相应搜索关键的网页集合。在获得相应的网页集合后，搜索引擎需要决定这个网页集合中网页的展示顺序，这等价于如何评定网页的重要性。因此 Page 等人利用 PageRank 算法通过网页之间链接的关系计

算网页的 PageRank 值，在检索到的网页集合中依据各个网页的 PageRank 值的大小识别中哪些网页是相对更重要的和哪些网页是相对次要。PageRank 算法的重要思想在于模拟用户的浏览行为。假设用户浏览网页的行为类似于抛硬币的随机行为。如果是硬币的正面，则一个用户在浏览当前网页之后将随机点击页面上的链接，然后跳转到一个新的网页。如果硬币是反面，则一个用户在浏览当前网页之后将以一定的概率跳转到任意的网页，这个行为称为瞬移 (teleport)，瞬移是 PageRank 算法中非常重要的特性。我们称抛到正面的概率为阻尼系数 (damping factor)，抛到反面的概率为瞬移概率。PageRank 算法认为在随机浏览过程中出现概率越高的网页的重要性越高。基于网页之间的链接以判断网页重要性的算法除了 PageRank 算法外，还包括了康奈尔大学的 Kleinberg 等人提出的 HITS 算法[2]。随着 PageRank 算法研究的越来越广泛，PageRank 算法已经被应用到了网页查询[3]、社交网络分析[4]、生物学[5]等很多的领域。在这些应用场景上，我们都可以将问题转换成在图中顶点重要性的问题。

PageRank 算法是关联全局所有顶点的，因此算法复杂度很高。但是在有些应用场景中，只有少部分的顶点的 PageRank 值是需要，例如论文检索[6]、网页抓取[7, 8]、语义相关性分析[9]和链接作弊[10]。对所有顶点进行 PageRank 算法求解从而获得所需顶点的 PageRank 值并不是最恰当的解决方式，因为这样的方式将导致求解的效率不高。在这些情形中能够快速对单个顶点求解其 PageRank 值的是很有必要的。Chen 等人首先提出基于不需要访问整个的图的情况下，对少部分点进行 PageRank 值排序的解法，并定义为 Local PageRank 问题[11]。他们希望能够尽量减少对于存储图中信息的数据库的访问次数。他们的做法是通过构造一个包含目标顶点的子图，然后通过子图对目标顶点的 PageRank 值进行估计以达到减小需要访问图中顶点的规模。在构造子图的过程中，他们引入了一个“影响力”的概念以量化每个顶点对目标顶点的 PageRank 值的影响，该子图中包含了对目标顶点具有高影响力的顶点以对准确估计目前顶点的 PageRank 值。在子图构造完成后，利用 PageRank 算法对子图进行计算从而

减少整体的计算开销，因此子图构造的好坏决定了计算的开销和计算的精度。Bar-Yossef 等人在这个基础上对 Local PageRank 算法进行了改进[12]。在他们提出的方法中不需要通过构造子图的方式对目标顶点进行计算，而是利用相邻顶点的影响力直接计算目标顶点的 PageRank 值。因此 Bar-Yossef 等人方法以递归的方式利用顶点的入边计算影响力和目标顶点的 PageRank 值，从而进一步提高计算的精度和降低计算的开销。我们对需要全局所有顶点的 PageRank 值进行计算的问题定义为 PageRank 问题，并且定义只需要对部分目标顶点的 PageRank 值进行计算的问题为 Local PageRank 问题，从而对两类问题进行加以区分。这两类问题都涉及到了对顶点的 PageRank 值求解，因此对于 PageRank 问题的求解方法可以应用于 Local PageRank 问题的计算，Local PageRank 问题的求解方法也可以应用于 PageRank 问题的计算。但是由于前提条件的约束从而使得这两类问题的求解方法分别适用于不同的场景。尽管工业界如图 1-1 所示按着摩尔定律还在持续减小晶体管的大小，但是通过提高主频方式增强计算核心的方式已经很难满足当前对计算能耗。从架构的角度看指令级并行已经到达极限，单核计算很难也满足当前对计算并行性的需求。GPU (Graphics Processing Unit)、Xeon Phi[13]、Echelon[14]和 Runnemedede[15]等为代表的众核计算是维持摩尔定律的主要方式。GPU 以大量高效的计算单元和高 速的显存带宽满足图形计算对计算能力和内存带宽的巨大需求。

