软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(3):528−544 [doi: 10.13328/j.cnki.jos.005441]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

流式处理的异步图处理框架*

李金吉[1], 张岩峰[1], 巩树凤[1], 于 戈[1], 高立新[2]

[1](东北大学 计算机科学与工程学院,辽宁 沈阳 110819)

[2](Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, USA)

通讯作者: 张岩峰, E-mail: zhangyf@mail.neu.edu.cn

摘 要: 分布式图计算是目前处理大图数据的主流技术,但是存在诸多无法避免的问题,比如分布式计算的负载均衡和分布式实现的调试和优化仍然非常困难.另一方面,近几年的研究结果表明:通过设计合理的数据结构和处理模型,在单个 PC 上基于大容量磁盘的大图计算往往可以获得与分布式图计算相当的处理性能.例如,GraphChi 在单机上的处理性能与 Spark 在 50 台节点上的处理性能相差无几.结合累加迭代计算和单机并行处理技术,提出流式处理的异步计算模型 ASP.它实现了对磁盘的完全顺序访问,允许流式的顺序载入结构数据的同时进行异步更新计算.基于 ASP 模型,提出了一种流式处理的异步图处理框架 S-Maiter,实现了高效率的基于外存的单机大图处理,通过 I/O 线程优化、内存资源监控、shard 级优先级调度等优化技术,提高了系统处理大图数据的性能.实验结果表明:在处理大图数据(1 300 万顶点,5 亿连边)时,仅仅需要 1 台 PC 机计算资源的 S-Maiter 与在 16 台 PC 上运行的分布式 Maiter 的性能几乎相当.并且,S-Maiter 比另外一个流行的单机大图处理系统 GraphChi 要快 1.5 倍.

关键词: 外存;异步累加模型;I/O;流式处理

中图法分类号: TP311

中文引用格式: 李金吉,张岩峰,巩树凤,于戈,高立新.流式处理的异步图处理框架.软件学报,2018,29(3):528−544. http://www.jos.org.cn/1000-9825/5441.htm

英文引用格式: Li JJ, Zhang YF, Gong SF, Yu G, Gao LX. Streamlined asynchronous graph processing framework. Ruan Jian Xue Bao/Journal of Software, 2018,29(3):528−544 (in Chinese). http://www.jos.org.cn/1000-9825/5441.htm

Streamlined Asynchronous Graph Processing Framework

LI Jin-Ji[1], ZHANG Yan-Feng[1], GONG Shu-Feng[1], YU Ge[1], GAO Li-Xin[2]

[1](College of Computer Science and Engineering, Northeastern University, Shenyang 110819, China)

[2](Department of Electrical and Computer Engineering, University of Massachusetts Amherst, Amherst, USA)

Abstract: Distributed graph processing is mainstream but suffers from a few unavoidable issues, such as workload imbalancing and the debugging/optimizing difficulties in distributed programs. On the other hand, recent research results show that with a reasonable design of data structure and processing model, graph processing on a single PC can achieve comparable performance as the systems using large number of machines. For example, GraphChi on a single PC can achieve almost the same performance with Spark with 50 nodes. In this paper, a streamlined asynchronous graph processing model, ASP is proposed based on accumulated iterative model and external storage based parallel computing techniques. ASP relies on sequential disk access and allows asynchronous computations on the graph structure

* 基金项目: 国家自然科学基金(61672141, 61528203); 计算机体系结构国家重点实验室开放课题(CARCH201610); 中央高校基本科研业务费专项资金(N161604008)

Foundation item: National Natural Science Foundation of China (61672141, 61528203); State Key Laboratory of Computer Architecture, CAS (CARCH201610); Fundamental Research Funds for the Central Universities (N161604008)

本文由基于图结构的大数据分析与管理技术专刊特约编辑林学民教授、杜小勇教授、李翠平教授推荐.

收稿时间: 2017-07-31; 修改时间: 2017-09-05; 采用时间: 2017-11-07; jos 在线出版时间: 2017-12-05

CNKI 网络优先出版: 2017-12-06 15:23:14, http://kns.cnki.net/kcms/detail/11.2560.TP.20171206.1522.006.html

data. Based on ASP, a streamlined graph processing framework, S-Maiter is designed and implemented to provide high performance graph processing ability on a single PC. By optimizing I/O threading, memory monitoring, and shard-level priority scheduling, the performance of S-Maiter is greatly improved. Experimental results on a big graph dataset (13 million nodes and 500 million edges) show that, 1-node S-Maiter can achieve comparable performance with distributed Maiter with 16 nodes. Furthermore, S-Maiter is 1.5 times faster than the popular single-PC graph processing system GraphChi.

Key words: external storage; asynchronous accumulated model; I/O; streamlined processing

近年来,图数据分析成为学术界和工业界的热点研究问题,一系列针对大规模图数据的计算框架与处理技术也应运而生[1−14],其中比较有代表性的图计算框架有 Maiter[4]、Pregel[5]、GraphLab[6]、GraphChi[7]、PowerGraph[15]等,这些框架使用以顶点为中心的计算模型,负责图数据的分割、计算、容错等,用户只需要通过实现简单的编程接口,便可完成各种复杂的大规模图数据分析任务.这些图计算框架的出现提高了分析大规模图数据的能力.

现有的分布式图计算框架,如 Maiter[4]、Pregel[5]、GraphLab[6]、PEGASUS[8]、PowerGraph[15]、Spark[16]、Imapreduce[17]、PrIter[18]可以处理数十亿级规模的图数据,但是高效地进行大规模图计算仍然存在很多挑战.

1) 分布式图计算的首要任务是对图数据的合理分割,如何找到有效的图划分方法来减少集群中顶点之间的通信量并且保证负载均衡,是一个非常困难的问题;
2) 对于普通用户来说,获取和管理大规模集群需要昂贵的代价;
3) 在分布式集群上对算法的调试和优化也是一项困难的工作.

另一方面,一系列基于大容量磁盘的单机大图计算框架(如 GraphChi[7]、TurboGraph[12]和 X-Stream[13])在大图分析上展现出巨大的潜力,在一个包含 15 亿条边的 Twitter 图数据集上运行 PageRank 算法,Spark[16]使用 50 台机器(100 个 CPU)花费了 8.1 分钟[19],而 GraphChi[7]在一台只有 8GB RAM 和 256GB SSD 的机器上只花费了 13 分钟[14];在一个有 67 亿条边的 Web 图数据集上运行置信度传播算法,PEGASUS[8]使用 100 台机器花费了 22 分钟[8],而 GraphChi[7]在一台 PC 上只花费了 27 分钟[14].这些结果表明:基于磁盘的单机图计算框架与分布式图计算框架相比,有足够的能力在合理的时间内处理相同的问题,且成本低廉.

但是,为了保证计算结果的正确性,现有的基于磁盘的图计算框架会将中间结果同步到磁盘,以便更新结果对后续计算是可见的.如在 GraphChi[7]中,图的边数据在磁盘上被组织成若干个分片(称为 shard),GraphChi[7]在计算完一个 shard 之后,已经更新的顶点值需要传播到所有磁盘上其他的 shard;X-Stream[13]会将部分中间结果写回磁盘以便后续的处理.这种频繁更新和读取磁盘数据操作会产生很严重的后果:产生大量 IO,从而导致额外的计算成本和数据加载开销.

此外,现有的基于磁盘的图计算框架普遍采用了同步计算模式,这种同步迭代的缺点是:对任何顶点的第 k 次更新计算,必须要等到所有顶点的第 k–1 次更新计算结束才可以开始,每一次迭代都要求每一个顶点更新且只更新一次.先完成更新计算的顶点要等待未完成更新计算的顶点,在一次迭代计算中,计算未更新的顶点时并没有利用在本次迭代中产生的部分最新更新结果.更重要的是,这种同步模式限制了各执行线程的并行性,这在很大程度上限制了计算框架的处理能力,影响收敛速度.

针对以上问题,我们采用异步累加图计算的方法[4],摆脱同步计算的限制,增加计算收敛速度.但是在基于磁盘的图处理框架上应用异步累加迭代有很多挑战,异步累加迭代存在累加和优先级调度的性质,所以会产生大量的随机读写和重复访问从而影响计算性能,这就产生了两个核心问题.

