计 算 机 研 究 与 发 展 DOI:10.7544/issn1000-1239.2014.20121044
Journal of Computer Research and Development 51(1):1-16,2014

一种大规模图数据处理关键技术的评估模型

高赟[1,2] 周薇[1,2] 韩冀中[1] 孟丹[1]

[1] 中国科学院信息工程研究所信息智能处理技术研究室，北京 100093

[2] 中国科学院大学，北京 100049

(gaoyun，zhouwei，hanjizhong，mengdan@iie.ac.cn)

An Evaluation Model on Key Technologies of Large-Scale Graph Data Processing

GAO Yun [1,2], ZHOU Wei[1,2], HAN Ji-Zhong[1] and MENG Dan[1]

[1] (Institute of Information Engineering, Chinese Academy of Science, Beijing, 100093, China)

[2] (University of the Chinese Academy of Science, Beijing, 100049, China)

Abstract With the development of social network analysis, knowledge graph and other graph based applications, processing large scale graphs with billions of vertices has been urgently needed and becomes the hotspot of both industry and academia. However, there is no comprehensive evaluation model to measure and compare the current mainstream framework. To solve the problem, in this paper we analyzed and summarized four key issues in large scale graph processing, including graph data distribution in memory, on-disk data organization, iterative programming model, message model and synchronization policy. Combined with the analysis of mainstream graph processing frameworks, this paper built a scalable and universal evaluation model to estimate the influence of these key issues on graph processing quantitatively. We also conducted comprehensive experiments and the results verified the effectiveness of our evaluation model. In our experiments we found some unusual phenomena. First, although compared with edge cut partition, vertex partition which has been thought as more effective could use only half of time on neighbors-based algorithms, but it may use 3 times of time on non-neighbors based algorithms. Second, compared with the synchronous policy, the asynchronous policy could decrease the total workload by 20%~30%, however, the running time may be twice due to fine grained lock conflicts on dense graphs. Third, disk-based framework MapReduce surpasses memory-based framework Giraph on performance when the graph increased to 40 million vertices and 1.3 billion edges.

Key words Large-Scale Graph Data; Performance Evaluation; Evaluation Model; Programming Model

摘要 随着社交网络、知识图谱等图应用的不断发展，对亿万个顶点级别大规模图的处理能力的需求愈加迫切，这是当前海量数据处理领域的研究和开发热点。但是，目前并没有一个全面的评估模型来衡量和比较当前主流框架的适用场景及利弊。针对以上问题，本文全面分析和总结了大规模图数据处理的四个关键问题，包括图数据分布策略、磁盘数据组织策略、迭代编程模型、消息模型与同步策略等。结合主流的大规模图处理框架，建立了评估模型定量地分析这些关键问题对大规模图数据处理的影响，对未来图计算框架的设计具有指导意义。

基金项目：本课题得到国家自然科学基金项目（60903047）;国家“八六三”高技术研究发展计划基金项目（No.2012AA01A401，No.2013AA013204）;中国科学院先导专项项目(No.XDA06030200)资助.
通讯作者：周薇，E-mail:zhouwei@iie.ac.cn

DOI:10.7544/issn1000-1239.2014.20121044
51(1):1-16,2014

最后通过全面的实验评测证实了本文提出的评估模型的有效性,在我们的测试结果中发现了如下不同寻常的现象:与图数据边分割相比,通常认为更快的顶点分割方法(如 PowerGraph)虽然在邻域算法上运行时间确实能够达到边分割的 50%左右,但是在非邻域算法上时间开销却是边分割的 3 倍;与同步策略相比,异步策略可以减少约 20%~30%的总计算量,但在稠密图上由于细粒度的锁冲突,其运行时间反而可能达到同步策略的 2 倍;当数据集达到 4 千万顶点和 13 亿条边时,基于磁盘的 MapReduce 比基于内存的 Giraph 等框架性能反而更高。

关键词 大规模图数据;系统评测;评估模型;编程模型

中图法分类号 TP391

1. 引言

图是一种由顶点和边构成的数据结构,由于它能够较好的表达实体间的关联关系,图与基于图的计算被广泛应用于各个领域,如社交网络,知识图谱,机器学习等[1, 2]。随着互联网的发展,需要处理的图的数据量也不断增大。例如,在社交网络应用中,截止到 2014 年,社交网络 FaceBook 一共有 13 亿用户[1]和 3 千亿条好友关系[2],Twitter 约有 6 亿用户[3]和 1 千亿条关注关系[4]。

为了充分挖掘这些海量数据中的有用信息,业界研究了一系列大规模社交网络数据的存储方法与计算模型。如 Pregel[3],Spark[4],PowerGraph[5]等。目前,为了提升图数据的处理数据,大部分研究工作都倾向于将数据载入内存以提升速度。并且,由于图算法的特性,其需要迭代计算以达到收敛。图数据处理大致可分为四部分,包括图数据的分布策略、磁盘数据组织、编程模型、消息模型以及同步策略。首先,将海量图数据划分为子图,按照图数据分布策略载入内存,将存储不下的图数据按照外存组织方式保留在磁盘上;然后,每个任务按照编程模型迭代计算属于自己的子图;之后,每次迭代的中间结果通过消息传递模型与其他任务共享,其他任务接收到消息后,更新当前值以用于下一次迭代;最后,在每两次迭代之间,需要一个全局同步以保证结果的一致性和正确性。

结合图数据处理的四个步骤,本文分析和总结了以下四个关键问题。第一,图数据的分布策略,以及相关的消息模型,直接影响了图算法的执行性能。第二,内存容量的增长速度远小于图数据的膨胀规模,因此,一部分图数据要溢出存储在磁盘上,图数据在磁盘上如何组织以提升速度是图计算领域的关键问题。第三,编程模型是指图计算任务的执行逻辑,图分析算法种类繁多,何种编程模型能够表达更多的图算法,又能获得高效的计算性能也是大规模图计算领域的一个重要问题。第四,每两次迭代之间需要同步,但是该操作具有短板效应的特性,直接受限于运行得最慢的任务,对图算法的执行造成较大影响。

针对以上关键问题,国内外学者展开了大量的研究工作。首先,[6]提出了磁盘和内存的混合组织模式,[7]设计了基于磁盘的图数据顺序组织与访问方式。然后,有多种编程模型可用于图算法,如 MapReduce,DAG(Directed Acyclic Graph,有向无

[1]http://www.statisticbrain.com/facebook-statistics

[2]http://www.statista.com/statistics/232499/americans-who-use-socialnetworking-sites-several-times-per-day

[3]http://www.statisticbrain.com/twitter-statistics

[4]http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/#.VAc209LAe2M

基金项目:本课题得到国家自然科学基金项目(60903047);国家“八六三”高技术研究发展计划基金项目(No.2012AA01A401,No.2013AA013204);中国科学院先导专项项目(No.XDA06030200)资助.
通讯作者:周薇,E-mail:zhouwei@iie.ac.cn

环图），BSP（Bulk Synchronization Parallel，大同步）等，这些编程模型提供不同的图计算执行逻辑。之后，许多框架使用了消息传递作为中间数据共享的机制，如 Pregel[3]，Trinity[8]等，随后，GraphLab[9]和 PowerGraph[5]提出了基于多副本的消息传递策略，该方式可以减少消息数目，进而减少运行时间，但是该方法表达能力有限，无法高效的支持部分图算法。最后，大同步方案是传统图算法中的必要阶段，其中 Pregel[3]，Hama[10]等都是基于大同步设计的，而 Trinity[8]和 GraphLab[9]提出了异步模型，取消了大同步的过程，但是也引入了一些问题，如适用面变窄，有一部分图算法无法表达。

基于上述分析，我们发现不同的框架在不同的场景下有不同的取舍。尽管这些不同设计的图计算系统为用户提供了更多的选择，根据机器、算法和数据等因素选择一个最优的框架并不是一个简单的任务。为了解决这一问题，本文全面分析和总结了大规模图数据处理的四个关键问题，包括图数据分布策略、磁盘数据组织、迭代编程模型与同步策略等。本文结合现有的大规模图处理框架，建立了评估模型定量的分析这些关键问题对大规模图数据处理的影响。最后通过全面的实验评测证实了本文提出的评估模型的有效性。另外，本文对大规模图数据处理中的关键问题分析也可用于指导未来的图数据处理框架的设计。总结而言，本文贡献如下：

1. 分析和总结了大规模图数据处理领域的四个关键问题。
2. 针对以上关键问题，建立了一个评估模型定量分析和比较现有解决方案的优缺点。上述评估模型和分析结果对未来大规模图数据处理框架的设计有一定的指导意义。
3. 在多个数据集上全面评测本文提出的评估模型，实验结果表明本文提出的模型具有很高的普适性和可扩展性。
4. 通过分析和实验，我们得到如下不寻常的结论：首先，与图数据边分割相比，通常认为更快的顶点分割方法（如 PowerGraph）虽然在邻域算法上运行时间确实能够达到边分割的 50%左右，但是在非邻域算法上时间开销可能达到边分割的 3 倍；其次，与同步策略相比，异步策略可以减少约 20%～30%的总计算量，但稠密图由于细粒度的锁冲突，其运行时间反而是同步策略的 2 倍；最后，当数据集达到 4 千万顶点和 13 亿条边时，基于磁盘的 MapReduce 比基于内存的 Giraph 等框架性能反而更高。