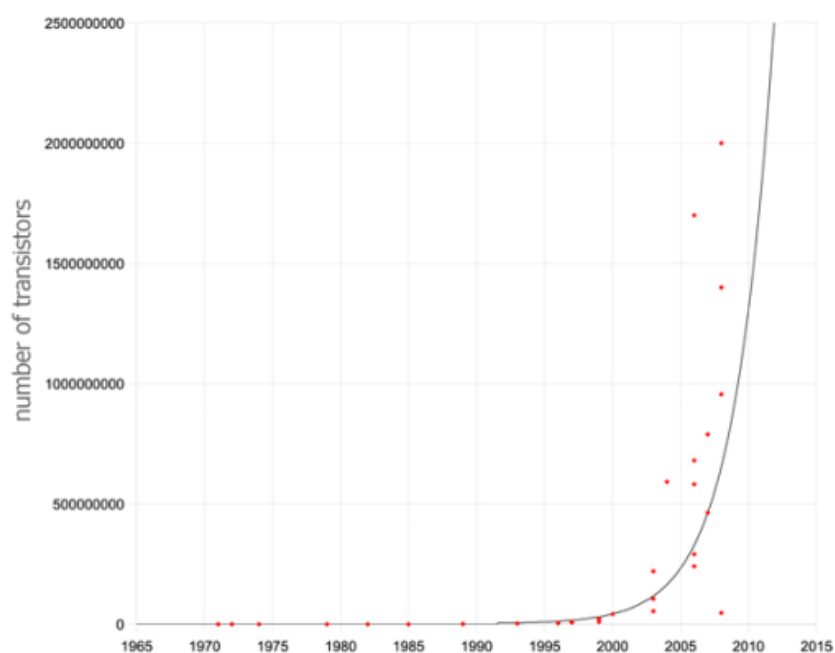


图 1-1 单位面积内晶体管的数量

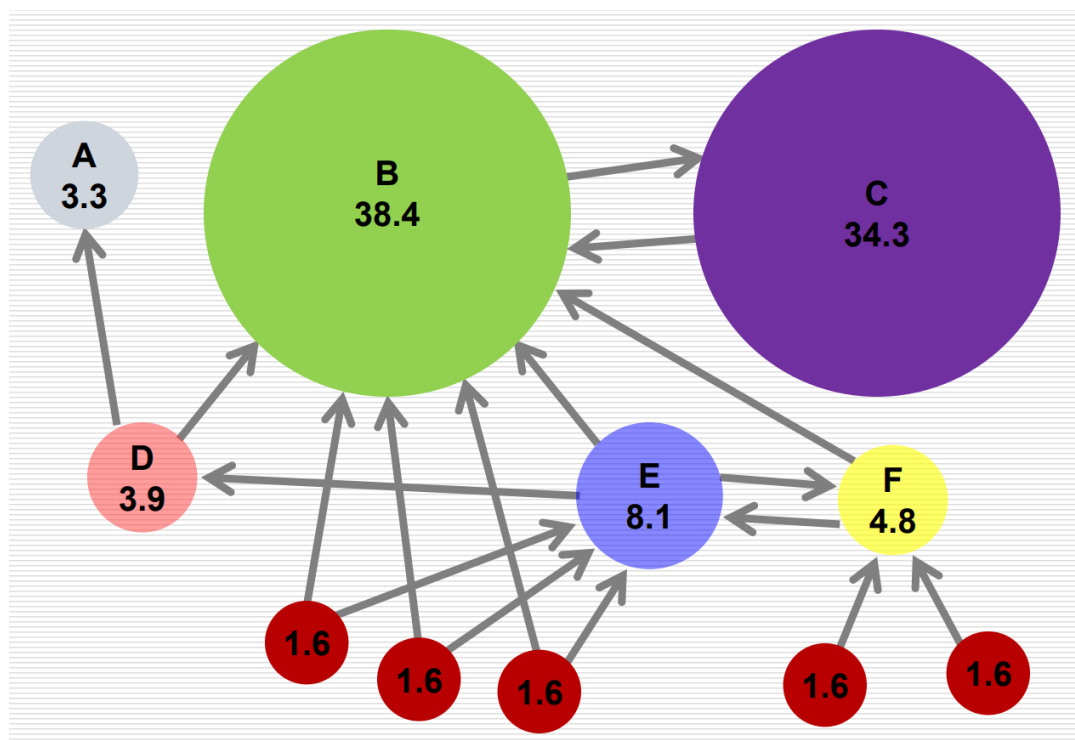
自 NVIDIA 公司在 1999 年发布第一代 GPU 起,随着 GPU 技术的发展和实际需求的变化, GPU 通用计算 (General-purpose computing on GPU, 简称 GPGPU) 成为了 GPU 重要的应用领域[16-21], GPGPU 利用了 GPU 强大的浮点计算能力对通用计算进行加速。为了让 GPU 有可用的编程环境, CUDA (Compute Unified Device Architecture,)、OpenCL (Open Computing Language)、OpenGL (Open Graphics Library) 等编程框架被提出, 从而能通过程序控制底层的硬件进行通用计算, 其中 CUDA 是 NVIDIA 公司提出 GPGPU 的编程框架。目前基于 GPGPU 的计算是近年来高性能计算研究领域中的一个重点关注方向。在最新发布的 Top 500 超级计算机排行榜上, 在使用众核体系的 96 台超算中有 60 台使用了 NVIDIA 的 GPU 进行加速[22]。GPU 相较于 CPU 在浮点计算能力以及内存带宽上有着明显的优势。在 PageRank 问题上, 近些年有一系列相关工作着眼于如何利用 GPU 加快 PageRank 算法的计算[23-29], 其中 Rungsawang 等人[23]和 Duong 等人[26]分别分析如何在 GPU 集群中实现 PageRank 算法的计算; NVIDIA 公司于 2016 年推出了可用于 PageRank 加速的

nvGraph 计算库[29]；Wu 等人则将 PageRank 计算转化成基于 SpMV (Sparse matrix-vector multiplication) 计算后利用 GPU 进行加速[24]，在 SpMV 方面利用 GPU 加速的方法也可有效借鉴至基于 GPGPU 加速的 PageRank 计算中[30-36]。此外许多图计算框架也利用 GPU 对 PageRank 进行加速[37-42]，PageRank 算法还被作为一种基准测试算法用于检验基于 GPGPU 的大数据系统的处理能力[43]。

三、实验原理

3.1、基本的 PageRank 算法

PageRank 是一种不容易被欺骗的计算网页重要性的工具，也是一个函数。它对 Web 中的每个网页赋予一个实数值，PageRank 越高，网页就越“重要”。



如上图，将 Web 结构以一张有向图来表示，网页 A、B、C、D、E、F 等分别表示为图中的一个节点，如果一个网页到另一个网页存在一个或多个链接，则这两个网页节点间存在一条有向边。每个节点都有一个 PageRank Scores，PageRank Scores 越大，说明该网页节点越

重要。一个网页的 PageRank Scores 实际上可以看作随机冲浪者处于该网页的概率，PageRank 认为较多冲浪者访问的网页的重要性高于较少冲浪者访问的网页，且 PageRank 方法的有效性已经在实际中得到验证。

以节点 D 为例，它有两条出链，分别链向节点 A 和 B，所以当前处于节点 D 的随机冲浪者下一步会以各 $1/2$ 的概率分别访问节点 A 和 B，如果当前处于节点 D 的概率为 a ，则下一步节点 A 和 B 分别会获得来自节点 D 的 $(1/2)a$ 概率；同样的，当前处于节点 D 的概率来自所有链入节点上一步的概率除以它们自己的出链数然后求和得到。如此循环一步一步迭代，直至最终各个节点的 PageRank Scores 较为稳定（在满足迭代收敛的条件下）。

一般地，定义一个 Web 转移矩阵来描述随机冲浪者的下一步访问行为。如果网页数目为 n ，则该矩阵 M 是一个 n 行 n 列的方阵。如果网页 j 有 k 条出链，那么对每一个出边链向的网页 i ，矩阵第 i 行第 j 列的矩阵元素 m_{ij} 值为 $1/k$ （即处于 j 的随机冲浪者将以各 $1/k$ 的概率访问每一个出边链向的网页 i ），而其他网页 i 的 $m_{ij} = 0$ 。另外随机冲浪者位置的概率分布可以通过一个 n 维列向量来描述，其中向量中的第 j 个分量代表冲浪者处于网页 j 的概率，该概率就是理想化的 PageRank 值。

假定随机冲浪者处于 n 个网页的初始概率相等，那么初始的概率分布向量就是一个每维均为 $1/n$ 的 n 维向量 v_0 。假定 Web 转移矩阵为 M ，则第一步之后随机冲浪者的概率分布向量为 Mv_0 ，第二步之后的概率分布向量为 $M(Mv_0) = M^2v_0$ ，其余以此类推。总的来说，随机冲浪者经过 i 步之后的位置概率分布向量为 $M^i v_0$ 。

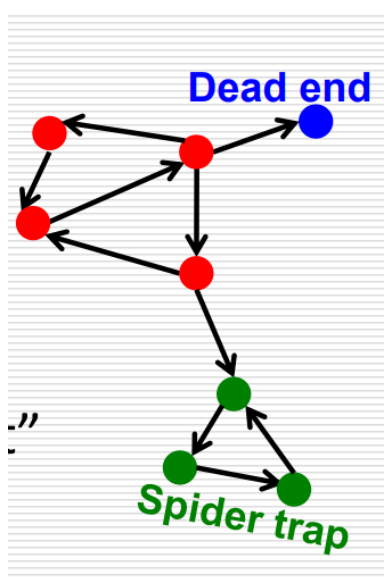
上述描述的行为实际上是一个马尔可夫过程。如果满足以下两个条件：

- a. 图是强联通图，即可以从任意节点到达其他节点；
- b. 图不存在终止点（dead ends），即那些不存在出链的节点。

则随机冲浪者的分布将逼近一个极限分布 v ，该分布满足 $v = Mv$ 。此时 v 是 M 的主特征

向量。如果直接通过高斯消去法求解该方程来得到 v ，其时间复杂度是节点个数的三次方，而在真实情况下，组成图的节点个数可能上百亿或上千亿，所以高斯消去法没有可行性，这种规模下的方程组求解只能通过迭代过程来实现，同时转移矩阵 M 往往非常稀疏，也可以利用这一点加快执行速度，最终直至前后两轮迭代产生的结果向量差异很小时停止。