1) 如何设计有效的图存储模型来避免频繁更新和读取磁盘数据所产生的大量随机 I/O 和重复 I/O;
2) 如何设计有效的计算模型来充分利用顺序 I/O 访问磁盘数据,提高计算和 I/O 的并行性.

为了解决这两个关键问题,本文首先提出了一种新的图处理模型,流式处理的异步计算(ASP),它采用顶点为中心的计算模型.在 ASP 中,结合了异步累加迭代自身的特点提出了一种图存储模型,为了解决频繁更新和读取磁盘数据导致的大量 I/O 问题,新的图存储模型将可变的顶点数据与只读的结构数据进行分离,并将可变的顶点数据有效地缓存到内存中,这样对于多个 shard 来讲,对任意顶点的更新和重复访问可以在缓存中完成,并

不需要将中间结果同步到磁盘.另外,我们的计算模型实现了对磁盘的完全顺序访问,允许流式地顺序载入计算所需要的结构数据的同时进行异步更新计算,并且尽可能地最大化利用顺序访问时磁盘的带宽,减少数据的访问次数,并有效地使用内存和 CPU 来提供性能保证.

基于 ASP 模型,本文实现了完整的基于外存的异步图处理框架 S-Maiter,提供高效的异步计算能力.用户只需要通过 S-Maiter 提供的 API 实现特定的更新函数,S-Maiter 就会在内存或者外存模式下自动部署异步累加迭代计算.

本文的主要贡献如下:

1) 提出了流式处理的异步计算方案.包括适用于异步累加迭代的图存储模型和计算模型.图存储模型解决了频繁更新和读取磁盘数据导致的大量 I/O 问题,有效避免了大量随机 I/O 和重复 I/O.计算模型实现了对磁盘数据的完全顺序访问,能并行化图数据的流式载入和更新函数的异步执行;
2) 设计并实现了基于外存的流式异步图处理框架 S-Maiter.通过 I/O 线程优化、内存资源监控、shard 级优先级调度等优化技术,极大地提高了系统处理大图数据的性能;
3) 对 S-Maiter 在多个数据集上进行了实验评估.实验结果表明:无论是在固态硬盘上还是机械硬盘上,S-Maiter 都表现出优异的性能.对于 PageRank 算法,相比较于 GraphChi[7],速度提高了 1.5 倍,与用 16 节点的分布式 Maiter[4]处理速度相当.

本文第 1 节介绍累加迭代计算的背景知识.第 2 节提出流式处理的异步计算.第 3 节详细介绍 S-Maiter 的设计与实现.第 4 节展示实验评估结果.第 5 节是相关工作.第 6 节是结论.

1 累加迭代计算

累加迭代计算[4]适用于更新函数可以表达为如下形式的图计算:

$$v_j^k = g_{\{1,j\}}(v_1^{k-1}) \oplus g_{\{2,j\}}(v_2^{k-1}) \oplus ... \oplus g_{\{n,j\}}(v_n^{k-1}) \oplus c_j \tag{1}$$

其中,v_j为图顶点 j 的状态,k 为迭代次数,c_j为常量,符号'⊕'代表抽象二元运算,函数 $g_{\{i,j\}}(x)$代表当顶点 i 状态为 x 时对顶点 j 的影响.也就是说,顶点 j 的状态更新依赖于其连入邻居顶点 i 对其的影响 $g_{\{i,j\}}(x)$,通过'⊕'操作将这些邻居顶点的影响和其初始状态聚合来更新其本身状态 v_j.例如,PageRank 算法参考连入网页 j 的网页权值 R_j 来迭代更新网页 j 的权值 R_j:

$$R_j^k = d \cdot \sum_{\{i|(i\to j)\in E\}} \frac{R_i^{k-1}}{|N(i)|} + (1-d) \tag{2}$$

其中,d 为阻尼因子(一般取值 0.85),$|N(i)|$为网页 i 的出度,$(i\to j)$为从网页 i 到网页 j 的链接,E 为所有链接集合.PageRank 的更新函数即是公式(1)的一种表达方式,其中,c_j=1−d,'⊕'为'+'.如果从网页 i 到网页 j 存在链接,$g_{\{i,j\}}(v_i^{k-1}) = d \cdot \frac{v_i^{k-1}}{|N(i)|}$;否则,$g_{\{i,j\}}(v_i^{k-1}) = 0$.

累加迭代计算基于消息模型,通过对消息的处理和交互来完成计算,每个顶点 j 执行 receive 和 update 两个操作:

$$\left.\begin{aligned} receive: &\begin{cases} \text{Whenever a value } m_j \text{ is received,} \\ \Delta v_j \leftarrow \Delta v_j \oplus m_j. \end{cases} \\ update: &\begin{cases} v_j \leftarrow v_j \oplus \Delta v_j. \\ \text{For any } h, \text{ if } g_{\{j,h\}}(\Delta v_j) \neq 0, \\ \text{send value } g_{\{j,h\}}(\Delta v_j) \text{ to } h; \\ \Delta v_j \leftarrow 0. \end{cases} \end{aligned}\right\} \tag{3}$$

其中,每个顶点的 update 操作由系统调度执行,v_j代表顶点 j 的迭代状态,Δv_j代表顶点 j 累积的变化,当调度 update 操作时,此累积的变化Δv_j将作用到顶点状态 v_j 上.另外,要让这些变化通过消息传递的方式影响 j 的所有邻居顶点 h,即:将 $g_{(j,h)}(\Delta v_j)$处理后的结果发送到所有邻居顶点 h,同时要将这些处理过的变化清零.每个顶点一旦收到

消息即触发 receive 操作,receive 操作的主要功能是将这些邻居顶点传递来的部分变化消息 m_j 累积到其自身维护的 Δv_j 上.例如:在累加迭代式的 PageRank 算法中,每一个网页 j 附带一个缓存 ΔR_j 来累加收到的 R 的变化量,当网页 j 被激活执行 update 操作时,ΔR_j 将累加到 R_j 并更新 R_j,同时发送 $d\cdot\frac{\Delta R_j}{|N(j)|}$ 给它所有链出的顶点.

简单来说,累加迭代计算是基于顶点状态的变化来做更新迭代计算,而非传统的顶点状态值.文献[4]给出了累加迭代计算的收敛性和正确性证明,证明其可以异步执行并得到与同步执行完全相同的结果.因为计算只基于变化量,可以采用某些过滤方法[4,6]降低计算量并加速图算法收敛;另外,异步计算也可以避免并行计算时的同步开销.

累加迭代计算具有两个重要性质.

1) 累加性.即,迭代更新的顶点状态是多次迭代的结果(变化值)累加起来的;
2) 异步性.即,迭代计算过程中各顶点的 update 操作可以在任意时间执行且不影响最终结果,解除同步限制,即,支持并发地异步执行.

本文将利用累加迭代计算模型的累加性和异步性,设计基于外存的单机流式异步处理模型.

2 流式处理的异步计算

对于一个图 $G=(V,E)$,V 代表顶点集,E 代表边集.每一个顶点 $v\in V$ 会有一个顶点值,给定一个从顶点 u 到顶点 v 的有向边 $e=(u,v)$,e 是 v 的入边,也是 u 的出边.u 称为 v 的入边点,v 称为 u 出边点.以顶点为中心的计算模型来进行迭代计算,每次更新顶点的值通常只涉及其入边点的值,一旦一个顶点值被更新,会将新产生的消息传播到其出边点,并更新出边点的值.这种动态更新的迭代过程直到满足收敛条件才会终止.正如许多框架[5−7]所展示的,以顶点为中心的计算模型可以表达一大类的问题.

本文提出的 ASP 基于累加迭代计算,采用以顶点为中心的计算模型,结合异步累加迭代的特点设计有效的图存储模型和图计算模型,基于磁盘有效地管理图数据,最小化磁盘数据的访问量.同时,充分利用内存和 CPU 资源进行并行更新计算,提高计算效率.

2.1 图存储模型

我们主要考虑以下 3 点设计 ASP 的存储模型.