2. 相关工作

随着图分析算法的迫切日益需求以及数据量的不断增加，业界研究了一系列的大规模数据处理框架，使用这些框架可以简化图分析算法的分布式实现。MapReduce[11]是最早出现的面向分布式集群的并行编程框架，开创了分布式计算的新时代。针对 MapReduce 基于磁盘计算延迟过高的问题，Spark[4]引入了内存计算，通过合理利用内存来降低任务开销。相对于以上两种通用编程框架，Pregel[3]是针对图计算的专门框架，它引入了以图顶点为单位的编程框架以及大同步更新的策略。Giraph[5]是 Pregel 的开源实现，而 GPS[12]对 Pregel 在动态图切割等方面进行了改进。Trinity[8]同时提供了图数据管理与图数据分析的功能，其图数据分析框架与 Pregel 类似，但是引入了消息合并等优化机制。GraphLab[9]引入了多副本的图数据分布方式以及与异步更新的策略。针对由 Power Law 引起的任务间负载不均衡的问题，PowerGraph[5]提出了 GAS 图数据分布策略以及顶点切割的图分割策略。为了更好的将图计算作业与其它类型的作业结合，GraphX[13]实现了基于 Spark 的边分割图计算框架。针对内存计算的不足，GraphChi[7]提出了基于单机磁盘的图计算框架。

本文工作主要侧重在大规模图数据处理中的关键问题分析。之前的研究学者已经相关领域开展了大量工作。[14]讨论了 MapReduce 和传统并行编程模型之间的性能对比，[15]比较了 BSP 以及 MapReduce 等编程模型。然而，和本文工作相比，这些工作主要关注不同编程模型的理论分析，而没有建立评估比较模型，也没有通过实验评测验证理论分析的正确性。并且，这些相关工作主要侧重在大规模编程模型本身，并没有像本文一样关注在某个特定的领域，如图数据处理。

文章[16]评测和比较了现有的图算法和框架，但是，它并没有总结不同框架间的主要设计点，并且仅提供了 k-core 算法的比较。文章[17]仅通过实验比较了 MapReduce 与 Spark 上 PageRank 算法。文章[18]综述了云计算环境下大规模图数据的组织与计算方法，但是其对图计算框架的比较集中在 MapReduce 与基于 BSP 模型的 Pregel、Hama 的比较。文章[19]总结了 MapReduce 并行计算的进展，但其对于图计算的比较也集中于 MapReduce 与基于 BSP 模型的 Pregel 及其不同实现的区别。

在描述图计算框架的文章中也提供了部分性能

5 http:// www.giraph.apache.org

对比测试。例如，PowerGraph[5]的文章提供了 GAS（Gather-Apply-Scatter）方式与 SR（Send Receive）方式的比较，GraphChi[7]提供了 GraphChi 与其它框架的运行时间的比较。然而，这些评测通常仅仅关注该框架与其他框架在特定环境下的比较，并没有分析总结大规模图数据处理框架中的关键问题，也没有从各个方面对其做全面的实验评测。

3. 背景

3.1 图算法特征

为了挖掘图数据中的有用信息，业界提出了大量的图分析算法，如图的结构分析、遍历、聚类[1]等等。这些图分析算法有两个共同点，即面向顶点和多轮迭代：这些算法的计算目标是为每个顶点获取某个属性值，计算过程是通过迭代的方式根据图中其它顶点的当前值不断更新顶点值，直至收敛。一些常见的图算法和它的分类如表 1 所示。

本文分析总结常见的图算法。根据每轮更新中需要参考图中的哪些顶点，这些算法可以分为两类，即邻域算法和非邻域算法。在邻域算法中，每一轮迭代更新每个顶点属性值时仅需访问相邻顶点的值。而在非邻域算法中，更新每个顶点属性值还需要访问除邻域外其它顶点的值。

表 1. 常见的基本图算法和它们的分类。

算法	分类
BFS（Breadth First Search，宽度优先遍历）[20]	邻域算法
SSSP（Single Source Shortest Path，单源点是短路径）[21]	
PageRank[22]	
连通分支[23]	
SpMV（Sparse Matrix Vector Multiplication，稀疏矩阵乘）[24]	
MST（Minimum Spanning Tree，最小生成树）[25]	非邻域算法
图的粗化[26]	
图的稀疏化[27]	

3.2 图计算关键问题

由于图算法的面向顶点和迭代性质，大规模图数据处理一般可以分为如下步骤：首先将图数据按一定策略分割为子图[28-30]，然后启动多个任务并为每个任务分配一个或多个子图，各个任务同时更新所维护的子图数据，迭代直至收敛。针对这一过程，在图数据处理框架的设计中，需要考虑四个关键问题：

1. 图数据分布策略与消息模型：大规模图数据被划分为子图，不同分割方式导致了图数据分布策略不同，从而进一步影响了图数据的访问性能以及各个任务的负载均衡。此外，每两次迭代之间每个任务需要将自己生成的中间结果发送给其他任务，其他任务根据接收到的中间结果更新自己的值，然后再将更新后的值传递出去，以进行下一次迭代计算。不同的分布策略也决定了可用的消息模型，并进一步决定了消息数量和通信代价。
2. 磁盘图数据组织：虽然近年来可用内存大小已有较大提升，但是相对于数据量的增加速度仍然不足，因此有必要使用磁盘参与计算。基于磁盘的计算策略需要对磁盘数据结构、内存数据的换出进行组织来减少磁盘的随机读写。
3. 编程模型：编程模型是每个任务执行的程序逻辑。随着大数据的发展，涌现出了许多通用的大数据计算框架，如 MapReduce 和 Spark。图编程模型既可以复用这些已有的框架，也可以根据图算法的特性专门设计。不同的编程模型对图算法的支持也不尽相同，需要考虑如何设计或者复用现有的编程模型以达到高效的图计算性能。
4. 同步策略：每两次迭代之间需要同步，但是该操作具有短板效应的特性，直接受限于运行得最慢的任务，对图算法的执行造成较大影响。同步策略本质上是保证顶点更新的一致性，由于单个顶点数据的更新需要依赖图中其它的顶点数据，图顶点数据更新存在读写冲突，需要采用一定的同步策略来保证一致性。

4. 关键技术

本章分析总结了大规模图数据处理中的四类关键技术。第一节分析了图数据分布策略，以及与其紧密关联的消息模型，第二节对比内存不足时不同磁盘数据组织策略，第三节评估不同编程模型对计算性能的影响，第四节比较不同的迭代同步策略。

4.1 图数据分布策略与消息模型

本节讨论图数据分布策略，以及紧密依赖于图数据分布策略的消息模型对图计算性能的影响。

针对分布式图计算，业界提出了两种图数据分布策略与消息模型，即 SR（Send-Receive）方式[3]与 GAS（Gather-Apply-Scatter）[31]方式。SR 方式首先将图顶点分割成不相交的子图，而边则分配到起始顶点所在的子图中。对于某一顶点而言，其相邻顶点可能存在其它任务中，所以在计算过程中，顶点只能通过接收（Receive）其它顶点所发送（Send）消息的方式来访问其它顶点的数据。GAS 方式在做图分割时首先将边划分到不相交的子图，然后为每个顶点在所有存在邻边的子图中创建一个副本，并

选择其中一个作为主副本。在该策略下，图顶点需要访问相邻顶点时，不需要向其他任务发送数据，直接读取本任务中的副本即可。GAS 方式可以细分为三步：图顶点所有副本同时聚合相邻顶点的数据并将聚合结果发送给主副本（Gather）；主副本将收到的结果进一步聚合并根据聚合的结果更新顶点当前值，然后将将更新后的值同步到所有副本中（Apply）；每个副本使用更新后的值按需向相邻顶点发送消息（Scatter）。

首先，两种方式不同任务间的负载均衡情况不同。对于有V个顶点和E条边的图，图算法中每一轮的计算和通信复杂度一般为$O(V+E)$。在大规模图中，E远大于V，因此每一轮中任务的工作负载随着子图的边数目的增加而线性增长。文献[5]证明了与 SR 方式相比，GAS 方式的负载均衡程度更高，尤其是对于图顶点出度分布不均匀的图。这是因为在 GAS 方式中，出度较大的顶点可以被分割到多个任务中进行并行处理，而在 SR 方式中则只能由单个任务完成更新。