3.2、dead ends 和 spider trap 节点



如上图，在实际情况下，Web 结构中可能会存在 dead ends（终止点）和 spider trap（采集器陷阱）。

如果允许终止点（没有出链的网页节点）存在的话，那么 Web 转移矩阵中某些列之和不为 1 而为 0，该转移矩阵就不再是随机矩阵了，这种情况下不断迭代，到达终止点的冲浪者会不断消失，这会造成所有能到达终止点的网页最终没有任何概率，最终的结果向量的部分或全部分量会变为 0，从而无法得到任何有关网页相对重要性的信息。

采集器陷阱指的是一系列节点集合，它们当中虽然没有终止点，但是也没有出链指向集合之外，从而导致在计算时将所有 PageRank 都分配到采集器陷阱之内。

上述两个问题都可以通过一种称为“抽税”的方法来解决。即允许每个随机冲浪者能够以一

个较小的概率随机跳转到一个随机网页，而不一定要沿着当前网页的出链前进。使用 PageRank 估计值 \mathbf{v} 和转移矩阵 \mathbf{M} （稀疏矩阵，压缩存储）估计新的 PageRank 向量 \mathbf{v}' 的迭代公式为：

$$\mathbf{v}' = \beta \mathbf{M} \mathbf{v} + (1 - \beta) \mathbf{e} / n$$

其中， β （teleport parameter）是一个选定的常数，通常取值在 0.8 和 0.9 之间； \mathbf{e} 是一个所有分量都为 1、维数为 n 的向量；而 n 是 Web 图中所有节点的数目。 $\beta \mathbf{M} \mathbf{v}$ 表示随机冲浪者以概率 β 从当前网页选择一个出链前进的情况。 $(1 - \beta) \mathbf{e} / n$ 是一个所有分量都是 $(1 - \beta) / n$ 的向量，代表一个新的随机冲浪者以 $(1 - \beta)$ 的概率随机选择一个网页进行访问。

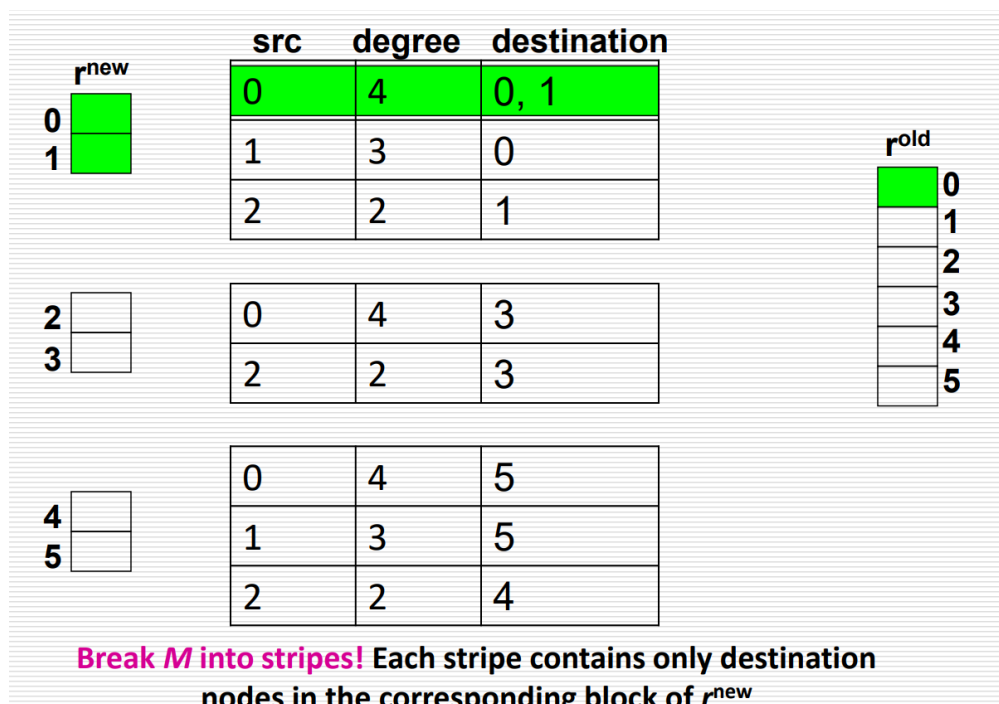
如果图中没有终止点，那么引入新的随机冲浪者的概率与随机冲浪者决定不沿着当前页面的出链前行的概率完全相等，这种情况下，把冲浪者想象成要不沿着某个出链前行，要不随机跳转到某个随机网页十分合理。如果图中存在终止点，那么冲浪者可能无处可走。由于 $(1 - \beta) \mathbf{e} / n$ 并不依赖于向量 \mathbf{v} 的分量之和，Web 的冲浪者总有部分概率处于 Web 之中。也就是说，即使存在终止点， \mathbf{v} 的分量之和可能会小于 1，但是永远不会为 0。如果图中存在采集器陷阱，PageRank 都分配到采集器陷阱之内的效果也会受到限制。

3.3、稀疏矩阵优化与 Block-Stripe Update algorithm

在实际情况下，转移矩阵会十分稀疏，一个表示任意稀疏矩阵的合理方法是列出非零元素值及其位置，即每一个非零元素值可以表示为一个三元组 (i, j, m_{ij}) ，其中 i 为行号， j 为列号， m_{ij} 为该非零元素值。在此基础上还可以进一步压缩，按列给出非零的元素，对每个列用一个整数表示该页面的出度，知道了出度就知道了该列中每个非零元素的值，同时对列中每个非零元素用一个整数来保存它所在的行号，即如下图所示

source node	degree	destination nodes
0	3	1, 5, 7
1	5	17, 64, 113, 117, 245
2	2	13, 23

另外，在当今的 Web 规模下， v' 很可能已经大到无法在内存存放，一个可行的方式是对 v' 进行分块，但是如果只对 v' 进行分块，每次迭代都需要读取扫描一次转移矩阵 M 和旧向量 v ，这会导致大量的开销： $k|M| + (k+1)|v|$ (k 为 v' 的分块数)。所以可以进一步优化，对转移矩阵 M 也进行分块，每个转移矩阵 M 分块对应一个相应的 v' 的分块，每次迭代只读取扫描 M 的一个分块即可，如下图



尽管这会使得转移矩阵 M 的存储有一些额外的空间开销，但是是值得的，开销为

$|M|(1+\epsilon) + (k+1)|v|$ (ϵ 代表了额外的开销)。

四、研究问题描述

基于给定的数据集 Data.txt，计算网站的 PageRank score，并按照从大到小排序，给出前 100 个 NodeID 和对应的 PageRank score，结果以 txt 格式保存，格式为 [NodeID] [Score]。

其中，参数 teleport parameter 选取 0.85，并且根据所学内容通过 pthread 等并行编程工具对主要的 Block-Stripe Update algorithm（分块迭代计算）进行并行化。