1) 为了提高单机图计算的并行性,采用分片的图存储结构,使每个执行线程负责一个或多个分片,以便并行计算;
2) 为了综合利用高性能内存和大容量外存,考虑到真实世界图数据中顶点数远远小于边数的性质,使用内存主要存储不断迭代更新的顶点状态数据,使用外存主要存储不变的图结构数据,从而充分地利用内存的随机读写能力和外存的大容量特点;
3) 为了避免外存的随机 I/O,合理组织图结构数据,使对图结构数据的访问为顺序 I/O.

具体来说,图 G 的顶点集 V 分成 P 个不相交的区间(interval),每个 interval 关联一个 shard,shard 包含用于顶点进行更新计算所需要的信息.由于累加迭代计算的异步性,图分割方法对 ASP 性能的影响不大,因此,图分割并不是本文的研究重点.本文仅支持基于图顶点编号的哈希分割(hash partition)或者范围分割(range partition).每个 shard 包含一个 g-shard 和一个 v-shard.g-shard 包含其关联的 interval 里所有顶点的出边(包括边相关联的属性),v-shard 包含其关联的 interval 里所有顶点的顶点编号和 shard 编号.每个 g-shard 中的边是按源顶点排序,我们把这些边称为结构数据,这些结构数据存储在磁盘上连续的块中.假设把图 1 的顶点集分为 3 个 interval,$I1=[1,2]$,$I2=[3,4]$,$I3=[5,6]$,每个 interval 关联着一个 shard,包括 g-shard 和 v-shard.所有的 v-shard 会级联成一个顶点表,用于初始化顶点信息,所有的 g-shard 也会级联成一个结构数据流,用于流式更新顶点信息.

在累加迭代计算中,图数据由只读的图结构数据、不断更新的顶点值 v、顶点累积的变化量 Δv 组成.我们观察到:Δv 会参与到其相邻顶点的更新,通常会被访问多次.I/O 是基于磁盘方法的一个瓶颈,为了避免频繁更新

和读取磁盘上的 v 和Δv 而导致大量的随机 I/O 和重复 I/O,将只读的结构数据与可变的顶点值 v,Δv 分离,并将只读的结构数据组织到磁盘 g-shard 中.我们结合累加迭代计算的累加性,将所有的顶点值 v 和Δv 缓存到内存中,因为顶点值 v 和Δv 所占的空间远远小于结构数据所占空间的大小,现代计算机的内存容量完全可以满足需求.ASP 采用流式计算(第 2.2 节),结构数据在内存中所占空间是动态平衡的,并且是可控的,也说明了将顶点数据缓存到内存中是可取的.由于累加性,对顶点值 v 和Δv 的更新和重复访问可以在内存中完成,此时,更新每个 interval 时只需要对相应只读结构数据进行一次顺序的扫描,最小化对图数据访问产生的 I/O 开销.

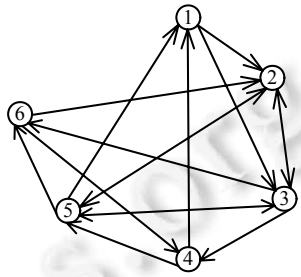

Fig.1 Example graph

图 1 示例图

2.2 图计算模型

本文基于传统累加迭代理论,提出并行环境下的图计算模型,以适用于单机大图数据的处理.总体图计算框架采用并行计算模型,每个执行线程负责一个或多个分片,每个分片包括一个 v-shard 和与之对应的 g-shard,较小的 v-shard 被加载到内存以支持频繁更新,数据量更大但是不变的图结构数据安置在磁盘上以节省内存空间.计算框架的整体示意图如图 2 所示,在迭代计算执行过程中,每个执行线程流式地从磁盘 g-shard 上顺序读取结构数据信息,并基于本地 v-shard 里面顶点的状态(v 和Δv)更新邻居顶点的顶点状态.线程间的通信是为了传递Δv.在这个模型里面存在两个主要开销:一是从磁盘读取图数据的 I/O 开销,另一个是线程间通信的开销.我们将说明,该计算模型利用累加迭代计算的性质可以极大地降低这两种开销.

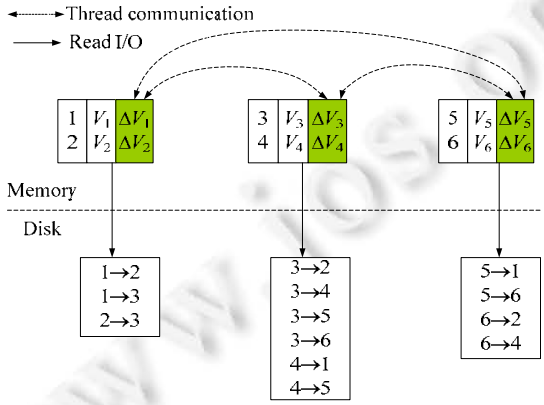

Fig.2 Schematic view of calculation model

图 2 计算模型示意图

算法 1 描述了单机图计算的核心流程,所有执行线程并行地执行该算法,这也是累加迭代计算模型在单机并行计算环境下的实现方式.首先,从磁盘 g-shard 中顺序读取任意顶点 i 的结构数据记录,根据该结构数据的源顶点编号 i 在内存 v-shard 中定位到该顶点信息的记录(v_i 和Δv_i),当顶点 i 的所有结构数据流入内存后,判断该顶点信息是否为有效的变化信息(即,判断Δv_i=0?):若为有效信息($\Delta v_i \neq 0$),首先将Δv_i 累加到顶点值 v_i 上,然后执行更新操作并将更新之后的结果作用到邻居顶点 j 上的Δv_j,并对顶点 i 的变化信息清零.当对顶点 i 的操作结束后,

立即将顶点 i 的结构数据从内存中删除,释放内存,为流入的其他未计算的顶点结构数据腾出空间.反复执行这些操作,直到算法收敛.

算法 1. Computational model.

```
1  while !terminated do
2  //read I/O
3     sequentially read outgoing edges of i
4  //update
5     if Δv_i≠0
6        v_i←v_i⊕Δv_i
7  //update thread communication
8        for any j, Δv_j←Δv_j⊕g_{i,j}(Δv_i)
9        Δv_i←0
10    endif
11    remove outgoing edges of i
12 endwhile
```

需要注意的是:一个执行线程可能会对某一顶点 j 的 Δv_j 进行写操作,同时其他执行线程可能会对同一顶点 j 的 Δv_j 进行读操作或写操作,这样会产生读写冲突,导致计算结果错误.为了避免多个执行线程对同一表项的 Δv_j 的读写冲突,对 Δv_j 的写操作(算法 1 的第 6 行~第 9 行)被实现为原子操作,对 Δv_j 的写操作必须在临界区执行.

关于该并行计算模型的正确性,通过以下定理可以保证.

定理 1(正确性). 若某图算法中顶点的迭代更新公式符合公式(1)的形式,且收敛结果为 v^*.若'⊕'操作满足交换律和结合律,且 $g(x\oplus y)=g(x)\oplus g(y)$,基于算法 1 得到的收敛结果为 v',那么 $v'=v^*$.

证明:让 Δv_i^k 代表从第 $k-1$ 次迭代结果 v_j^{k-1} 到第 k 次迭代结果 v_j^k 的变化,即 $v_j^k = v_j^{k-1} \oplus \Delta v_j^k$.