其次，两种方式支持的消息模型不同，从而导致 SR 方式与 GAS 方式执行相同图算法时需要发送的消息数量不同。在 GAS 方式中每一轮结束时每个顶点是否需要向相邻顶点发送消息可以将邻域算法分为两类。例如，PageRank 算法每轮每个顶点在 Gather 过程中直接从相邻顶点读取当前值进行更新，因此在 Scatter 过程时不需要显式发送消息。而对于 SSSP 算法，要求每个顶点在 Gather 过程时读取相邻顶点数据进行更新将导致大量无用更新，因此更优的策略是如果当前顶点值发生变化，则在 Scatter 过程时向相邻顶点发送消息且只有收到消息的顶点下轮才会执行更新。对于不需要在 Scatter 时发送消息的算法，SR 方式每轮需要发送$d\cdot|V|$条消息，其中$|V|$是活动顶点的数量而d是它们的平均出度。而在 GAS 方式中，每个顶点在 Gather 和 Scatter 过程可以直接访问相邻顶点的本地副本数据，仅仅在 Apply 过程前后需要各个副本与主副本进行通信与同步。因此，GAS 需要发送的消息数目为$k(r-1)|V|$，其中k是每个从副本与主副本每轮发送的消息数量，r是每个顶点的平均副本数。尽管[5]中证明了(r-1)|V| <= d|V|，但是由于一般情况下$k>1$，无法直接证明$M_{GAS}<M_{MP}$。但是，实际情况下，由于d一般较大，而r仅在 1 到 4 之间，因此 GAS 方式比 SR 方式发送的消息数更少。对于需要在 Scatter 时发送消息的算法，SR 方式需要发送的消息数量不变，而 GAS 方式除同步消息外还需要向相邻顶点发送消息。虽然 GAS 方式中的消息与 SR 方式中的消息看似一一对应，但是，与 SR 方式相比，GAS 方式可以根据相邻顶点的值过滤掉部分消息。比如在 SSSP 算法中，每个顶点不需要向已经获得更小值的顶点发送消息。因此，对于这些算法，$M'_{MP}=M_{MP}=d|V|$，且$M'_{GAS}=k(r-1)|V|+\beta d|V|$，β是被过滤掉的消息的比例。如果$(1-\beta)d<k(r-1)$，SR 方式发送的消息更少，反之 GAS 方式发送的消息更少。

第三，两类方式适用的算法范围不同，在 GAS 方式上实现非邻域算法比在 SR 方式上有更高的复杂度。例如，对于非邻域算法，一类比较常见的操作是在一个子图中将一个顶点的数值同步到整个子图中[32]。假设子图的直径是R，在 SR 方式上这一操作可以通过指针跳跃算法[33]在$logR$次迭代内完成，而 GAS 方式上则需要使用洪泛算法在R次迭代内完成。这是因为 GAS 方式限定每个顶点只能和相邻顶点进行通信，在同步过程中，消息每步只能传递一跳。另一方面，直接在 GAS 方式上忽略掉 Gather 过程和 Scatter 过程，在 Apply 过程中按照 SR 方式的方式直接发送消息仍然不高效。因为对于非邻域算法，相邻顶点的值无法用于消息过滤，与 SR 方式相比，每轮需要多发送$k(r-1)|V|$条消息进行顶点副本同步。

通过以上分析可以看出，GAS 的数据分布方式和消息模型对邻域算法进行了专门的优化，而 SR 方式则可以更好的支持非邻域算法。

4.2 磁盘数据组织

尽管集群内存的规模已经随着硬件水平的发展有了较大的提高，单纯的基于内存进行计算，即将所有顶点数据、图结构以及中间消息存放于内存，其可扩展性仍然受到内存大小的限制。例如，在 twitter 数据集（4 千万顶点，13 亿条边）上执行 MST 算法时，64G 内存中无法同时存放顶点数据和图结构。因此，在较小的集群规模上进行大规模图分析时仍然需要基于磁盘的计算框架。由于图计算数据访问的随机性，直接依赖于操作系统的换页机制进行基于磁盘的计算将导致大量的磁盘随机访问和换入换出，从而极大的降低计算的效率。因此，需要为图计算专门设计磁盘计算策略。

基于磁盘计算的一般策略是在内存中维护缓冲区，按需与磁盘上的数据进行换入换出，只有缓冲区内的数据可以参与计算。图算法中需要维护的数据分为三类，即顶点数据，图结构和中间消息，而图算法每轮操作的最小单位是对单个顶点调用更新函数，更新函数需要同时访问三类数据中该顶点对应部分。因此，图算法基于磁盘计算时在内存中维护固定大小缓冲区并对顶点数据进行流式遍历，并

对缓冲区中的顶点调用更新函数。由于图结构数据保持不变，图结构数据按顶点顺序预先排序，并按照与消息数据相同的顺序换入换出。而对于中间消息数据，由于每一轮消息数据均需要重新接收，接收到的消息数据需要按一定顺序来组织以保证可以高效的按顶点顺序遍历。现有的基于磁盘的图计算框架，即 MapReduce, Giraph 和 GraphChi 采用不同的方式组织和访问消息数据，本节将比较这三种方式。

MapReduce 采用预先使用外排序算法将消息进行排序的方式对数据进行组织。MapReduce 是专门针对磁盘进行计算的框架，其设计的目标之一是最小化内存使用。预先排序使得随后的消息数据的访问可以使用固定大小缓冲区顺序进行，从而实现了最小化内存开销的目标。另外，预先排序和随后访问对消息数据的访问均顺序进行，从而避免了额外的磁盘随机访问。但是，预先外排序的方式增加了每次迭代执行的开销，而且这种方式无法有效利用多余的内存，即使在内存充足的情况下也难以降低开销。

Giraph 通过维护多个排序的溢出文件，并且在需要读取消息数据时打开多个溢出文件同时读取。在 Giraph 中，每个任务维护一个固定大小的缓冲区并按顶点数据对消息进行组织，当缓冲区满的时候，Giraph 将缓冲区内容写入到磁盘上的溢出文件中。下一轮中间消息参与计算时，Giraph 同时打开所有溢出文件。由于每个溢出文件都按顶点顺序排序，对单个溢出文件的读取是顺序的。这种方式避免了预先排序带来的额外开销，对于小的数据集，这种方式可以减少处理延迟，但是，对于大的数据集，溢出文件数目过多，同时读取这些文件时虽然单个文件顺序访问，在不同溢出文件之间的切换仍然会导致大量的随机磁盘访问，从而使处理时间迅速增加，进一步导致较差的扩展性。

GraphChi 采用的预分配存储空间的方式组织消息。为了实现高效的中间消息数据组织，GraphChi 限制所支持的算法：GraphChi 假设每一轮沿每条边均只发送一条固定大小的消息。采用这种方式，每轮消息的数量和大小都是固定的，GraphChi 可以在磁盘上为每条消息预留存储位置，收到的消息可以直接写入到预留位置中。由于消息预留的存储空间可以预先按顶点顺序进行排序，GraphChi 避免了计算过程中每一轮对消息进行排序的开销以及同时打开多个文件导致的随机磁盘访问。但是，这种方式限制了算法的表达能力。例如，SSSP 每轮发送的消息数量不一致，MST 算法中消息的大小可能变化，在 GraphChi 上，这些算法均无法高效实现。

从上述分析可以看出，三种磁盘数据组织方式有不同的应用场景。对于相对较小的数据集，同时打开多个溢出文件的额外开销较小，而对于大的数据集，预先的排序合并可以减少磁盘随机访问的次数。此外，如果仅考虑支持特定的算法，预先分配存储空间的开销最小。

4.3 迭代编程模型

在确定了数据组织和消息模型之后，采用不同的编程模型完成图算法的流程不同，其代价也不同。当前可以用于图计算的编程模型包括 MapReduce 模型、DAG 模型和顶点编程模型。本节分析不同编程模型的执行效率。

MapReduce 是一种通用的两步计算模型。它首先通过 Map 操作将输入的 Key-Value 数据转化为对应的中间 Key-Value 数据，然后通过 Reduce 操作将 Key 值相同的 Key-Value 数据聚合以得到最终结果。在 Reduce 操作之前 MapReduce 通过 Shuffle 操作来交换 Map 任务与 Reduce 任务间的数据。

算法 1. 基于 MapReduce 模型的图算法实现

输入：

f: 消息生成函数，即根据目标顶点和当前值生成要发送的消息

g: 顶点更新函数：根据顶点当前值、邻边和收到的所有消息更新顶点的值

输出：无

METHOD *map*(Vertex *v*, VertexValue *vv*, Edge *E*)

FOR $e \in E$; **DO**

输出消息中间键值对 m_e： $\mathrm{e.to} \rightarrow \mathrm{f(vv, e.weight)}$

END FOR

输出图结构中间键值对m_{gs}：$\mathrm{v}.id \rightarrow (vv, E)$

END METHOD

METHOD reduce(Vertex *v*, $m_{gs}, [m_1, m_{2,} \cdots])$

$(v, vv, E) \leftarrow m_{gs}$

$vv' \leftarrow g(vv, [m_1, m_{2,} \cdots])$

输入结果键值对：$\mathrm{v}.id \rightarrow (v, vv', E)$

END METHOD

DAG 模型提供了一种基于有向无环图的数据转换视图。在数据转化图中，每一个顶点是一个数据集，而每一条边定义了从输入数据集到输出数据集的转换操作。在每次转换中，DAG 编程模型将输入数据集进行切分并为每一个数据分片启动一个子任务。为了降低开发复杂度，提高并行度，DAG 模型对框架的表达能力进行了限制。它假定所有的数据集都是只读的并且只能用于输入，从而避免细粒

度的同步；另外，它要求同一转换中所有子任务相互独立，只有在相邻转换的不同任务间才能进行数据 shuffle 操作。

顶点编程模型是一种专门为图计算设计的模型。顶点编程模型直接提供了顶点和边等图的语义，并且允许用户进行细粒度的数据访问。它在内存中直接维护与顶点和边相对应的数据结构并允许用户在顶点更新时直接读取和修改这些数据结构；除图结构外，顶点编程模型允许用户在更新函数中显式地向其它顶点所送消息，并能够访问之前收到的来自其它顶点的消息。