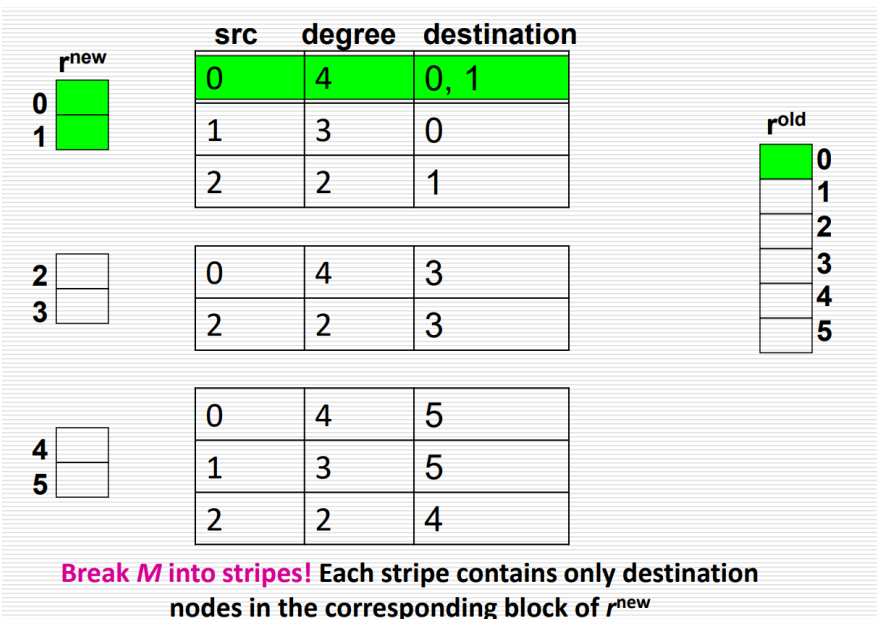
五、并行算法设计与实现

5.1、串行算法及其并行潜力分析

Block-Stripe Update algorithm 的主要代码逻辑在于迭代公式的实现：

$$\mathbf{v}' = \beta \mathbf{M} \mathbf{v} + (1 - \beta) \mathbf{e}/n$$

迭代的次数取决于达到收敛目标所需的次数，每次迭代的计算主要由两部分组成：前一部分的稀疏分块矩阵向量乘法，后一部分的向量标量乘法 and 加法。由于采用了下图这样的分块压缩存储方式：



稀疏矩阵的分块和新向量的分块之间有着彼此一一对应的关系，所以主要的稀疏分块矩

阵向量乘法串行算法就是遍历稀疏矩阵的每个分块，分块内部遍历每一列，并与旧向量中对应的元素相乘，然后加到新向量对应分块的对应元素上。显而易见，如果用 δ 表示稀疏矩阵的稀疏因子， n 表示问题规模（矩阵行数或列数，或向量元素数）， i 表示迭代次数， $links$ 表示网站链接数，则串行算法时间复杂度可以表示为 $O(n^2 \delta i)$ 或 $O(i * links)$ 。

可以看到，在稀疏分块矩阵向量乘法中，实际上各分块的计算彼此之间互不影响，没有相互的数据依赖关系，可以使用多线程并行执行；但由于矩阵元素和向量元素的不规则稀疏分布，使得难以通过 SIMD 并行化来对算法进行加速。

5.2、并行算法设计

对该串行算法多线程并行化适合采用输出数据划分的任务分解方式，即每个线程分配一个稀疏矩阵分块和对应的新向量分块，各自负责各自向量分块的迭代，旧向量由所有线程共享只读。各分块线程每次迭代完后需要同步（例如 barrier 同步），然后将本次迭代后的完整新向量作为共享只读旧向量开启下次迭代。

虽然稀疏矩阵的非零元素位置分布难以确定，但是为了尽量保持负载均衡，减小线程空闲等待，采取均匀的分块划分。同时由于需要多次迭代，如果每次迭代开始创建所有线程，迭代结束销毁所有线程，开销较大，所以编程范式采用静态线程范式，在程序初始化时创建好线程（池）。

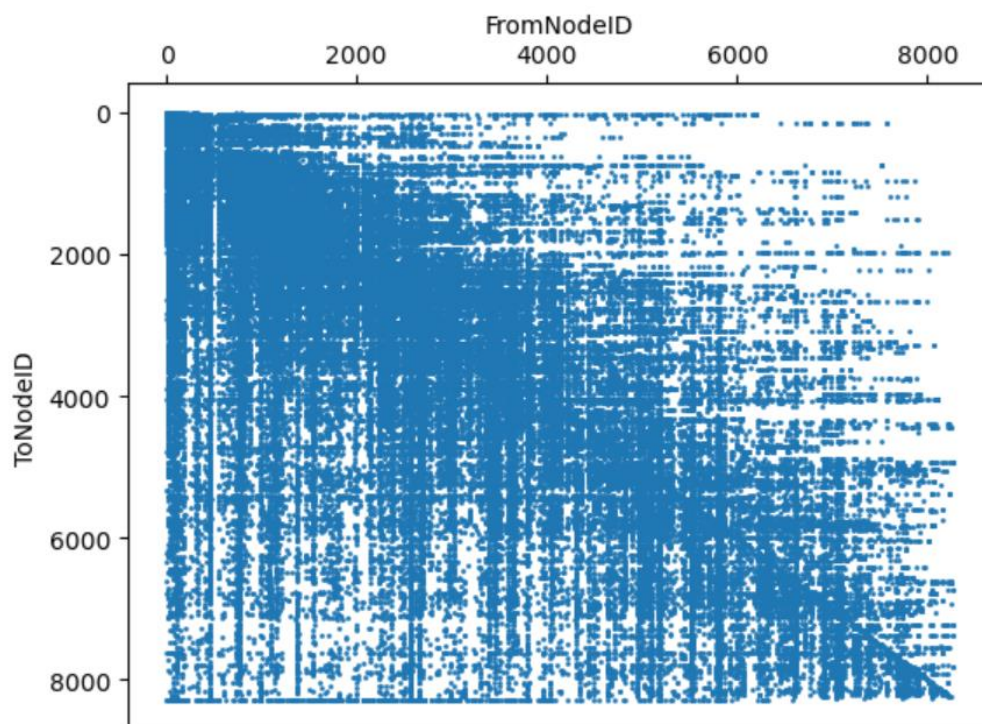
实验方面，改变输入数据集的大小（即稀疏矩阵稀疏因子或网站链接数）、线程数等参数，观测各算法运行时间的变化，对结果进行性能分析。

5.3、数据集说明

Data.txt 数据集是一个有向图的边列表，其中每一行表示一条有向边，即从一个网站到另

一个网站的链接，第一个数字 FromNodeID 表示边的起点网站 ID，第二个数字 ToNodeID 表示边的终点网站 ID。

在对应的转移矩阵中，非零元素的位置分布图如下



分别统计网站节点 ID 的最大值、最小值、节点个数以及链接个数如下：

```
#读取数据
with open('Data.txt', 'r') as f:
    lines = f.readlines()

# 解析数据
website_ids = []
num_links = 0
for line in lines:
    from_node, to_node = map(int, line.strip().split())
    website_ids.append(from_node)
    website_ids.append(to_node)
    num_links += 1

# 统计网站ID的最大值、最小值、数量
max_website_id = max(website_ids)
min_website_id = min(website_ids)
num_website_ids = len(set(website_ids))

# 输出统计结果
print(f"网站ID的最大值为{max_website_id}，最小值为{min_website_id}，数量为{num_website_ids}")
print(f"Data.txt中有{num_links}条链接")
```