因为 $g(x\oplus y)=g(x)\oplus g(y)$,用 $v_i^{k-2} \oplus \Delta v_i^{k-1}$ 代替式(1)中的 v_i^{k-1},可以得到:

$$v_j^k = g_{\{1,j\}}(v_1^{k-2}) \oplus g_{\{1,j\}}(\Delta v_1^{k-1}) \oplus \ldots \oplus g_{\{n,j\}}(v_n^{k-2}) \oplus g_{\{n,j\}}(\Delta v_n^{k-1}) \oplus c_j \tag{4}$$

因为'⊕'满足交换律,即 $x\oplus y=y\oplus x$;且'⊕'满足结合律,即 $(x\oplus y)\oplus z=x\oplus(y\oplus z)$;我们把公式(4)中所有 $g_{\{i,j\}}(v_i^{k-2})$, $i=1,2,\ldots,n$ 和 c_j 按'⊕'运算累加起来得到 v_j^{k-1}.另外,又因为 $v_j^k = v_j^{k-1} \oplus \Delta v_j^k$, 公式(4)中所有余下 i 的 $g_{\{i,j\}}(\Delta v_i^{k-1})$ 按'⊕'运算累加将得到 Δv_j^k.所以,可以把公式(1)变换,得到如下两步的更新规则:

$$\begin{cases} \Delta v_j^{k+1} = \sum_{i=1}^{n} \oplus g_{\{i,j\}}(\Delta v_i^k) \\ v_j^{k+1} = v_j^k \oplus \Delta v_j^{k+1} \end{cases}$$

上面的更新公式是以接收顶点 j 为中心的更新公式.对应地,考虑发送顶点 i 及其任意出边 j,以发送顶点为中心的更新公式为

$$\begin{cases} v_i \leftarrow v_i \oplus \Delta v_i \\ \Delta v_j \leftarrow \Delta v_j \oplus g_{\{i,j\}}(\Delta v_i), \forall j \\ \Delta v_i \leftarrow 0 \end{cases} \tag{5}$$

由于算法 1 的更新方法由公式(1)推导而来,因此得到的收敛结果一致,即 $v'=v^*$. □

假设 1(迭代无穷性). 对于图计算中任意顶点,在任意时间 t 后的有限时间 Δt 内,即 $[t,t+\Delta t]$,算法 1 的更新操作至少在某个处理器被执行一次.即:任意顶点的算法 1 操作被执行无穷多次,直到图计算收敛.

引理 1(收缩性). 假设某时刻所有图顶点状态向量为 v,$f(v)$为作用到任意图顶点 i 的一次算法 1 操作,假设 v^*为最终收敛结果,$\|f(v)-v^*\|_\infty \leqslant \partial \|v-v^*\|_\infty$,其中,$0\leqslant\partial\leqslant 1$,$\|\cdot\|_\infty$为无穷范数.

证明:由公式(5)可知,$v_i \leftarrow v_i \oplus \Delta v_i$,即,任意顶点状态 v_i 都是基于⊕线性递增的.对图中任意顶点执行一次算法 1

操作 f,都会导致 $v=f(v)$变大.而由假设可知,v 的最终收敛结果为 v^*,任意顶点的一次更新操作会使 v 更接近 v^*,因此,我们可以得到$||f(v)-v^*||_\infty \leqslant \partial||v-v^*||_\infty$. □

定理 2(并行异步性). 假设图数据按照某种规则分片($G=G_1\cup G_2\cup\ldots\cup G_n$,且$\forall i,j,G_i\cup G_j=\varphi$),$n$ 个处理器并行执行算法 1,每个处理器执行一个图数据分片,即,任意处理器 i 在 G_i 上执行算法 1,在不需要任何同步机制的情况下,满足假设 1 的情况下,最终整合所有分片上得到的收敛结果为 v'',则 $v''=v^*$.

证明:结合 Bertsekas 教授在文献[20]中提出的收缩定理,考虑其在无穷范数上的应用(即引理 1 描述的收缩性),我们可以得到:以图顶点计算为中心,同步执行或异步执行算法 1 的更新操作足够多次(假设 1),它们得到的结果一致. □

我们将 ASP 模型与 PSW[7]进行比较,ASP 模型具有以下两个好处.

1) 顺序 I/O.在 PSW[7]中,随着 shard 的增多,在滑动窗口的过程中会带来更多额外、随机的 I/O;在 ASP 中,数据采用流式处理,即使 shard 增多,ASP 也会对多个 shard 采用完全顺序 I/O,不会产生额外的随机 I/O,因此最大程度降低 I/O 开销;
2) 线程间可以异步执行.在 PSW[7]中,interval 的所有入边数据和出边数据必须在计算开始之前载入内存;在 ASP 中,整个执行过程中不存在线程同步操作,最大程度降低线程间通信与同步开销,而且能够并行化 I/O 操作和 update 操作.

- 收敛检测

迭代计算需要适时终止计算,我们需要对迭代的进度进行评估来确定计算是否收敛.通常采用的收敛检测方法:(i) 设置最大迭代次数;(ii) 比较连续两次迭代进度.如果它们之间的区别相同或者小于某个阈值,终止迭代.但是,这两种方法在 ASP 异步计算模型下却很难奏效,这是因为 ASP 模型并不是同步迭代,各执行线程对执行算法 1 的进度并不是统一步调,所以既没有全局的迭代次数也不能比较两次迭代的结果,两种传统的收敛检测方法都不可用.我们采用时间间隔来检查收敛进度,如果一段终止检测间隔内的结果差别小于一定阈值,终止计算.当检测到计算结果收敛时,终止迭代计算,并将收敛的顶点状态信息写回磁盘文件.

2.3 ASP支持算法总结

ASP 支持所有符合累加迭代计算规则的算法[4],也就是说,只要迭代更新公式可以表示为公式(1)的形式,即一系列满足分布律的函数 $g_{\{i,j\}}(x)$和一个常数 c_j 通过'⊕'运算累加起来的形式('⊕'运算满足交换律、分配律、对某数值 0 存在恒等性),这个迭代过程便可以通过 ASP 执行.ASP 支持的常见算法有单元最短路径(SSSP)、Adsorption 算法、Jacobi 迭代方法、Connected Components、PageRank、HITS 算法、Katz metric 算法、Rooted PageRank,SimRank 等.

2.4 数据传输与I/O代价

我们将 ASP 与 Graphchi[7]中的 PSW[7]、X-Stream[13]中的以边为中心的处理模型(ECP)在数据传输总量和 I/O 代价方面进行了比较,总结结果见表 2.接下来为分析的具体细节,为了便于参考,我们在表 1 中列出了各个表示符号.

Table 1 Notations

表 1 符号表

Notation	Definition
n,m	$n=\|V\|,m=\|E\|$
P	shard 的数量
C	顶点值数据的大小
D	顶点结构数据的大小
S	顶点编号和 shard 编号大小
I	迭代次数
M	内存大小
B	一个 I/O 单元所访问的磁盘块的大小

Table 2 Data transferred and I/O costs

表 2 数据传输与 I/O 代价

Category	PSW	ECP	ASP
Data size (read)	$2I(C+D)m+ICn$	$I(Cn+(C+D)m)$	$IDm+Sn$
Data size (write)	$2I(C+D)m+ICn$	$I(Cn+Cm)$	Cn
Read I/O	$(2I(C+D)m+ICn)/B+P^2$	$I(Cn+(C+D)m)/B$	$(IDm+Sn)/B$
Write I/O	$2I(C+D)m+ICn)/B+P^2$	$Cn/B+Cm\log_{M/B}P/B$	Cn/B

ASP 使用异步累加计算,并不存在一轮迭代的概念,但是为了便于分析比较,ASP 使用普通的轮询策略,我们可以认为,将所有的 g-shard 轮询一遍称为一次迭代.

在 ASP 中,整个计算过程只载入一次所有的 v-shard,每次迭代中,从磁盘依次载入所有的 g-shard.整个计算过程需要读 $IDm+Sn$ 的数据量.整个计算结束后,会将顶点的值数据写回磁盘,需要写 Cn 的数据量.注意,g-shard 是只读的.为了分析 I/O 代价,我们用 B 表示一个 I/O 单元所访问的磁盘块的大小.根据文献[13],B 在机械硬盘上为 16MB,在 SSD 上为 1MB.在整个计算过程中,ASP 读的 I/O 数量为 $(IDm+Sn)/B$,写的 I/O 数量为 Cn/B.如果 ASP 使用基于 shard 的优先级调度策略,会加快收敛速度,与之对应的迭代次数会变得更少,读写的数据总量与 I/O 数量也会更少.ASP 可以通过缓存访问其相邻顶点的值,但 PSW[7]需要通过边访问相邻顶点的值变得更少,读写的数据总量与 I/O 数量也会更少.ASP 可以通过缓存访问其相邻顶点的值,但 PSW[7]需要通过边访问相邻顶点的值.所以每条边的数据大小为 $(C+D)$.