由于图算法需要多轮迭代并且在相邻两轮迭代之间需要进行数据交换，图算法的一轮迭代需要被翻译成 MapReduce 模型中的一个独立的作业，DAG 模型中若干个相邻的转换以及顶点编程模型中对所有顶点更新逻辑的一轮调用。基于三种模型的框架实现 SR 方式如算法 1，2 和 3 所示。

算法 2. 基于 DAG 模型的图算法实现

输入：

f：消息生成函数，即根据目标顶点和当前值生成要发送的消息

g: 顶点更新函数：根据顶点当前值、邻边和收到的所有消息更新顶点的值

输出：无

METHOD *dag_main*(*edge_data, vertice_data*)

$M \leftarrow E.join(V).map(f)$ //生成第 0 轮中间消息

WHILE 未收敛; **DO**

$U \leftarrow M.reduceWithKey(g)$ //求顶点更新数据集

$V \leftarrow V.join(U)$ //更新顶点数据

M $\leftarrow E.join(V).map(f)$ //生成下一轮的消息

END WHILE

END METHOD

对于图算法，每轮迭代必需的工作包括调用顶点更新函数根据相邻顶点或接收的消息更新当前顶点，以将按需要将更新后的数据传播到相邻顶点或其它副本。对于 MapReduce 和 DAG，由于它们为支持更通用的计算问题所作的设计妥协，在用于实现图计算模型时存在额外开销。对于 MapReduce 模型，由于 MapReduce 没有提供中间数据管理，相邻两轮之间通过 HDFS 交换顶点更新后的值。另外，图结构的信息，如顶点的邻边或副本的位置信息，也需要与顶点数据一起参与 Shuffle 操作以及 HDFS 交换，否则每一轮 MapReduce 需要连接顶点数据、顶点消息与图结构信息，从而导致更大的开销。最后，MapReduce 每轮还有一个不可忽略的作业启动开销。

与 MapReduce 相比，DAG 编程模型不需要通过 HDFS 进行数据交换，并且通过将图的结构数据与顶点数据分离存储以避免图结构数据参与 shuffle 操作。由于 DAG 编程模型提供中间数据的维护，顶点数据和图结构可以按相同方式分割，每一轮中这两个数据集的连接可以通过本地连接进行。另外，由于每个数据集是只读的，顶点数据的更新无法直接写回，而是需要产生一个顶点更新数据集并与原数据集进行连接和聚合。由于顶点更新数据集也可以采用与顶点数据集相同的分割方式，它们之间的连接也可以通过本地连接进行，因此 DAG 编程模型每轮还需要两个额外的本地连接开销。

算法 3. 基于顶点编程模型的图算法实现

输入：

f：消息生成函数，即根据目标顶点和当前值生成要发送的消息

g: 顶点更新函数：根据顶点当前值、邻边和收到的所有消息更新顶点的值

输出：无

METHOD *vp_main*(*graph_file, partition_id*)

WHILE 未收敛; **DO**

FOR $v \leftarrow V$;**DO**

$v.value = g(v, v.M_{in})$

$M_{out} \leftarrow f(e, v.value, v.neighbors)$

发送消息M_{out}

END FOR

END WHILE

END METHOD

与 MapReduce 和 DAG 相比，顶点编程模型最为高效。由于它是为图计算专门设计的模型，顶点和边的修改可以直接写回，从而避免了额外数据集的生成和连接。此外，顶点编程模型可以采用多线程将顶点值的更新、消息发送与接收通过多线程并发处理，使各类开销相互重叠，进一步降低了执行开销。

假设每一轮需要时间T_c来计算更新后的顶点值和生成消息，顶点数据集与图结构数据集的大小为S_v和S_g，网络栈需要T_{queue}来将一条消息加入队列且带宽为B，每轮发送的消息数量为|M|，那么基于顶点编程模型实现 SR 方式的本轮计算时间为

$$O_{VP} = \max\left(T_c + |M|T_{queue}, \frac{S_v}{B}\right) \quad (1)$$

假设磁盘带宽为 D，那么 MapReduce 上本轮时间开销为

$$O_{MR} = T_c + |M|T_{queue} + \frac{S_v+S_g}{B} + T_{start} + \frac{2(S_v+S_g)}{D} \quad (2)$$

假设连接顶点数据与图结构数据以及连接顶点数据与顶点更新数据的时间分别为T_{ve}和T_{vv}，那么 DAG 模型的每轮的时间开销为：

$$O_{DAG} = T_c + |M|T_{queue} + \frac{S_v}{B} + T_{ve} + T_{vv} \quad (3)$$

综上所述，在所有三种底层框架编程模型中，采用顶点编程的时间开销最少，然后是 DAG 和 MapReduce。但是，由于后两类模型的框架支持更通用的计算，在此基础上实现图计算模型，可以将图算法作业与其它类型的作业连接起来依次执行，如对图按某种规则后筛选出子图再执行 PageRank 操作，从而可以获得更高的表达能力。此外，由于 DAG 模型的中间数据管理机制和数据集是只读的，使用 DAG 模型多个图计算作业可以共享图结构等数据，从而降低同时执行多个操作时的内存开销。

4.4 同步策略

图算法要求框架以一定顺序对图顶点进行更新并且同时更新多个顶点需要一定的同步策略来保证结果的一致性。不同的同步策略，即同步和异步更新将有不同的时间开销。使用同步策略，顶点的更新按锁步进行，即所有顶点更新一次后所有任务会进行一次同步，然后同时开始下一次所有顶点的更新。而对于异步策略，每个任务维护一个待更新顶点的调度队列，当一个顶点收到消息时，该顶点就被加入到调度队列中。与此同时，每个任务不停的从调度队列中弹出顶点进行更新。由于使用异步策略可能导致读写冲突，如两个相邻顶点可能同时根据自己的当前值更新对方的值，因此异步策略需要使用保护单个顶点数据的细粒度的锁保护顶点数据，以保证数据的一致性。

两种策略在支持框架范围、支持算法和执行效率上有所区别。

首先，与异步策略相比，同步策略可使用的图数据分布策略和编程模型更加广泛。采用 MapReduce 和 DAG 模型的框架仅能支持同步策略，因为模型要求各个作业或数据转换顺序进行，因此图算法中每轮必须待上轮结束后才能开始。而采用顶点编程模型和 SR 方式时，虽然异步策略是可实现的，但是支持的算法极为有限，因为 SR 方式中一个顶点在更新时只能访问它自上一次更新后收到的消息数据，而像 PageRank 等算法需要所有相邻顶点的数据才能够进行更新。与之相反，异步 GAS 方式则没有类似问题。因此，只有采用顶点编程模型与 GAS 方式的框架才能完全支持异步策略。

其次，同步策略比异步策略支持更多的算法。异步策略仅能支持那些不需要全局同步状态的算法。例如，MST 算法的实现被分为若干阶段，是否进入下一阶段取决于所有顶点值的聚合结果。因此，如果使用异步策略来实现 MST，待更新的顶点无法确定执行哪一阶段的更新逻辑，因此这类算法无法使用异步策略实现。

第三，对于那些可以同时使用两种策略实现的算法，异步策略可以降低总的运算量。使用同步策略时，每个顶点仅能处理上一轮收到的消息，本轮收到的新消息则需等到下一轮才能处理。而使用异步策略时，如果一个已经被加入到调度队列中的顶点再次收到消息，那么新收到的消息可以与之前收到的消息在顶点从调度队列中弹出时处理，从而减少了一次顶点更新。因此，与同步策略相比，异步策略可以合并多次更新，从而降低计算量。

尽量异步策略可以降低工作量，但是由于存在细粒度的锁冲突，它的执行时间不一定比同步框架更短。同步执行框架可以通过将存在冲突的数据读写操作安排到不同锁步执行的方式来避免细粒度的锁，如在 GAS 方式中，Gather 过程和 Scatter 过程只能读取数据而 Apply 过程只能更新所在顶点的数据，从而避免了单个锁步内部的数据读写冲突。但是使用异步策略时，由于顶点可能访问其它顶点正在更新的数据，因此必须使用锁来进行同步以保持顺序一致性的语义，否则算法多次执行的结果可能会不一致。细粒度的锁冲突将降低并行度，延长算法执行时间，因此即使异步策略具有较少的工作量，其总执行时间不一定能够更短。

5. 不同系统的比较

表 2 总结了常用的开源图计算框架以及大规模图数据处理中的四个关键问题。MapReduce[11]和 Spark[4]是采用 MapReduce 和 DAG 模型的通用计算框架，直接在这两个框架上实现算法，需要进行图模型的重复实现以将图数据的操作映射到相应的通用编程接口。Giraph[6]和 GPS[12]是 Pregel[3]的开源实现。Giraph 提供了标准的基于顶点编程模型的同步 SR 方式实现，并且支持基于磁盘的计算，GPS 在 Pregel 的基础上改进了动态图分割等方面。PowerGraph[5]与它早期的版本 GraphLab[9]，引入

[6] https://giraph.apache.org/

了 GAS 方式和顶点分割策略，同时实现了同步和异步的策略，但仅能基于内存计算。GraphX[13]在 Spark 上实现了 GAS 编程模型，由于底层框架的限制，GraphX 仅实现了同步策略。最后，GraphChi[7]是一个专门设计的运行在单个节点上的基于磁盘的图计算框架。

表 2. 开源图计算框架的列表以及它们在四个设计问题上的设计选择

框架	图数据组织策略/消息模型	磁盘访问策略	编程模型	同步策略
MapReduce	both	预先排序	MR	Sync.
Spark	both	预先排序	DAG	Sync.
Giraph	SR	多个文件	VP	Sync.
GPS	SR	-	VP	Sync.
PowerGraph	GAS	-	VP	Both
GraphX	GAS	预先排序	DAG	Sync.
GraphChi	GAS	预先分配	VP	Sync.