网站ID的最大值为8297，最小值为3，数量为6263
Data.txt中有83852条链接

计算该矩阵的稀疏因子如下：

$$\delta = \frac{t}{m \times n} = \frac{83852}{8297 \times 8297} = 0.0012180671909348794 \leq 0.05$$

可见确实为稀疏矩阵。

5.4、算法实现

综上所述，本文要实现的算法版本包括以下：

- a. 串行算法版本
- b. 多线程并行化版本

同时测试不同问题规模 and 不同线程数下的算法性能表现。

串行算法的核心迭代流程部分代码如下：

```
//串行PageRank迭代
vector<OutputNode*> PageRank::calculate(SparseMatrix* matrix, PageRankVector* pageRankVector, PageRankParameter* parameter)
{
    PageRankVector* vOld = pageRankVector;
    PageRankVector* vNew = pageRankVector1;
    //cout << "初始vOld分量之和为: " << vOld->sum() << endl;
    //cout << "初始vNew分量之和为: " << vNew->sum() << endl;
    int numIter = 0; //迭代次数
    double diff = 1; //向量迭代前后差异
    double constant = (1 - teleportParameter) / vOld->getNodeNumber();

    //cout << "开始迭代:" << endl;
    while (diff > convergenceThreshold) { //迭代前后差异是否小于收敛阈值

        //迭代公式
        PageRankVector* v = matrix->multiply(vOld, vNew); //cout << "v->sum(): " << v->sum() << endl;
        vNew = v->multiply(teleportParameter)->add(constant);

        numIter++;
        //cout << "vOld分量之和为: " << setprecision(15) << vOld->sum() << endl;
        //cout << "vNew分量之和为: " << setprecision(15) << vNew->sum() << endl;
        diff = vNew->calculateDifference(vOld);
        //cout << "第" << numIter << "次迭代: 迭代前与迭代后的差异为: " << diff << endl;
        PageRankVector* temp = vOld;
        vOld = vNew;
        vNew = temp;
        vNew->memset();
    }

    //cout << endl;
    //cout << "本次PageRank迭代计算结束, 相关参数如下: " << endl;
    //cout << "随机跳转概率teleportParameter为: " << teleportParameter << endl;
    //cout << "收敛阈值convergenceThreshold为: " << convergenceThreshold << endl;
    cout << "总共迭代了" << numIter << "次" << endl;
    return vOld->sort();
}
```

并行多线程算法线程数据结构及相关全局共享变量定义：

```

//静态线程版本线程数据结构定义
typedef struct {
    int t_id; //线程 id
    int num_threads; //线程数
    PageRankVector* vOld; //旧向量
    PageRankVector* vNew; //新向量
    SparseMatrix* matrix; //稀疏分块矩阵
} threadParam_t;

pthread_mutex_t amutex;
pthread_barrier_t barrier_Iteration;
double diff = 1; //向量迭代前后差异
int numIter = 0; //实际迭代次数
double constant;
double converThre; //收敛阈值
double teleParam; //随机跳转概率
PageRankVector* pRVector; //保存最终结果向量

```

线程函数定义：

```

//线程函数定义
void *threadFunc_v(void *param) {

    threadParam_t *p = (threadParam_t*)param;
    int t_id = p->t_id;
    int num_threads = p->num_threads;
    PageRankVector* vOld = p->vOld;
    PageRankVector* vNew = p->vNew;
    SparseMatrix* matrix = p->matrix;

    while (diff > converThre) {

        pthread_barrier_wait(&barrier_Iteration);
        if(t_id == 0){
            diff = 0;
        }

        pthread_barrier_wait(&barrier_Iteration);
        matrix->multiply(vOld, vNew, t_id);
        vNew->multiply(teleParam, t_id)->add(constant, t_id);

        double myDiff = vNew->calculateDifference(vOld, t_id);

        pthread_mutex_lock(&amutex);
        diff += myDiff;
        pthread_mutex_unlock(&amutex);
        //cout << "第" << numIter << "次迭代: 迭代前与迭代后的差异为: " << diff << endl;

        PageRankVector* temp = vOld;
        vOld = vNew;
        vNew = temp;

        pthread_barrier_wait(&barrier_Iteration);
        if(t_id == 0){
            diff = sqrt(diff);
            pRVector = vOld;
            numIter++;
        }

        vNew->memset(t_id);

        //所有线程一起进入下一轮
        pthread_barrier_wait(&barrier_Iteration);

        //if(t_id == 0){
        //cout << "第" << numIter << "次迭代: 迭代前与迭代后的差异为: " << diff << endl;
        //}
        //std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    pthread_exit(NULL);
}

```

可以看到，使用了互斥量来保证临界资源的互斥访问，使用了 barrier 来同步所有线程，确保结果的正确性。在这一部分要注意不同环节步骤之间的数据依赖关系，尤其在主要的数据对象使用指针的情况下，否则结果比较容易出问题。

并行多线程算法流程代码：

```
//并行PageRank迭代
vector<OutputNode>* PageRank::calculate_parallel(SparseMatrix* matrix, PageRankVecto

    int num_threads = matrix->getNumBlocks(); //线程数

    //初始化barrier和mutex
    pthread_barrier_init(&barrier_Iteration, NULL, num_threads);
    pthread_mutex_init(&mutex, NULL);

    diff = 1;
    numIter = 0;
    constant = (1 - tParameter) / pageRankVector->getNodeNumber();
    converThre = converThreshold;
    teleParam = tParameter;

    //创建线程
    pthread_t handles[num_threads]; // 创建对应的 Handle
    threadParam_t param[num_threads]; // 创建对应的线程数据结构
    for(int t_id = 0; t_id < num_threads; t_id++){
        param[t_id].t_id = t_id;
        param[t_id].num_threads = num_threads;
        param[t_id].vOld = pageRankVector;
        param[t_id].vNew = pageRankVector1;
        param[t_id].matrix = matrix;
        pthread_create(&handles[t_id], NULL, threadFunc_v, (void*)&param[t_id]);
    }

    for(int t_id = 0; t_id < num_threads; t_id++){
        pthread_join(handles[t_id], NULL);
    }

    cout << "总共迭代了 " << numIter << " 次" << endl;

    pthread_barrier_destroy(&barrier_Iteration);
    pthread_mutex_destroy(&mutex);

    return pRVector->sort();
}
```