在一次迭代中,PSW[7]每处理一个 shard 分为 3 个步骤:1) 从磁盘加载子图;2) 更新顶点值和边值;3) 将更新后的值写回磁盘.在步骤 1 和步骤 3 中,每个顶点会载入和写回磁盘一次,会读写 Cn 的数据量.对于每条边数据,在最坏的情况下,每条边被访问两次(每个方向一次),在步骤 1 会读 $2(C+D)m$ 的数据量.在步骤 2 计算更新边值,步骤 3 同样会写 $2(C+D)m$ 的数据量.在整个计算过程中,PSW[7]读写的总数据量都为 $2I(C+D)m+ICn$.在每次迭代过程中,PSW[7]会产生 P^2 次随机读写;在整个计算过程中,PSW[7]读写的 I/O 数量都为 $(2I(C+D)m+ICn)/B+P^2$.

在 X-Stream[13]中,一次迭代被分为:(1) 混合的 scatter/shuffle 阶段;(2) gather 阶段.在阶段 1 中,X-Stream 会加载所有的顶点值数据和边数据,更新每条边,并将 update 后的边数据写回磁盘.因为 update 后的边数据用来在相邻顶点之间传递值,我们假设一条更新的边数据大小是 C,则对于阶段 1,读取的数据量为 $Cn+Dm$,写入的数据量为 Cm.在阶段 2 中,X-Stream 加载所有 update 后的边数据,并更新每一个顶点,所以对于阶段 2,读取的数据量为 Cn,写入的数据量为 Cn,所以 X-Stream 在整个计算过程中,读取的总数据量为 $I(Cn+(C+D)m)$,写入的总数据量为 $I(Cn+Cm)$.在整个计算过程中,X-Stream 读的 I/O 数量为 $I(Cn+(C+D)m)/B$,写的 I/O 数量为

$$Cn/B+Cm\log_{M/B}P/B.$$

3 S-Maiter 原型系统设计与实现

我们基于 ASP 模型实现了 S-Maiter 系统.S-Maiter 用 C++实现,约 12 000 行代码,开源代码发布在 https://github.com/JinjiLi/S-Maiter_v0.9

S-Maiter 的架构如图 3 所示.S-Maiter 把每个图处理任务分为两个阶段:离线预处理阶段和在线计算阶段.离线预处理是为了构造图计算所需要的结构数据.在线计算完成图算法的核心流程,具体分为 3 个步骤.

1) 图数据分片和内存顶点信息初始化;
2) 流式载入结构数据到内存,更新顶点信息,并清除结构数据释放内存;
3) 将内存中的最终结果写回磁盘.

接下来介绍 S-Maiter 的几个重要实现优化技术:I/O 线程优化、内存资源监控、优先级调度优化、和内存-外存计算的自动切换.

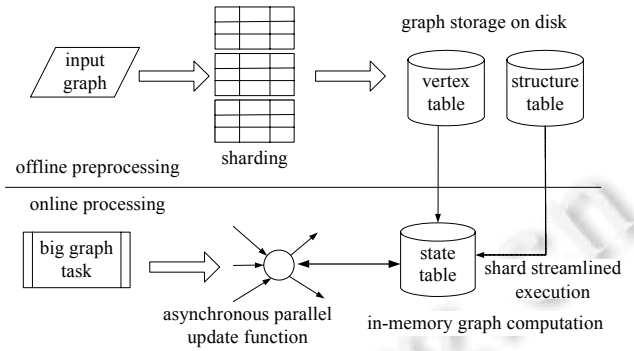

Fig.3 Execution process overview

图 3 整体流程图

3.1 I/O线程优化

S-Maiter 启动多个执行线程进行并行处理,这些执行线程需要读取 *g*-shard 上的图结构数据后才能进行后续的顶点更新操作,而这产生了大量的 I/O.由于执行线程间没有同步操作,计算和更新的速度非常快,但是往往需要等待 I/O 操作的结束,这样 I/O 就成为了 S-Maiter 的性能瓶颈.

在一个执行线程中,既包括 I/O 操作也包括 update 操作,I/O 操作将结构数据载入到内存,update 操作利用结构数据对顶点进行更新.但是这样就将 I/O 操作和 update 操作绑定在了一个执行线程中,即 I/O 资源与计算资源的配置比例固定为 1:1.在这种情况下,I/O 操作和 update 操作的同步次数会增多,导致 I/O 吞吐率和 CPU 资源的利用率下降.

为了解决这些问题,S-Maiter 将 I/O 操作与 update 操作分离,创建多个 update 线程,负责每个顶点的 update 操作,同时创建多个 I/O 线程,负责载入结构数据到内存,这样可以更加合理地分配 I/O 和计算资源.但是如果 I/O 线程相对于 update 线程过多,就会导致缓存数据过多,update 线程执行不过来,这样会使缓存急速扩大,占用大量内存;如果 I/O 线程相对于 update 线程过少,就会出现 update 线程执行太快,但是 I/O 线程太少,数据跟不上,出现 update 线程空闲等待 I/O 的情况.为了避免上述情况的发生,S-Maiter 允许用户根据资源配置和应用特性设置 I/O 线程与 update 线程的配置比例,并使用第 3.2 节的内存监控策略,可以保证 update 线程和 I/O 线程间的平衡,尽量使 I/O 和 CPU 资源处于饱和状态,最大限度地提升系统性能.

3.2 内存资源监控

在 S-Maiter 中,I/O 线程读取结构数据并缓存在内存中,update 线程消化这些结构数据做图顶点状态的更新操作,当这些图结构数据使用完成后释放内存.因为 I/O 线程和 update 线程并行执行,I/O 操作不受 update 线程的控制,可能会发生 update 线程处理图结构数据的吞吐量和 I/O 线程读取图结构数据的吞吐量不匹配的情况.如果 I/O 吞吐量太快,会导致从磁盘上载入的结构数据在内存中缓存的越来越多,最终会导致内存溢出;如果 I/O 吞吐量太慢,就会造成 update 线程等待的情况,造成 CPU 资源浪费.

为了解决这个问题,S-Maiter 设计了一个内存资源监控线程,它负责时时监控内存的使用情况,当检测到缓存的数据快要内存溢出时,监控线程就会给各个 I/O 线程发送信号让 I/O 线程阻塞,使它们不再载入结构数据,等待 update 线程消化完结构数据后释放出一定大小的内存,监控线程检测到不再有可能发生内存溢出时就会给各个 I/O 线程发送信号,让它们继续载入结构数据.内存资源监控策略让 S-Maiter 的内存利用率变高,能够最大限度地利用内存来增加计算速度,并且避免了内存溢出的情况.内存监控线程能够完美地协调 update 线程与 I/O 线程,进行并行计算和磁盘 I/O 操作的同时,系统更加健壮和协调.

3.3 shard级优先级调度

在累加迭代计算中,$|v \oplus \Delta v - v|$代表顶点的优先级,优先级高的顶点可能会对最终的收敛结果产生决定性的

影响.优先对顶点 $j=\text{argmax}|v_j \oplus \Delta v_j - v_j|$执行更新操作,可以加快收敛进度[4].在基于磁盘的方法中,更新计算顶点需要结构数据,从磁盘选择性地载入这些高优先级顶点的结构数据会产生大量随机 I/O,所以这种顶点粒度的优先级策略不可取.我们希望在不产生随机 I/O 的情况下,充分利用顶点优先级的性质.因为 ASP 计算模型在做更新计算时会顺序访问各个 shard 的结构数据,所以本文对顶点优先级进一步泛化,提出了 shard 优先级的概念,把每个 shard 顶点的优先级进行累加,得到 $\sum_{j \in shard(i)} |v_j \oplus \Delta v_j - v_j|$.这个结果就是 shard 的优先级,该结果越大,对应 shard 的优先级越高.基于 shard 的优先级调度适用于单机环境下异步累加迭代计算的调度,先顺序访问高优先级的 shard 的结构数据,会加快迭代计算的收敛速度.