6. 实验

本节在多个实验数据集上验证了本文在四个关键问题上的分析总结。本节使用的数据集如表 3 所示[7]。由于所使用的部分测试算法的输入为无向图，我们将所有的有向图通过添加反向边的方式生成对应的无向图。此外，由于原始图均无权重信息而部分算法需要权重信息，我们为每条边赋予一个 1 到 50000 之间的随机数作为权重。其中，实验 6.1 至 6.4 在 16 台虚拟机上进行。每一个虚拟机配备了 4 个双核 AMDopteron™6132HE 处理器（800MHZ）和 8GB 内存，所有节点通过千兆网络连接，操作系统为 Red Hat6.2(Linux kernel 2.6.32)。对于每一个框架，我们使用一台机器作为主节点，其它节点作为从节点。

表 3. 实验所使用的数据集

编号	名称	顶点数	有向图边数	无向图边数
1	eswiki-2013	972 933	23 041 488	42 369 862
2	frwiki-2013	1 352 053	34 378 431	62 074 604
3	ljournal-2008	5 363 260	79 023 142	99 028 542
4	indochina-2004	7 414 866	194 109 311	301 969 638
5	hollywood-2011	2 180 759	228 985 632	228 985 632
6	twitter-2010	41 652 230	1 468 365 182	2 405 026 092
7	arabic-2005	22 744 080	639 999 458	1 107 806 146
8	uk-2006-05	77 741 046	2 965 197 340	5 271 699 862

[7]本文所用的所有数据集均下载自law.di.unimi.it/datasets.php，由 WebGraph[34]和LLP[35]项目支持。

6.1 SR 方式与 GAS 方式

本节通过实验比较 SR 方式和 GAS 方式。两类模型的代表框架分别是 Giraph（1.0.0）和 PowerGraph（2.2）。另外，本节使用的比较算法为 PageRank，SSSP 和 MST 算法。如 4.1 节所述，基于 PowerGraph 实现 PageRank 时 Scatter 过程不需要发送消息，而 SSSP 需要发送消息。在 Giraph 上 SSSP 和 PageRank 的实现与[3]中相同。MST 算法在 Giraph 上的实现如[32]所示，在 PowerGraph 上的实现是将确定主顶点和同步主顶点的操作转化为洪泛算法。我们将两个框架的任务数均设定为 16，因为 PowerGraph 要求每台机器仅启动一个进程。

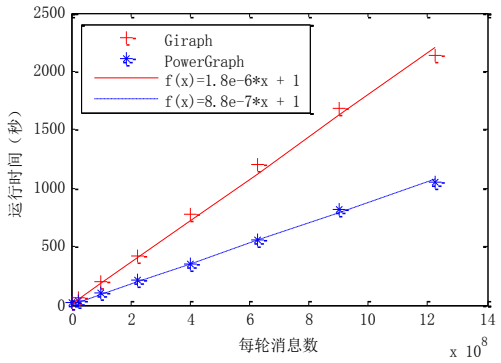


图 1. 在 PowerGraph 和 Giraph 上基准评测程序的运行时间与每轮消息数的关系。该时间在两个框架上均线性增长。

通过直接比较两个框架上同一算法的运行时间来比较不同模型是不准确的，因为运行时间受实现细节影响很大。例如，Giraph 使用 Java 编写而 PowerGraph 使用 C++编写，Giraph 使用 Netty 进行消息通信而 PowerGraph 使用 MPI 等等。为了估计和消除这些实现细节对模型比较带来的影响，我们首先在两个框架上运行一个基准评测算法。该基准评测算法运行 20 轮，每轮每个顶点需要计算所有相邻顶点的编号之和。由于基准评测算法在 Giraph 上每一轮中沿每条边发送一条消息，而在 PowerGraph 上每一轮对于每个从副本发送 3 条消息，为了使两个框架每轮的计算量和发送的消息数目相同，我们对两种框架使用不同的图作为输入。对于 Giraph，我们采用完全图，而对于 PowerGraph，我们采用为顶点i添加两条分别指向顶点$i+1$和$i+2$的边的图。假设 Giraph 中每个图有v个顶点，则该图有$v(v-1)$条边，相对应的 PowerGraph 的输入图有$v(v-1)/3(r-1)$个顶点，其中r是 PowerGraph 中每个顶点的副本个数。使用这种设计，两个框架每轮均需要发送$v(v-1)$消息，并进行$v(v-2)$次加法计算。

基准评测程序在两个框架上的运行时间如表 4

和图 1 所示。它们说明了进行相同操作，PowerGraph 比 Giraph 更快。当图的规模超过一定量的时候，两个框架的运行时间均随着每轮消息数量线性增长。通过线性拟合时间与消息数量的关系可得，PowerGraph 的增长系数为 8.8e-7，而 Giraph 的增长系统为 1.8e6，这说明执行相同操作 PowerGraph 仅需使用 Giraph 48%的时间。对于较小的图，这一比例有所增加，说明 PowerGraph 比 Giraph 的静态开销要大。根据上述结论，我们为每一个测试数据集定义了两类框架的基准对比系数：为每一个测试数据集生成相对应的基准测试程序输入图并保证基准测试程序每轮消息数目与测试数据集边数相同，定义基准测试程序在 PowerGraph 与 Giraph 上的运行时间之比作为基准对比系数。对于运行在该测试数据集上的实际算法，如果在 PowerGraph 上和 Giraph 上的运行时间的比值比该值大，则说明 SR 方式更优，反之则说明 GAS 方式更优。

表 4. 两种框架上基准评测算法的运行时间。

完全图顶点数	每轮消息数	Giraph 时间（s）	PowerGraph 时间（s）
1000	999000	17.1	7.5
2000	3998000	23.1	17
5000	24995000	64	40.3
10000	99990000	195.8	107.5
15000	2.25E+08	424.9	212
20000	4E+08	777.7	355
25000	6.25E+08	1196.7	558
30000	9E+08	1679.8	816
35000	1.22E+09	2137.9	1055

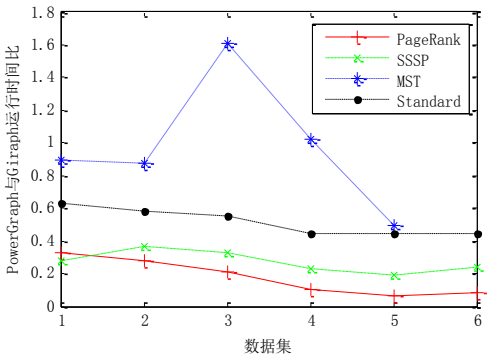


图 2. SSSP，PageRank 和 MST 算法在 PowerGraph 和 Giraph 上的运行时间比。

三个对比算法在所有数据集上的运行时间如表 5 和图 2 所示。图 2 展示了三个算法在所有数据集上的运行时间比值以及它们与基准对比系数的相对大小。从图中可以看出，运行 PageRank 和 SSSP 算法时 GAS 方式更优，考虑到基准对比系数所代表的实现细节的影响，在所有数据集上 GAS 方式的开销仅约为 SR 方式开销的一半。而运行 MST 算法时，SR 方式更优，在数据集 ljournal-2008 上，GAS 的开销甚至达到了 SR 方式的 3 倍。

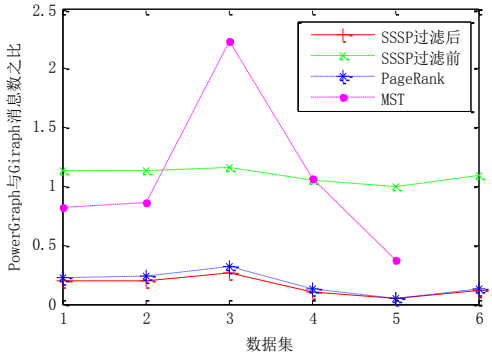


图 3. SSSP，PageRank 和 MST 算法在 PowerGraph 和 Giraph 上的发送消息数量比。对于 SSSP，在 PowerGraph 过滤前后的比例均已画出。