六、实验及结果分析

6.1、性能分析

程序不同问题规模下、不同算法/编程策略性能测试结果部分截图如下：


```

链接数Links: 10000

分块数/多线程的线程数: 2

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次
串行算法时间: 34.83ms

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次
多线程并行化算法时间: 21.6333ms

分块数/多线程的线程数: 4

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次
串行算法时间: 31.3644ms

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次
多线程并行化算法时间: 22.6777ms

分块数/多线程的线程数: 6

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次
串行算法时间: 34.3562ms

总共有3135个网站ID, 其中共有828个dead ends终止点
总共迭代了 159 次

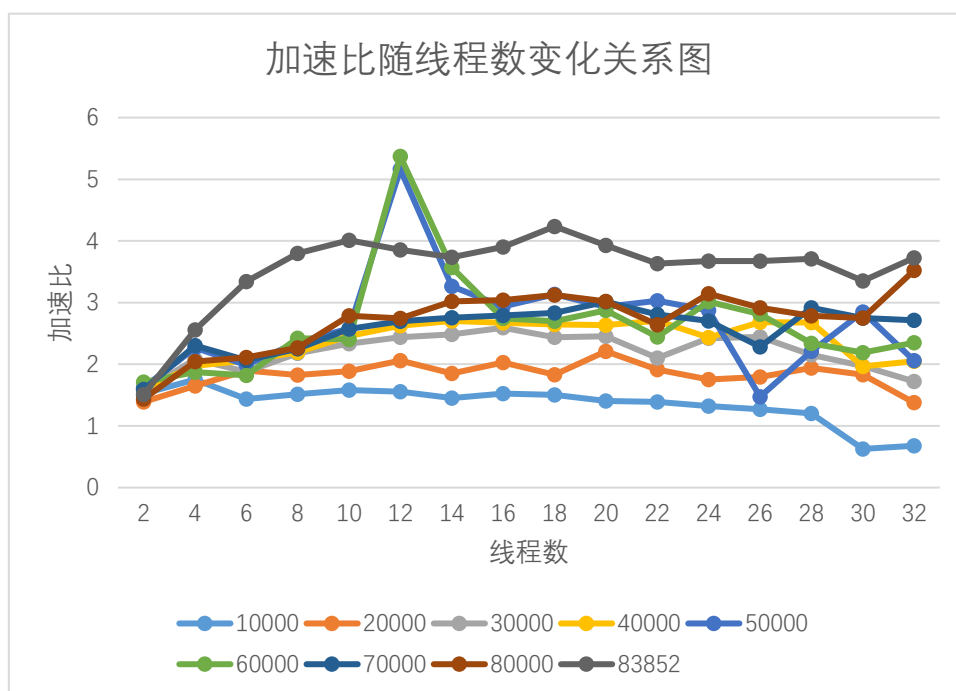
```

整理如下:

链接数	10000	20000	30000	40000	50000	60000	70000	80000	83852
网站数量	3135	3887	4413	4883	5289	5630	5911	6159	6263
迭代次数	159	143	145	156	155	62	62	60	60
串行分块数 2/ms	30.79	49.92	58.84	73.06	87.16	42.29	47.15	48.36	91.27
并行线程数 2/ms	20.21	35.95	37.33	45.57	53.72	24.72	29.64	33.64	60.45
加速比	1.523	1.388	1.576	1.603	1.622	1.710	1.590	1.437	1.509
串行分块数 4/ms	30.02	45.11	58.37	76.86	91.03	42.82	50.66	51.53	94.28
并行线程数 4/ms	17.13	27.37	28.07	38.91	40.13	22.88	22.00	25.23	36.85
加速比	1.752	1.647	2.07	1.974	2.267	1.871	2.302	2.042	2.558
串行分块数 6/ms	31.58	47.18	60.66	81.21	93.41	44.19	48.88	54.14	97.14
并行线程数 6/ms	21.97	24.84	32.26	39.47	46.96	24.30	23.70	25.63	29.09
加速比	1.43	1.898	1.879	2.057	1.988	1.818	2.062	2.111	3.339
串行分块数 8/ms	33.85	47.10	63.00	82.45	97.02	46.35	50.89	56.17	97.68
并行线程数 8/ms	22.36	25.77	28.90	37.54	41.19	19.12	22.60	24.80	25.72
加速比	1.513	1.827	2.179	2.196	2.355	2.423	2.251	2.264	3.796
串行分块数 10/ms	33.95	47.37	66.89	84.26	100.1	47.31	51.82	55.95	100.2
并行线程数 10/ms	21.4627	25.1093	28.6434	34.3019	38.8051	19.71	20.13	20.08	24.98
加速比	1.582	1.886	2.335	2.456	2.579	2.399	2.574	2.785	4.013
串行分块数 12/ms	34.29	48.19	65.38	86.13	211.6	107.5	53.69	57.30	100.9
并行线程数 12/ms	22.03	23.43	26.80	32.78	41.01	20.01	19.90	20.87	26.18
加速比	1.556	2.057	2.439	2.627	5.161	5.373	2.698	2.744	3.857
串行分块数 14/ms	34.65	50.60	66.80	88.79	123.4	56.21	54.12	56.86	105.2

并行线程数 14/ms	23.84	27.33	26.88	32.87	37.84	15.75	19.66	18.81	28.15
加速比	1.453	1.851	2.484	2.701	3.263	3.567	2.752	3.02	3.736
串行分块数 16/ms	35.06	56.53	68.84	88.24	107.7	49.41	56.03	58.76	106.5
并行线程数 16/ms	22.98	27.87	26.56	33.02	36.67	18.01	20.05	19.33	27.31
加速比	1.525	2.028	2.591	2.672	2.939	2.743	2.793	3.039	3.901
串行分块数 18/ms	34.59	56.23	68.53	90.85	109.2	50.81	55.89	61.35	121.5
并行线程数 18/ms	23.03	30.71	28.10	34.36	34.86	18.82	19.72	19.64	28.69
加速比	1.501	1.831	2.438	2.643	3.133	2.698	2.833	3.122	4.235
串行分块数 20/ms	34.51	58.81	67.95	92.35	109.4	52.97	57.85	64.68	113.8
并行线程数 20/ms	24.54	26.60	27.72	35.05	37.37	18.42	19.19	21.41	28.99
加速比	1.406	2.210	2.451	2.634	2.929	2.875	3.013	3.020	3.928
串行分块数 22/ms	35.14	51.43	70.48	95.58	113.1	51.59	58.77	65.02	108.1
并行线程数 22/ms	25.27	26.87	33.51	35.34	37.32	21.10	20.94	24.62	29.75
加速比	1.390	1.914	2.103	2.704	3.030	2.445	2.80	2.640	3.634
串行分块数 24/ms	34.69	52.21	73.24	96.88	111.9	55.26	58.60	64.06	109.3
并行线程数 24/ms	26.28	29.77	30.19	39.7	38.95	18.31	21.66	20.36	29.77
加速比	1.319	1.753	2.425	2.435	2.872	3.016	2.704	3.145	3.673
串行分块数 26/ms	33.67	53.28	80.22	97.51	113.9	53.92	60.78	64.07	111.1
并行线程数 26/ms	26.48	29.71	32.79	36.39	77.31	19.18	26.65	21.96	30.23
加速比	1.271	1.793	2.445	2.679	1.473	2.811	2.280	2.917	3.674
串行分块数 28/ms	35.62	59.74	78.49	98.31	181.4	54.41	64.02	65.19	112.4
并行线程数 28/ms	29.64	30.78	36.49	36.69	82.03	23.26	21.95	23.38	30.33
加速比	1.201	1.940	2.150	2.679	2.212	2.339	2.916	2.788	3.708
串行分块数 30/ms	50.80	63.46	77.81	96.78	127.7	53.90	63.74	65.30	113.1
并行线程数 30/ms	80.98	34.67	39.43	49.26	44.85	24.61	23.18	23.75	33.76
加速比	0.627	1.830	1.973	1.964	2.847	2.190	2.749	2.749	3.352
串行分块数 32/ms	51.15	56.69	72.17	99.11	115.0	55.02	69.34	109.0	116.5
并行线程数 32/ms	75.66	41.17	41.97	48.31	55.82	23.41	25.53	30.95	31.28
加速比	0.676	1.376	1.719	2.051	2.060	2.350	2.716	3.521	3.724