在 S-Maiter 中,最简单的调度策略就是以轮询的方式载入各个 g-shard,update 线程轮询地处理和更新这些 v-shard.虽然这种静态的调度策略实现简单,开销较少,实验结果显示了不错的性能(第 4.3 节),但是基于 shard 优先级的调度策略可以加快计算收敛速度,因此,S-Maiter 除了提供静态的轮询调度,还支持动态的基于 shard 优先级的调度策略.

S-Maiter 创建一个 I/O 优先级调度线程,该线程维护一个基于 shard 的优先级队列,优先级队列里只包含优先级较高的 shard 信息,I/O 线程只对优先级队列里的 shard 进行载入.在迭代计算的过程中,shard 的优先级会随着该 shard 里每个顶点优先级的改变而更新.当处理完队列里所有的 shard,I/O 优先级调度线程就会根据新的 shard 优先级提取新的优先级队列,以便用于后续 I/O 线程的载入.我们在每一轮中提取的 shard 数量,即优先级队列的大小,需要在从使用优先级队列带来的收益与频繁提取优先级队列所付出的代价、开启 I/O 线程的数量之间做出权衡,一般情况下,优先级队列的大小与 I/O 线程数量相同.

3.4 内存模式

上述基于磁盘的处理方式称为外存模式,同时,S-Maiter 也支持内存模式.当处理的图数据大小没有超过内存容量的情况下,S-Maiter 会切换到内存模式.

内存模式下,S-Maiter 将图数据全部读入内存进行迭代计算,由于不产生 I/O,所以比外存模式更高效.内存模式下,对每个顶点执行更新函数是完全并行的,而且 update 线程内支持顶点粒度的优先级调度策略,先计算优先级高的顶点,加快迭代计算的收敛速度.

4 实 验

为了评估 S-Maiter 的性能,本文将 S-Maiter 与流行的单机大图处理系统 GraphChi[7]、X-Stream[13]和分布式累加迭代图处理框架 Maiter[4]做了性能比较,并分析了 S-Maiter 在使用 SSD 时,update 线程与 I/O 线程的数量比对系统性能的影响.

GraphChi[7]是 PSW[7]方法的具体实现,X-Stream[13]是 ECP 的具体实现,Maiter[12]是累加迭代计算的分布式实现.我们将考虑基于轮询调度的 Maiter(distributed Maiter-RR)和基于优先级调度的 Maiter(distributed Maiter-Pri).在本文的对比图中,S-Maiter-RR(file)代表外存模式下基于轮询调度的 S-Maiter,S-Maiter-Pri(file)代表外存模式下基于 shard 级优先级调度的 S-Maiter,S-Maiter-RR(men)代表内存模式下基于轮询调度的 S-Maiter, S-Maiter-Rri(men)代表内存模式下基于优先级调度的 S-Maiter.

4.1 实验环境及实验数据集

所有实验使用的计算机具有相同的配置,每台计算机配有 Intel I5-4690 3.3GHZ 4Core 处理器,1000Mbs 以太网卡、1TB 7200r/m 普通硬盘、120G SSD、8G 运行内存.操作系统为 64 位 Ubuntu 14.04 LTS,Maiter[4]本地集群包括 16 台 slave 计算机、1 台 master 计算机.

我们从 Stanford 的大规模数据集(stanford large network dataset collection)[21]中选用两个真实的图数据集 LiveJournal和 Pokec.另外,我们还基于 SNAP[22]合成了 2 个生成图数据集,合成的图数据集中,每个顶点的入度符合对数正态分布,位置参数为–0.5,尺度参数为 2.3.基于此入度分布,我们随机选取一些顶点连接一个顶点.这些

对数正态分布的参数从一些实际生活中的图(包括社交网络图和网页链接图等)中提取[21].数据集的相关信息见表 3.

Table 3 Summary of the experimental datasets

表 3 实验数据集相关信息

Dataset	# Vertex (million)	# Edge (million)	#Size
LiveJournal	4.9	69	424MB
Pokec	1.6	30.7	1.1GB
Synthetic-10m	10	360	5.5GB
Synthetic-13m	13	500	10.1GB

与 Graphchi[7]相同,S-Maiter 允许用户明确指定内存预算大小,为了避免 Graphchi[7]和 S-Maiter 使用预算的内存之外的内存空间,我们在所有的实验中禁用了操作系统的页缓存功能.

实验应用了两种迭代算法,包括 PageRank 和弱连通分量(WCC).对于 S-Maiter,每隔一段时间检查迭代进度(PageRank 每隔 10s,WCC 每隔 5s).对于 Graphchi[7]和 X-Stream[13],在每次迭代结束之后检查算法是否收敛.离线运行这两个迭代算法,取 200 次迭代计算之后的结果作为这个迭代计算的收敛结果.当(每隔一段时间后的或每次迭代后的)迭代结果与这个收敛结果的距离小于 0.0001 时,终止迭代计算.每个实验运行 3 次,取平均值作为实验结果.第 4.1 节~第 4.3 节的实验都是在机械硬盘上对 S-Maiter 和 Graphchi[7]、X-Stream[13]进行性能评估.

4.2 整体收敛时间对比

本节首先测试 PageRank 和 WCC 算法在不同数据集下的收敛时间.内存预算大小设置为 2GB,数据集 Pokec 不足 2GB.但为了展示 S-Maiter 在外存模式下性能,我们依然在外存模式下进行计算.

图 4 显示了不同数据集下,PageRank 算法在 distributed Maiter-RR、distributed Maiter-Pri、Graphchi[7]、X-Stream[13]、S-Maiter-RR(file)、S-Maiter-Pri(file)下的收敛时间.distributed Maiter-Pri 使用了顶点级的优先级调度策略,可以比轮询调度的 distributed Maiter-RR 快一些;Graphchi[7]使用滑动窗口的方法,比使用 8 节点的分布式 Maiter[4]更快;X-Stream[13]使用了以边为中心的处理模型,进一步提升了性能;S-Maiter 使用了 ASP 模型,可以比 X-Stream[13]的性能更高;S-Maiter-Pri(file)使用了 shard 级先级调度策略,相对于轮询调度的 S-Maiter-RR(file)加快了收敛速度.

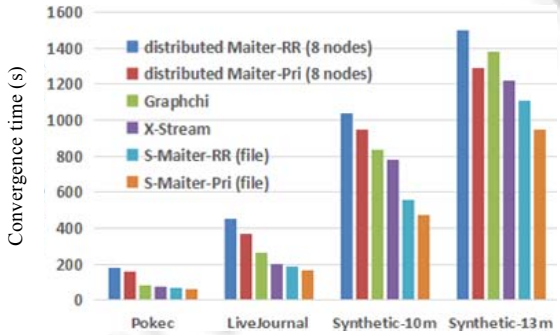

Fig.4 Convergence time of PageRank on different datasets

图 4 PageRank 在不同数据集下的收敛时间

图 5 显示在不同数据集下,WCC 算法在 distributed Maiter-RR、distributed Maiter-Pri、Graphchi[7]、X-Stream[13]、S-Maiter-RR(file)、S-Maiter-Pri(file)下的收敛时间,我们可以看到与 PageRank 类似的结果,但 X-Stream 运行 WCC 算法比其他框架要慢,主要是因为分布式 Maiter,S-Maiter 能利用累加迭代计算的性质天然的过滤掉一些不必要的顶点更新,Graphchi[7]通过选择调度策略能跳过一些不必要的顶点更新.由于 X-Stream 是以边为中心的计算框架,不支持顶点的选择调度策略,这在 WCC 算法中表现得尤为突出.