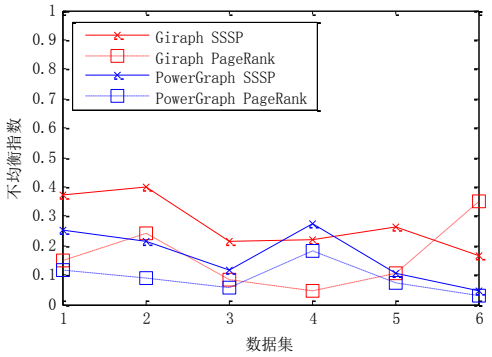


图 4. SSSP 算法和 PageRank 算法在两个框架上的不均衡度。同一算法在两个框架上实际差距不大。

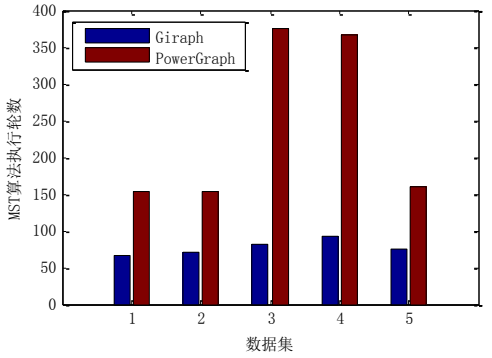


图 5. 在 PowerGraph 和 Giraph 上运行 MST 算法收敛所需的轮数

对于 PageRank 算法，GAS 方式的高效来自于更少的消息数量以及更均衡的计算负载。图 3 显示了 PowerGraph 和 Giraph 的总消息数的比值。从图中可以看出，运行 PageRank 算法时，PowerGraph 的消息数仅为 Giraph 消息数的 1/4 左右。此外，图 4 展示了两个框架的不均衡度指数，这一指数是通过计算每一轮消息数的标准差之和与总消息数的比值得到的。从图中可以看出，运行 PageRank 算法

时 PowerGraph 比 Giraph 各任务的负载更加均匀，从而进一步提高了 GAS 方式的效率。但是，从图中可以看出不均衡度的差异并不大。与消息数量因素相比，负载均衡所起的作用比较小，两种模型的差异几乎全部由消息数量决定。

对于 SSSP，图 3 显示如果不根据接收顶点的当前值进行过滤，PowerGraph 所发送的消息数目高于 Giraph。这是由于 PowerGraph 同时需要发送副本同步消息和到相邻顶点的消息而 Giraph 仅需要向相邻顶点发送消息。然而，由于 PowerGraph 可以根据目标顶点的当前数据过滤掉大部分消息，其实际发送的消息数量仅为 Giraph 的 1/4 左右。另一方面， PowerGraph 在 SSSP 算法上比 Giraph 更加均匀。因此，在 SSSP 算法上 GAS 算法仍然更优。

对于 MST 算法，SR 方式比 GAS 方式更优。这是因为 MST 算法在 SR 方式上的实现比 GAS 方式上的实现拥有更低的时间复杂度。图 5 展示了在两种模型上运行 MST 算法收敛所需要的迭代轮数，从图中可以看出，GAS 方式的算法收敛所需轮数大于 SR 方式上所需的轮数。

从以上三个算法的测试结果可以看出，GAS 方式更适用于邻域算法，而 SR 方式更适用于非邻域算法。

表 5. 两个框架运行三种算法的时间开销

		Giraph			PowerGraph		
		SSSP	PageRank	MST	SSSP	PageRank	MST
Time(s)	1	65.4	66.6	95	18.3	21.7	84.7
	2	57.4	92.4	122.8	20.9	25.8	107
	3	176.3	206.1	221	57.3	44.1	356
	4	392.3	455	370	89	44.9	377
	5	349.3	530.9	303	65	31.4	149
	6	2073.2	3372	-	490	296	-
Messages	1	1.70E+08	4.60E+08	2.30E+08	5.00E+07	1.60E+08	2.90E+08
	2	2.50E+08	6.90E+08	3.40E+08	5.00E+07	1.60E+08	2.90E+08
	3	9.20E+08	1.60E+09	5.80E+08	2.50E+08	5.00E+08	1.30E+09
	4	2.70E+09	3.90E+09	1.20E+09	2.80E+08	4.80E+08	1.20E+09
	5	2.70E+09	4.60E+09	1.20E+09	1.40E+08	2.30E+08	4.70E+08
	6	1.50E+10	2.90E+10	-	1.70E+09	3.70E+09	-

6.2 基于磁盘的计算

本节评估三种基于磁盘的计算框架 MapReduce（2.2），Giraph（1.0.0）和 GraphChi（2014 年 7 月的 Master 分支）在所有的 8 个数据集上的性能。为了模拟内存不足的情况，我们限制 MapReduce 和 Giraph 每个任务以及 GraphChi 执行进程的可占用的最大内存为 4G。对于 MapReduce 和 Giraph，我们为每个作业启动了 16 个任务。另外，对于 MapReduce，我们设置排序缓冲区的大小为 512M。对于 Giraph，我们设置内存消息缓存至少存储 2e6 条消息。在 Giraph 中，当缓冲区满时 Giraph 将产生一个溢出文件。对于 GraphChi，我们为每一个数据集使用默认的数据分片数量[7]。我们使用 PageRank 算法评估三个框架，在每一轮迭代中，PageRank 将沿每条边向相邻顶点发送一条消息。

在三个框架上运行一轮 PageRank 的时间如图 6 所示。从图中可以看出，MapReduce 运行最慢，但是其运行时间随消息数量线性增长。这是由于 MapReduce 使用预先排序中间数据的策略来保证对文件的读写均是顺序的。这种方式最小化了内存占用，但是增加了运行时间开销，因此 MapReduce 可具有的扩展性，但其绝对运行时间较长。

GraphChi 的运行时间也随着消息数量线性增长。这是由于 GraphChi 预先分配了消息的存储空间，每一条消息每轮仅需要一次读写并且读写时间均为常量。另外，GraphChi 随消息数量增长的斜率比 MpaReduce 要低，这是由于和 MapReduce 相比，GraphChi 上每条消息仅需要一次读写，而 MapReduce 中每条消息还需要经过排序以及网络传输，因此 MpaReduce 的开销更大。

对于 Giraph，我们画出了采用两种不同消息缓存大小时运行时间的增长情况。如图 6 所示，使用较大消息缓存的 Giraph 在大部分数据集上比 GraphChi 和 MapReduce 运行得更快。这是由于 Giraph 比

MapReduce 使用更多的内存并通过在内存中建立各类数据结构来加快执行；另外，Giraph 比基于单个节点的 GraphChi 拥有更高的并行度，可以同时处理更多顶点。但是，无论消息缓存大小如何，Giraph 的执行时间随着消息数量的增加快速增长，在数据集 8（twitter-2010）上，$Giraph_1$ 的运行时间甚至超过了 MapReduce。这说明同时打开多个文件进行读取的策略随着消息数量增长的可扩展性很差。这是由于同时读取多个打开文件将导致大量的随机磁盘访问。$Giraph_1$ 运行在数据集 8（twitter-2010）时，甚至出现了多线程的情况下，尽管某个顶点仅收到了一条消息，但是其需要等待 8 秒才能完成加载的情况。

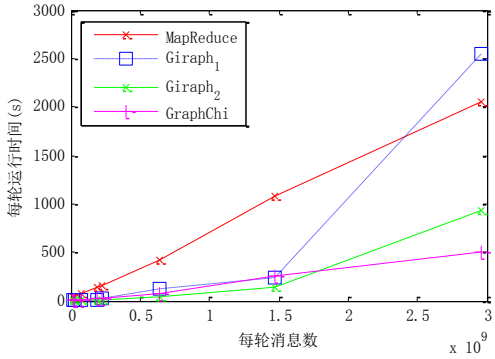


图 6. PageRank 在三个框架上每轮运行时间。Giraph1 和 Giraph2 所采用的内存缓存大小分别为 2e6 和 1.6e7，其中 1.6e7 为 4G 内存限制下可采用的最大缓存大小。

如图 7 所示，Giraph 的运行时间随着缓存大小的增加而快速下降，当缓存大小增加一倍时，运行时间约下降 30%。这进一步说明 Giraph 的运行时间随内存不足而超线性增长。因此，对于小的数据集，同时打开多个文件可以降低延迟，但是对于较大的数据集，预先归并的方式反而更加高效。

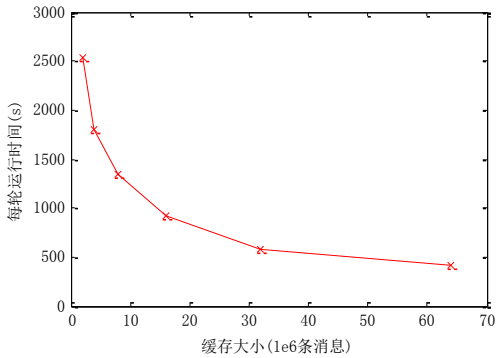


图 7. 在 Giraph 上运行 PageRank 每轮运行时间随内存缓存大小变化情况。

6.3 迭代编程模型

本节比较不同编程模型对于计算开销的影响。我们使用 Hadoop MapReduce（2.2），Spark（0.9.0）和 Giraph（1.0.0）分别作为 MapReduce，DAG 和顶点编程模型的代表。在这三个框架上，我们实现了基于 SR 方式的 PageRank 算法。由于我们选择了 SR 方式，在 Spark 上我们没有使用 GraphX，因为它是基于 GAS 方式的实现。在 MapReduce 和 Spark 上实现 PageRank 如算法 1 和 2 所示。

基于 SR 方式的 PageRank 算法在三个框架上的运行时间如图 8 所示。从图中可以看出，Giraph 运行最快，其次是 Spark 和 MapReduce。这与文中顶点编程模型的时间开销最小的分析一致。