绘制不同链接数情况下加速比随线程数变化图如下：


























通过对以上结果进行分析，可以得到以下结论：

- 1、 除去一些异常点以及排除其他因素影响的话，在问题规模固定（链接数不变）条件下，当线程数小于等于 16 时（可看做线程数即为处理器数目。测试环境 CPU 核心数为 4P+8E，16 线程），加速比与线程数的关系基本符合 Amdahl 加速比模型。
- 2、 除了当问题规模较小、迭代次数较多时偶尔出现并行算法慢于串行算法外，多线程并行化算法大部分情况下获得的加速比至少在 2 左右，而限制加速比难以达到理想情况的主要因素是同步的操作使得程序中串行部分代码占据了一定的比例，并且线程数越多，同步的开销越大。

6.2、结果正确性分析

对于结果的正确性，每种链接数、线程数下，串行算法和并行算法各自会生成一份结果文件，串行算法的结果文件命名格式为“result_分块数_链接数”，并行算法为“result_parallel_线程数_链接数”，如下：

 result_30_83852.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_10000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_20000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_30000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_40000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_50000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_60000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_70000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_80000.txt	2023/7/27 9:30	文本文档	3 KB
 result_32_83852.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_10000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_20000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_30000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_40000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_50000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_60000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_70000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_80000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_2_83852.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_4_10000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_4_20000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_4_30000.txt	2023/7/27 9:30	文本文档	3 KB
 result_parallel_4_40000.txt	2023/7/27 9:30	文本文档	3 KB

相同链接数对应的结果文件，无论线程数/分块数如何变化，都应该是相同的。在链接数

相同条件下每种类型的结果文件各选一个，如下：

result_32_83852.txt			result_parallel_32_83852.txt			result_parallel_2_83852.txt		
文件	编辑	查看	文件	编辑	查看	文件	编辑	查看
4037 0.00187072911044603			4037 0.00187072911044603			4037 0.00187072911044603		
2625 0.00161501792166737			2625 0.00161501792166737			2625 0.00161501792166737		
6634 0.00148391127174924			6634 0.00148391127174924			6634 0.00148391127174924		
15 0.00133619421953771			15 0.00133619421953771			15 0.00133619421953771		
2398 0.00111313472585821			2398 0.00111313472585821			2398 0.00111313472585821		
2328 0.00110330897827902			2328 0.00110330897827902			2328 0.00110330897827902		
5412 0.000999596237518941			5412 0.000999596237518941			5412 0.000999596237518941		
7632 0.000962892687660873			7632 0.000962892687660873			7632 0.000962892687660873		
2470 0.000955979958563646			2470 0.000955979958563646			2470 0.000955979958563646		
3089 0.000949064577926984			3089 0.000949064577926984			3089 0.000949064577926984		
3352 0.000924376398406677			3352 0.000924376398406677			3352 0.000924376398406677		
737 0.000916210790896145			737 0.000916210790896145			737 0.000916210790896145		
2237 0.000908272301387497			2237 0.000908272301387497			2237 0.000908272301387497		
4191 0.000902330040099333			4191 0.000902330040099333			4191 0.000902330040099333		
5254 0.000897316870580154			5254 0.000897316870580154			5254 0.000897316870580154		
3456 0.000896130240752467			3456 0.000896130240752467			3456 0.000896130240752467		
7553 0.000890516984729817			7553 0.000890516984729817			7553 0.000890516984729817		
6832 0.000886151143805515			6832 0.000886151143805515			6832 0.000886151143805515		
2066 0.000824043152301863			2066 0.000824043152301863			2066 0.000824043152301863		
1297 0.00081977169428467			1297 0.00081977169428467			1297 0.00081977169428467		
7092 0.000808603187784819			7092 0.000808603187784819			7092 0.000808603187784819		
4310 0.000795787725967325			4310 0.000795787725967325			4310 0.000795787725967325		
6774 0.000775094190114867			6774 0.000775094190114867			6774 0.000775094190114867		
4712 0.000757654819567558			4712 0.000757654819567558			4712 0.000757654819567558		
762 0.000744092549699038			762 0.000744092549699038			762 0.000744092549699038		
1186 0.000726757466600943			1186 0.000726757466600943			1186 0.000726757466600943		
993 0.000713805019949374			993 0.000713805019949374			993 0.000713805019949374		
2958 0.000710220337344524			2958 0.000710220337344524			2958 0.000710220337344525		
665 0.000707214458163011			665 0.000707214458163011			665 0.000707214458163011		
6006 0.000703591630723214			6006 0.000703591630723214			6006 0.000703591630723214		
2657 0.000700927168534795			2657 0.000700927168534795			2657 0.000700927168534795		
4335 0.000688676657549704			4335 0.000688676657549704			4335 0.000688676657549704		

上述三个结果文件从左到右分别是串行的 32 分块数 83852 链接数的结果、并行的 32 线程数 83852 链接数的结果、并行的 2 线程数 83852 链接数的结果，截图均为排序最靠前的若干个网站 ID（左列）及其 PageRank 分数（右列）。可以看到排序结果完全相同，当然由于精度问题可能不同结果文件中同一个网站 ID 的 PageRank 分数的最后几位会略有差异，但是这种误差基本完全不会影响到网站的排序。

七、可能的优化改进方法

进一步的改进方向主要考虑以下方面：

- 1、 如果将向量分块中的有效分量组织为连续存储的数组，对于向量与标量、向量与向量的主要运算可以通过 SIMD 并行化，可能可以进一步获得更高的加速比和更好的并行效果。

- 2、 通过 GPU 进行加速。
- 3、 优化负载均衡。本文中的划分方式只是对向量进行均等划分，并以此为基础对稀疏矩阵进行了相应的划分，但实际上在不同数据集的一般情况下，向量的有效分量以及稀疏矩阵的非零元素的分布没有规律，只是对向量进行均等划分并不意味着每个分块中的有效分量也是均等的。如果刚开始数据预处理阶段能够对链接数（稀疏矩阵非零元素）、网站 ID 数（向量有效分量）进行统计，并以此为基础均等划分，获得更好的负载均衡，可以进一步优化线程空闲等待时间以及同步开销，从而整体获得更好加速比。

八、附件

Git 项目链接: <https://github.com/XFLasdf/PageRank>

九、参考文献

- [1] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: Bringing order to the web [Z]. Stanford InfoLab, 1999.
- [2] Kleinberg J M. Authoritative sources in a hyperlinked environment [J]. Journal of the ACM (JACM). 1999, 46(5): 604-632.
- [3] Brin S, Page L. The anatomy of a large-scale hypertextual web search engine [J]. Computer networks and ISDN systems. 1998, 30(1): 107-117.
- [4] Weng J, Lim E, Jiang J, et al. Twitterrank: finding topic-sensitive influential twitterers [C]. In: Proceedings of the third ACM international conference on Web search and data mining.ACM, 2010. 261-270.