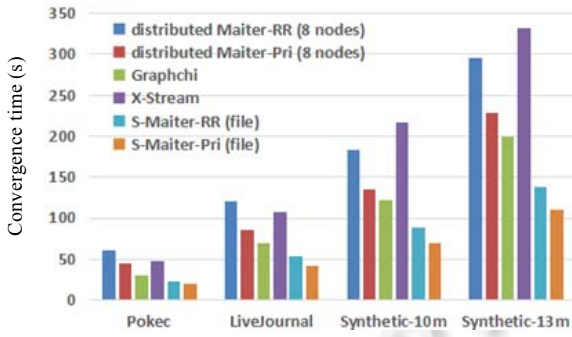

Fig.5 Convergence time of WCC on different datasets

图 5 WCC 算法在不同数据集下的收敛时间

4.3 内存配额对性能的影响

图 6 显示了随着内存预算大小从 0.5GB~3GB,PageRank(运行 Synthetic-10m 数据集)在 GraphChi[7]、X-Stream[13]、S-Maiter-RR(file)、S-Maiter-Pri(file)下的收敛时间.从图中可以看到:S-Maiter-RR(file)的性能相对 GraphChi[7]提高了 1.5 倍,相对 X-Stream[13]提高了 1.4 倍;S-Maiter-Pri(file)性能相对 GraphChi[7]提高了 1.7 倍,相对 X-Stream[13]提高了 1.6 倍.例如:在内存预算大小为 3GB 时,GraphChi[7]需要 820.658s,X-Stream[13]需要 777.22s,S-Maiter-RR(file)只需要 559.156s,S-Maiter-Pri(file)只需要 474.156s.

S-Maiter 相对于 Graphchi[7]会产生更少的 shard,如图 7 所示.在内存预算大小从 0.5GB~3GB 的情况下,Graphchi[7]产生了 5 个~23 个 shard;随着 shard 的增多,在滑动窗口的过程中会带来更多额外的随机的 I/O.而 S-Maiter 只产生 4 个 shard,这是因为 S-Maiter 分片只是为了支持基于 shard 的优先级调度,在 S-Maiter 中,即使 shard 增多,也不会产生随机的 I/O.

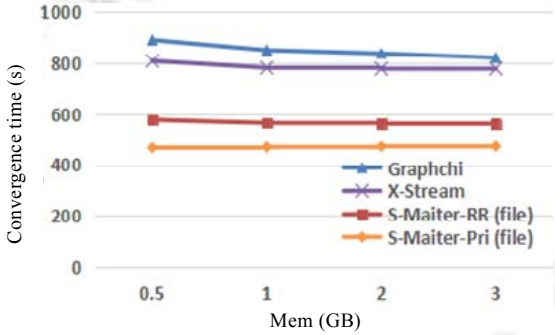

Fig.6 Convergence time comparison with different memory quotas (Synthetic-10m)

图 6 不同内存配额下的收敛时间对比(Synthetic-10m)

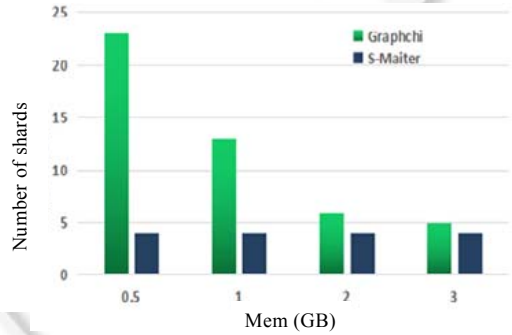

Fig.7 Number of shards comparison with different memory quotas (Synthetic-10m)

图 7 不同内存配额下的 shard 数量对比(Synthetic-10m)

4.4 读写数据量对比

图 8 和图 9 分别显示了 Graphchi[7]、X-Stream[13]、S-Maiter-RR(file)、S-Maiter-Pri(file)的写/读数据量.结果显示,S-Maiter 写/读数据量比 Graphchi[7]和 X-Stream[13]要小得多.具体来说:在整个计算过程中,在内存预算大小从 0.5GB~3GB,无论内存预算是多少,Graphchi[7]都要将 86.5GB 的数据量写入磁盘,并从磁盘读取相同大小的数据量到内存.X-Stream[13]将 56.438GB 的数据量写入磁盘.S-Maiter-RR(file)和 S-Maiter-Pri(file)只将最终收敛结果写回磁盘,所以只写了 0.17G 的数据量.在读取数据量方面,X-Stream[13]读取了 112.65GB 的数据量,

S-Maiter-RR 读取 41.7GB 数据量,S-Maiter-Pri(file)只读取 36GB 的数据量.这是因为 S-Maiter-Pri(file)使用了 shard 级的优先级调度,加快了收敛速度,所以读取的数据量相对 S-Maiter-RR(file)会少一些.S-Maiter 在数据访问中的优越性主要是由于将不变的结构数据与可变的值数据分离,并将可变的值数据缓存在内存中,对值数据的更新和重复访问可以在内存中完成,最小化了对图数据访问产生的 I/O 开销.

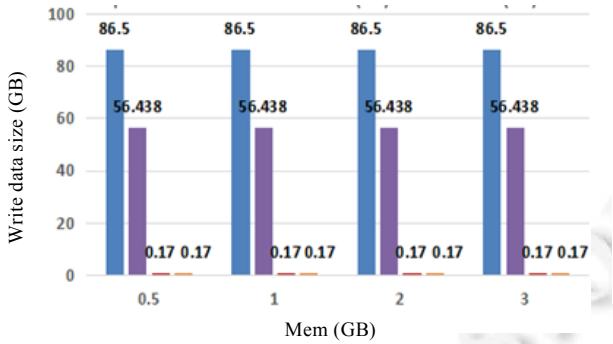

Fig.8 Write data size comparison (Synthetic-10m)

图 8 写入数据量对比(Synthetic-10m)

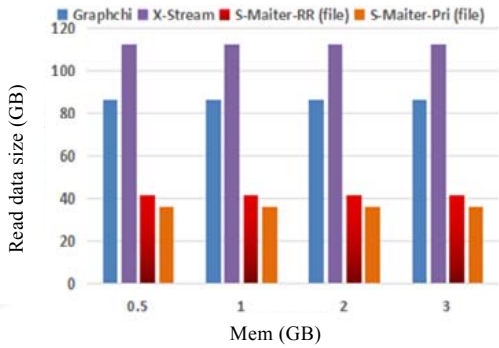

Fig.9 Read data size comparison (Synthetic-10m)

图 9 读入数据量对比(Synthetic-10m)

4.5 与分布式Maiter的性能对比

本节对 S-Maiter 和分布式 Maiter 进行了性能对比实验,内存预算大小设置为 4GB.S-Maiter 的内存模式相比于外存模式性能提升很大.图 10 所示为 PageRank(运行 Pokec 数据集)算法在 S-Maiter-RR(file)、S-Maiter-Pri(file)、S-Maiter-RR(men)和 S-Maiter-Pri(men)下的收敛时间.可以看出:内存模式相对外存模式性能提高了大约 1.5 倍,这是因为内存模式并不产生 I/O 开销.

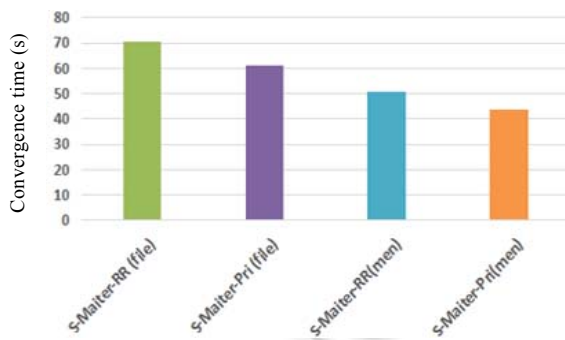

Fig.10 Memory mode vs.external memory mode on convergence time (Pokec)

图 10 内存模式与外存模式的收敛时间比较(Pokec)

然而,当运行比较大的数据集(如 Synthetic-13m)时,由于数据集过大,4GB 的内存已经容纳不下了,只能在外存模式或分布式环境下进行计算.为此,我们用 S-Maiter-RR(file)、S-Maiter-Pri(file)与 distributed Maiter-RR、distributed Maiter-Pri 在 Synthetic-13m 数据集上进行对比实验.