表 6. 三个框架在特定子图（180 万顶点与 650 万边）上运行一轮 PageRank 各部分时间开销

	MapReduce	Spark	Giraph
接收到的消息大小	104M	16M	42M
发送消息大小	104M	46M	79M
图结构消息大小	40M	-	-
消息发送开销	34s	17s	2.5s
消息接收开销	17s	1.3s	-
外排序开销	12.5s	-	-
文件读入开销	131M/1.4s	-	-
文件输出开销	114M/5s	-	-
消息序列化开销	9s	7.5s	6s
顶点状态更新开销	13s	4.5s	
连接顶点和边	-	3.6s	-
连接顶点和顶点更新	-	3s	-
最大内存消耗	1.1G	3.5G	1.2G
作业启动	14s	-	-
本轮总时间	99s	37s	8.5s

为了进一步分析各类开销的比例，我们将分解三个框架上对一个有 1814375 个顶点和 6560047 条边的子图运行一轮 PageRank 的时间开销，如表 6 所示。如第 4.3 节所述，由于图结构参与消息通信，MapReduce 比另外两个框架有更大的通信开销，在该实验中，通信开销共占总运行时间的 36%。另外，MapReduce 共花费 8%的时间用于从磁盘读入上一轮结果和写出本轮结果。最后，MapReduce 中，还有 14%的时间用于作业启动。

Spark 的运行时间介于 MapReduce 与 Giraph 之间。与 Giraph 相比，在 Spark 上进行两次本次连接操作所用的时间占总时间的 17%。与 MapReduce 相比，Spark 降低了 IO 开销。由于 Spark 是基于内存计算的，并

且要求所有数据集可读，对数据集的少量修改也需要生成新的数据集，因此它占用的内存比其它两个框架都要大。

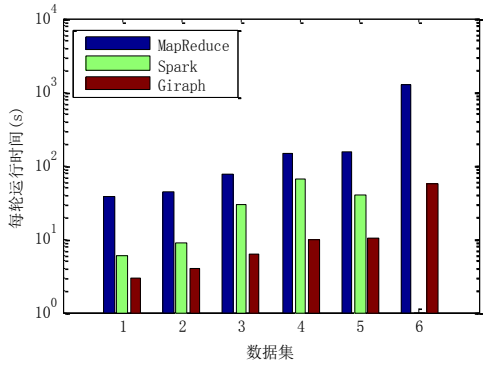


图 8. 在三个框架上运行 PageRank 每轮所需时间。其中 Giraph 运行最快。

Giraph 运行时间最短，因为它的通信量最小，并且没有额外的计算开销。另外， Giraph 中计算与消息传递的大部分时间重叠，这进一步降低了 Giraph 上算法的执行时间。从以上实验可以看出，针对单个图算法，Giraph 拥有最小的执行时间开销。

6.4 同步策略

本节通过实验比较不同的同步策略。我们使用 PowerGraph（2.2）的同步和异步引擎作为两种策略的实现。在 PowerGraph 中，同步引擎将每一轮迭代进一步分割为若干个更小的锁步，每一步中各任务同时执行所维护顶点的 Gather, Apply 或 Scatter 操作。异步引擎使用调度队列来异步更新顶点。对于每一次更新，引擎对于顶点的每一个非本地副本使用远程方法调用的方式同时进行 Scatter 或 Gather 操作。为了提高吞吐量，异步引擎使用用户级的线程并发执行多个顶点更新操作。

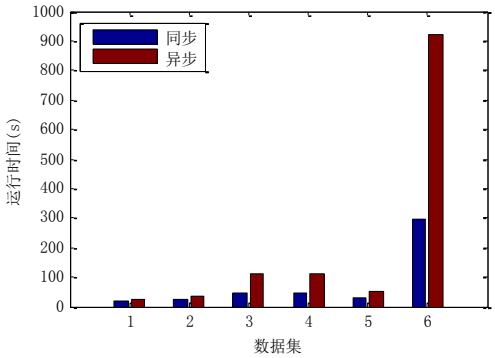


图 9. 同步和异步策略下 PageRank 的运行时间，对于 PageRank，同步策略总是优于异步策略。

我们首先测试两类策略执行 PageRank 的时间。在 PageRank 算法中，每个顶点仅仅读取相邻顶点的值而不会显式地发送消息，因此在异步引擎中不会出现向调度队列中的顶点发送消息导致顶点更新操作合并的情况，两种策略执行的顶点更新数量完全一致。从图 9 可以看出，异步策略在所有数据集上的运行时间均大于同步策略。这说明使用异步策略，执行相同的操作的开销更大。

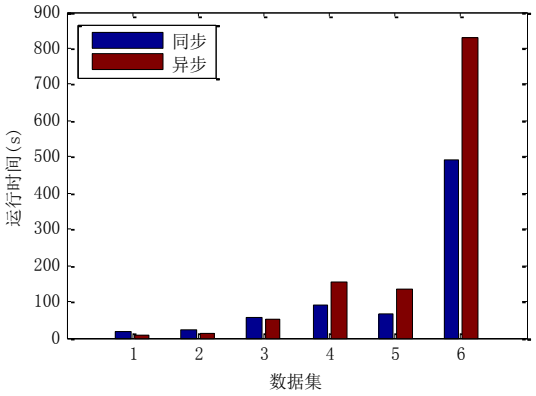


图 10. 同步和异步策略下 SSSP 的运行时间，两种策略均在部分数据集上更优。

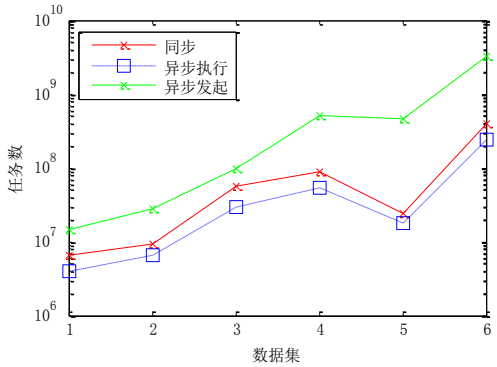


图 11. 同步和异步执行 SSSP 时发起和执行的任务数。

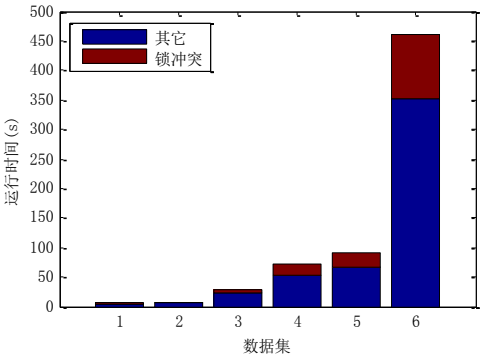


图 12. 同步执行 SSSP 时锁冲突所占运行时间比例。其中锁冲突的时间不能忽略。

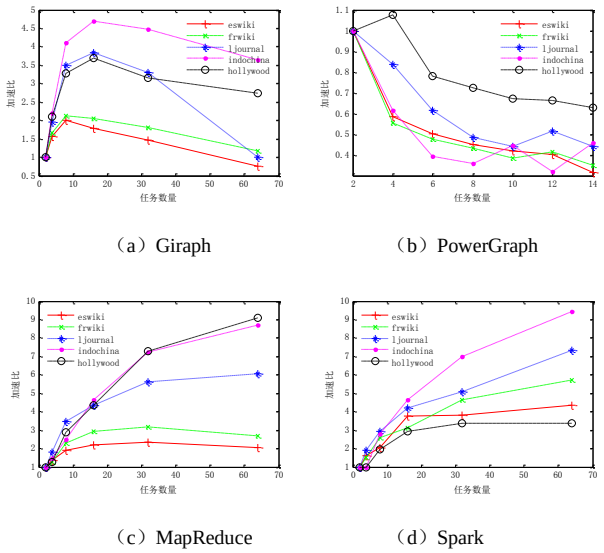
然后，我们测试两类策略执行 SSSP 算法的时间。SSSP 算法中可能会出现某一顶点多个相邻顶点同时发送消息从而导致更新合并的情况。图 11 展示了两类策略在各个数据集上的总运行时间。从图中可以看出，异步引擎在一些数据集上运行得更快，但在另一些数据集上则运行得更慢，对于最为稠密（平均每个顶点边数 105，远大于其它数据集）的 hollywood-2011，异步引擎所花时间甚至达到了同步引擎的 2 倍。

为了进一步分析原因，我们在图 11 中展示了两类策略所进行的顶点更新次数。如图所示，尽管异步

策略发起了更多的更新，但是部分更新被合并，实际执行的更新数量仅为同步策略的 65~72%。这说明使用异步策略可以通过更新合并降低总的计算量。在图 12 中，我们画出了异步策略中获取锁的时间占总执行时间的比例。获取锁的时间占总执行时间的 19%～26%，这一比例不可忽略。通过以上实验，可以看出尽管异步策略可以降低总计算量，但是由于细粒度的锁的影响，总的时间开销反而可能更长。

6.5 可扩展性

本节通过实验，对不同系统的随可用资源增加的水平扩展能力进行测试。我们选择 PowerGraph（2.2），Giraph（1.0.0），MapReduce(2.2)以及 Spark（0.90）作为各类框架的代表。所有框架均采用基于内存的计算策略。此外，我们选择基于 SR 策略的 PageRank 算法作为评测算法。可扩展性的实验在 14 台物理机器上进行，其中 5 台机器拥有 32 个 AMDopteron™6132HE 处理器（800MHZ）和 64GB 内存，其余 9 台机器拥有 48 个 AMDopteron™6337HE 处理器（1GHZ）和 32GB 内存。为了验证水平扩展能力，根据框架自身的资源分配与调度策略，对于 MapReducet、Giraph 和 Spark，我们采用增加任务数的方式为框架增加资源，每个任务数占用相同的资源。对于 PowerGraph，我们采用增加机器数量的方式为其增加资源。实验采用前 5 个数据集，这是由于采用更大的数据集时各框架在大部分任务数下执行失败。