- [5] Morrison J L, Breitling R, Higham D J, et al. GeneRank: using search engine technology for the analysis of microarray experiments [J]. BMC bioinformatics. 2005, 6(1): 233.
- [6] London A, Németh T, Pluhár A, et al. A local PageRank algorithm for evaluating the importance of scientific articles [C]. In: Annales Mathematicae et Informaticae.szte, 2015. 131-141.
- [7] Cho J, Garcia-Molina H, Page L. Efficient crawling through URL ordering[J]. Computer Networks and ISDN Systems. 1998, 30(1): 161-172.
- [8] Adar E, Teevan J, Dumais S T. Large scale analysis of web revisitation patterns [C]. In: Proceedings of the SIGCHI conference on Human Factors in Computing Systems.ACM, 2008. 1197-1206.
- [9] Tarau P, Mihalcea R, Figa E. Semantic document engineering with WordNet and PageRank [C]. In: Proceedings of the 2005 ACM symposium on Applied computing.ACM, 2005. 782-786.
- [10] Andersen R, Borgs C, Chayes J, et al. Local computation of PageRank contributions [C]. In: International Workshop on Algorithms and Models for the Web-Graph.Springer, 2007. 150-165.
- [11] Chen Y, Gan Q, Suel T. Local methods for estimating pagerank values [C]. In: Proceedings of the thirteenth ACM international conference on Information and knowledge management.ACM, 2004. 381-389.
- [12] Bar-Yossef Z, Mashiach L. Local approximation of pagerank and reverse pagerank [C]. In: Proceedings of the 17th ACM conference on Information and knowledge management.ACM, 2008. 279-288.

- [13] Chrysos G. Intel® Xeon Phi™ coprocessor-the architecture[J]. Intel Whitepaper. 2014, 176.
- [14] Keckler S W, Dally W J, Khailany B, et al. GPUs and the future of parallel computing [J]. IEEE Micro. 2011, 31(5): 7-17.
- [15] Carter N P, Agrawal A, Borkar S, et al. Runnemedede: An architecture for ubiquitous high-performance computing [C]. In: High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on.IEEE, 2013. 198-209.
- [16] Lee S, Min S, Eigenmann R. OpenMP to GPGPU: a compiler framework for automatic translation and optimization [J]. ACM Sigplan Notices. 2009, 44(4): 101-110.
- [17] Merrill D G, Grimshaw A S. Revisiting sorting for GPGPU stream architectures [C]. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques.ACM, 2010. 545-546.
- [18] Han T D, Abdelrahman T S. hiCUDA: High-level GPGPU programming [J]. IEEE Transactions on Parallel and Distributed Systems. 2011, 22(1): 78-90.
- [19] Maia J D C, Urquiza Carvalho G A, Manguiera Jr C P, et al. GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations [J]. Journal of Chemical Theory and Computation. 2012, 8(9): 3072-3081.
- [20] Trapnell C, Schatz M C. Optimizing data intensive GPGPU computations for DNA sequence alignment [J]. Parallel computing. 2009, 35(8): 429-440.
- [21] Lee J, Choi K, Kim Y, et al. Exploiting remote GPGPU in mobile devices [J]. Cluster Computing. 2016, 19(3): 1571-1583.
- [22] Top 500 list[Z]. <http://www.top500.org>.

- [23] Rungsawang A, Manaskasemsak B. Fast pagerank computation on a gpu cluster [C]. In: Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on. IEEE, 2012. 450-456.
- [24] Wu T, Wang B, Shan Y, et al. Efficient pagerank and spmv computation on amd gpus [C]. In: Parallel Processing (ICPP), 2010 39th International Conference on. IEEE, 2010. 81-89.
- [25] Cevahir A, Aykanat C, Turk A, et al. Efficient PageRank on GPU clusters [J]. IPSJ SIG Notes. 2010: 1-6.
- [26] Duong N T, Nguyen Q A P, Nguyen A T, et al. Parallel pagerank computation using gpus [C]. In: Proceedings of the Third Symposium on Information and Communication Technology.ACM, 2012. 223-230.
- [27] Choudhari P, Baikampadi E, Patil P, et al. Parallel and improved pagerank algorithm for gpu-cpu collaborative environment [J]. International Journal of Computer Science and Information Technologies. 2015, 6.
- [28] Kumar T, Sondhi P, Mittal A. Parallelization of PageRank on multicore processors [C]. In: International Conference on Distributed Computing and Internet Technology.Springer, 2012. 129-140.
- [29] Nvidia. nvGraph [Z]. 2016.
- [30] Su B, Keutzer K. clSpMV: A cross-platform OpenCL SpMV framework on GPUs [C]. In: Proceedings of the 26th ACM international conference on Supercomputing.ACM, 2012. 353-364.
- [31] Liu L, Liu M, Wang C, et al. LSRB-CSR: A low overhead storage format for SpMV on the GPU systems [C]. In: Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International

Conference on. IEEE, 2015. 733-741.

[32] Liu Y, Schmidt B. LightSpMV: Faster CUDA-Compatible Sparse Matrix-Vector Multiplication Using Compressed Sparse Rows [J]. Journal of Signal Processing Systems. 2016: 1-18.

[33] Steinberger M, Derlery A, Zayer R, et al. How naive is naive SpMV on the GPU? [C]. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE. IEEE, 2016. 1-8.

[34] Yang X, Parthasarathy S, Sadayappan P. Fast sparse matrix-vector multiplication on GPUs: implications for graph mining [J]. Proceedings of the VLDB Endowment. 2011, 4(4): 231-242.

[35] Yan S, Li C, Zhang Y, et al. yaSpMV: yet another SpMV framework on GPUs [C]. In: Acm Sigplan Notices. ACM, 2014. 107-118.

[36] Ashari A, Sedaghati N, Eisenlohr J, et al. Fast sparse matrix-vector multiplication on GPUs for graph applications [C]. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE Press, 2014. 781-792.

[37] Wang Y, Davidson A, Pan Y, et al. Gunrock: A high-performance graph processing library on the GPU [C]. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2016. 11.

[38] Zhong J, He B. Medusa: Simplified graph processing on GPUs [J]. IEEE Transactions on Parallel and Distributed Systems. 2014, 25(6): 1543-1552.

[39] Khorasani F, Vora K, Gupta R, et al. CuSha: vertex-centric graph processing on GPUs [C]. In: Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. ACM, 2014. 239-252.

- [40] Chen L, Huo X, Ren B, et al. Efficient and simplified parallel graph processing over cpu and mic [C]. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.IEEE, 2015. 819-828.
- [41] Khorasani F, Gupta R, Bhuyan L N. Scalable SIMD-efficient graph.
processing on gpus [C]. In: Parallel Architecture and Compilation (PACT), 2015 International Conference on. IEEE, 2015. 39-50.
- [42] Shi X, Liang J, Luo X, et al. Frog: Asynchronous graph processing on GPU with hybrid coloring model [J]. Huazhong University of Science and Technology, Tech. Rep. HUST-CGCL-TR-402. 2015.
- [43] Bisson M, Phillips E, Fatica M. A CUDA implementation of the pagerank pipeline benchmark [C]. In: High Performance Extreme Computing Conference (HPEC), 2016 IEEE.IEEE, 2016. 1-7.