（a）Giraph （b）PowerGraph

（c）MapReduce （d）Spark

图 13. 不同大小数据集上各框架每轮 PageRank 计算随机器数量增加的加速比。

我们首先测试了对于固定的数据集，每个框架之上 PageRank 每轮随资源增加的加速比变化情况，如图 13 所示。如图所示，随着资源数量的增加，MapReduce 与 Spark 的加速比增大到一定程度后保持不变。这时由于在水平扩展的情况下，资源的增加是通过任务数的增加来实现的。虽然增加资源会降低计算所花的时间，但是任务数的增加将导致非本地消息通信的增加。根据文献[5]，在采用随机图分割的情况下，p 个任务时非本地消息的占全部消息数量的 $1-1/p$。因此，计算的时间的减少与通信时间的增加相互抵消，导致加速比难以随资源进一步增加。与 MapReduce 和 Spark 不同，Giraph 与 PowerGraph 的加速比在增大到极值后反而会降低，甚至出现资源增加导致运行更慢的情况（如图 13(b)）所示。这是由于如 4.3 节所述，Giraph 与 PowerGraph 这类基于顶点编程模型的框架同时进行顶点状态更新与消息发送，因此，它们之上 PageRank 每一轮的时间取决于两者间的最大值而非总和。当任务数超过阈值时，通信时间成为每轮时间的瓶颈并随任务数增加而不断增加，从而导致运行得更慢。

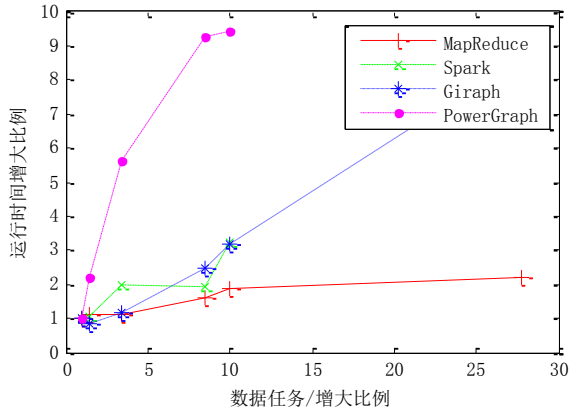


图 14. 不同框架按比例同时增大数据量和机器数量时的 PageRank 每轮运行时间变化情况

然后，我们对各个框架同时增加工作负载和资源的情况下每轮运行时间的变化情况进行测量。我们使用任务数为 2 时数据集 1（eswiki）的运行结果作为基准测试。对于其它数据集，如 4.1 节所述，我们使用图的边数作为工作负载的指标。对于每个数据集，我们根据它与 eswiki 数据集边数的比例作为工作负载和任务数增加的比例，并对 PageRank 每轮运行时间的变化情况进行测量。如果框架可扩展性较好，那么它可以充分利用增加的资源处理额外的负载，从而保证运行时间基本不变。实验结果如图 14 所示。从图中可以看出，数据量增加时随着数据量和资源几乎按比例增加，而非保持恒定。这说明各个框架未充分利用增加的资源并行处理额外的负载。其中，绝对时间最短的 PowerGraph 和 Giraph 增长得也最快，说明它们的可扩展性弱于 MapReduce 与 Spark。

通过以上实验可以看出，现在框架在水平扩展性能上存在不足，并未能够充分利用增加的资源高效得

处理更大的负载。因此，未来框架设计需要对可扩展性进行改进。

7. 结论

本文全面分析和总结了大规模图数据处理的四个关键问题，之后结合主流的大规模图处理框架，建立了评估模型定量地分析这些关键问题对大规模图数据处理的影响。最后在多个大规模数据集上评测了本文提出的评估模型的有效性。在我们的测试结果中发现了如下与当前主流观点不一致的现象：首先，与图数据边分割相比，通常认为更快的顶点分割方法（如 PowerGraph）虽然在邻域算法上运行时间能够达到边分割的 50%左右，但是在非邻域算法上时间开销却是边分割的 3 倍；其次， 与同步策略相比，异步策略可以减少约 20%～30%的总计算量，但由于细粒度的锁冲突，其运行时间反而可能达到同步策略的 2 倍；最后，当数据集达到 4 千万顶点和 13 亿条边时，基于磁盘的 MapReduce 比基于内存的 Giraph 等框架性能反而更高。

为进一步完善本文的评估模型，未来还需要在大规模图数据处理的容错和图分割算法等方面，深入开展研究工作。

参 考 文 献

[1] C. C. Aggarwal, H. Wang. Managing and mining graph data. Springer US. 2010

[2] C. C. Aggarwal, H. Wang. An introduction to graph data. Springer US. 2010

[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel : A system for large-scale graph processing. in Proceedings of the the 2010 ACM SIGMOD. 2010: 135-145

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. in Proceedings of the the 9th OSDI. 2012:

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. in Proceedings of the the 10th OSDI. 2012:

[6] B. Shao, H. Wang, Y. Li. The trinity graph engine. 2012

[7] A. Kyrola, G. E. Blelloch, C. Guestrin. Graphchi: Large-scale graph computation on just a pc. in Proceedings of the OSDI. 2012: 31-46

[8] B. Shao, H. Wang, Y. Li. Trinity: A distributed graph engine on a memory cloud. in Proceedings of the the 2013 international conference on Management of data. 2013: 505-516

[9] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. the VLDB Endowment, 2012, 5(8): 716-727

[10] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. in Proceedings of the Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. 2010: 721-726

[11] J. Dean, S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107-113

[12] S. Salihoglu, J. Widom. Gps: A graph processing system. in Proceedings of the the 25th International Conference on Scientific and Statistical Database Management. 2013: 22-32

[13] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica. Graphx: A resilient distributed graph system on spark. in Proceedings of the First International Workshop on Graph Data Management Experiences and Systems. 2013: 2

[14] H. Karloff, S. Suri, S. Vassilvitskii. A model of computation for mapreduce. in Proceedings of the the 21th SODA. 2010:

[15] M. T. Goodrich, N. Sitchinava, Q. Zhang. Sorting, searching, and simulation in the mapreduce framework. Algorithms and Computation, 2011, 374-383

[16] B. Elser, A. Montresor. An evaluation study of bigdata frameworks for graph processing. in Proceedings of the IEEE International Conference on Big Data. 2013: 60-67

[17] H. L. Lei Gu. Memory or time: Performance evaluation for iterative operation on hadoop and spark. in Proceedings of the the 15th HPCC. 2013:

[18] 于戈, 谷峪, 鲍玉斌, 王志刚. 云计算环境下的大规模图数据处理技术. 计算机学报, 2011, 34(10): 1753-1767

[19] 应毅, 刘亚军. Mapreduce 并行计算技术发展综述. 计算机系统应用, 2014, 23(4): 1-6, 11

[20] E. Reghbati, D. G. Corneil. Parallel computations in graph theory. SIAM Journal on Computing, 1978, 7(2): 230-237

[21] P. Mateti, N. Deo. Parallel algorithms for the single source shortest path problem. Computing, 1982, 29(1): 31-49

[22] S. Brin, L. Page. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 1998, 30(1-7): 107-117

[23] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. in Proceedings of the the eighth annual ACM symposium on Theory of computing. 1976: 55-57

[24] Y. Saad. Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics. 2003

[25] S. G. Akl. An adaptive and cost-optimal parallel algorithm for minimum spanning trees. Computing, 1986, 36(3): 271-277

[26] G. Karypis, V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. in Proceedings of the PPSC. 1997: 1-11

[27] V. Satuluri, S. Parthasarathy, Y. Ruan. Local graph sparsification for scalable clustering. in Proceedings of the Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. 2011: 721-732

[28] I. Stanton, G. Kliot. Streaming graph partitioning for large distributed graphs. in Proceedings of the Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining. 2012: 1222-1230

[29] A. Abou-Rjeili, G. Karypis. Multilevel algorithms for partitioning power-law graphs. in Proceedings of the IPDPS. 2006: 1-10

[30] G. Karypis, V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing, 1998, 20(1): 359-392

[31] Y. Low, C. Guestrin, A. Y. Ng. Graphlab : A distributed abstraction for large scale

machine learning. 2013

[32] S. Salihoglu, J. Widom. Optimizing graph algorithms on pregel-like systems. PVLDB, 2014, 7(1): 577-588

[33] S. Chung, A. Condon. Parallel implementation of borvka's minimum spanning tree algorithm. in Proceedings of the International Parallel and Distributed Processing Symposium. 1996: 302-308

[34] P. Boldi, S. Vigna. The webgraph framework: Compression techniques. in Proceedings of the the Thirteenth International World Wide Web Conference. Manhattan, USA, 2004: 595-601

[35] P. Boldi, M. Rosa, M. Santini, S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. in Proceedings of the the 20th international conference on World Wide Web. 2011: 587-596