图 11 显示了 PageRank 算法(运行 Synthetic-13m 数据集)在分布式 Maiter[4]与单机外存 S-Maiter 下的收敛时间对比.当分布式 Maiter[4]在一个计算节点上计算时,虽然在内存运算,但是相对于 S-Maiter 的内存模式,分布式 Maiter[4]在单个计算节点上并没有使用多线程进行更新计算,S-Maiter 在内存模式下的计算速度快于外存模式(如图 10 所示),S-Maiter 在外存模式下的计算速度快于分布式 Maiter[4]在单个计算节点上的计算速度.集群数量大于 1 时,分布式 Maiter[4]存在大量的网络开销,但随着集群大小从 2 增长到 16 时,Maiter[4]的性能也会逐渐提升.当集群大小为 16 时,distributed Maiter-RR 需要 953s,distributed Maiter-Pri 需要 780.83s,S-Maiter-RR(file)需要

1 013.42s,S-Maiter-Pri(file)需要 848.167s.我们可以看到,单机运行 S-Maiter 的收敛时间与 16 台分布式集群上 Maiter[4]的收敛时间相差不多.S-Maiter 在合理的时间内,只用了 1 台计算机解决了分布式 Maiter[4]用 16 台计算机集群解决的问题,减少了获取和管理大规模集群需要的昂贵代价,也不用考虑复杂的图分区工作.

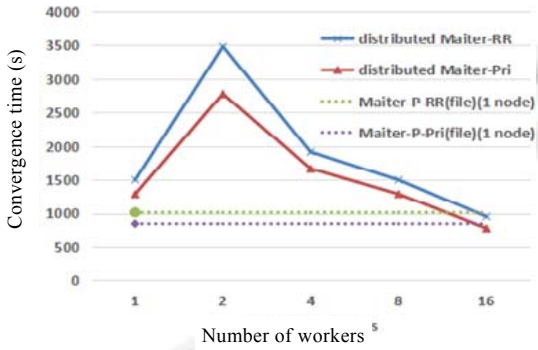

Fig.11 S-Maiter (1 computing node) vs. distributed Maiter (1~16 computing nodes) on convergence time

图 11 单机 S-Maiter 与分布式 Maiter (1~16 个节点)的收敛时间对比

4.6 update线程与I/O线程数量比

在使用 SSD 时,我们可以开启多个 I/O 线程提高读取数据的存储量,提高计算效率.为了探究 update 线程与 I/O 线程的数量比对 S-Maiter 性能的影响,本文设计了在不同比例下的 S-Maiter 执行迭代算法的收敛速度对比实验.图 12 显示了在不同的 update 线程与 I/O 线程比例下,WCC(运行LiveJournal数据集,内存预算大小 0.5GB)在 S-Maiter-RR(file)下收敛的平均时间.从图中可以看出:随着 update 线程与 I/O 线程数量比例不断增大,收敛速度出现先提高后降低的现象,并在 update 线程与 I/O 线程数量比为 8:2 时收敛速度最快.update 线程与 I/O 线程的数量比对 S-Maiter-Pri(file)的影响类似,在此不再详述.

经分析发现:如果 I/O 线程相对于 update 线程过多,就会出现缓存结构数据过多,update 线程执行不过来,出现“撑着了”的情况;如果 I/O 线程相对于 update 线程过少,就会出现 update 线程执行太快,但是 I/O 线程太少,数据跟不上,出现“吃不饱”的情况.这两种情况导致了图 12 所示的现象,所以应该避免这两种情况的发生.

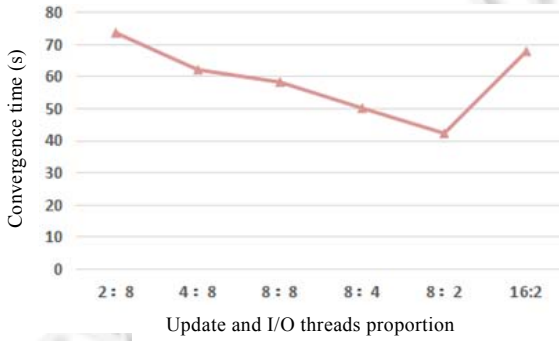

Fig.12 update and I/O threads proportion (LiveJournal)

图 12 update 线程与 I/O 线程比例(LiveJournal)

5 相关工作

随着大规模数据集的产生以及学术界和工业界对图数据分析的深入研究,出现了一系列大规模图数据处理系统,来完成各种复杂的大规模图数据分析任务.

- 基于同步方式的分布式图处理系统

PEGASUS[8]和 GBase[23]基于 MapReduce,并且支持矩阵向量乘法.Pregel[5]并不基于 MapReduce,Pregel[5]在概念模型上遵循 BSP 模型[24],整个计算过程由若干顺序执行的超级步(super step)组成,系统从一个超级步迈向下一个超级步,直达到算法的终止条件.Blogel[25]基于 BSP 模型,扩展了以顶点为中心的编程模型,提出了以块为中心(block-centric)的编程模型.它以块为单位进行图处理,开发人员可以综合考虑块内其他顶点的状态来指定图顶点执行不同的计算.GraphX[26]是一个基于 Spark[16]构建的图处理系统,支持弹性分布式图(resilient distributed graph,简称 RDG)的抽象数据结构.它将图存储为表格结构,并将图计算操作实现成了对几个表格的分布式连接(join)操作,以利用 Spark[16]提供的底层计算引擎实现高效的内存图处理.Pregelix[27]是一个基于 Hyracks[28]构建的 BSP 模型的分布式外存实现版本,它将图数据和消息数据存储为数据多元组(data tuples),通过分布式连接操作来实现数据分发.PrIter[17]是基于 Hadoop Online[29]构建的 BSP 模型的图处理系统,它支持带优先级的图处理,可以确保加速图计算的收敛,尤其适合在线 top-*k* 查询.Trinity[11]利用分布式内存键值对存储来支持在线图查询操作和离线的 BSP 图处理.

- 基于异步方式的分布式图处理系统

分布式版本的 GraphLab[6]同时支持同步图处理和异步图处理,采用拉取(pull)模式从邻居顶点获取状态信息,并基于 Gather-Apply-Scatter(GAS)模型,通过分布式锁来保证图计算过程中需要满足的多种一致性需求. GraphLab 的同步计算表现出了较好的性能,性能优于大部分同步图处理系统.然而对于异步图计算,仅在支持部分异步图算法(置信传播算法 Belief Propagation 等)上有着不错的性能,对于大多数图算法(PageRank 等),异步图处理的性能却相当糟糕[6],甚至比同步计算性能差数十倍.究其原因,是由于为了实现异步数据一致性而实现的分布式锁,它造成了极大的性能开销,导致得不偿失.近年来,除了 GraphLab[6],也涌现除了其他几个典型的支持异步图处理的系统.GRACE[30]支持用户可定制的图顶点调度和消息选择处理策略,然而它的运行环境是单机共享内存环境下,而不是分布式环境,计算调度等在共享内存环境下容易处理的问题一旦转移到分布式环境下将变得非常难以解决,这导致了 GRACE 系统的可用性受限;Giraph++[31]提出以子图为中心(graph-centric)的处理模型,通过图数据分割后,允许子图内图顶点多次迭代,而全局同步被尽量推迟执行,避免频繁全局同步带来的大量系统等待开销; GiraphUC[32]相对应于 BSP 模型提出 BAP(barrierless asynchronous parallel)模型,并基于开源的同步图处理系统 Giraph[33]实现,它的核心思路与 Giraph++[31]类似,都是通过区分本地同步和全局同步来提高性能.

- 基于累加计算的分布式图处理系统 Maiter

Maiter[4]采用基于推送(push)模式的异步累加式的图处理方法,各图顶点之间交互的信息并不是顶点状态而是顶点状态的变化,并且累积这些变化信息即可得到最终的收敛结果.它不要求包括本地同步或者全局同步等的任何同步操作,各计算节点在交互信息时完全独立,这样即达到了理想的完全异步状态.Maiter 目前已经支持一大类算法,包括 PageRank,SSSP,SimRank 等.本文提出的 S-Maiter 同样基于异步累加计算模型,利用单机大容量磁盘和并行计算技术进行图处理优化,对于没有必要在分布式环境执行的图处理问题或者缺少分布式计算条件的情况下提供一种更好的解决方案.分布式 Maiter 利用分布式集群结合分布式计算的特点加速图计算,而 S-Maiter 利用单机大容量磁盘结合图处理的 I/O 特点和并行计算特点来加速图计算.

- 单机图处理系统

X-Stream 遵循以边为中心的计算模型.X-Stream 会将部分中间结果写回磁盘以便后续的处理,这将产生双倍的 I/O 代价,导致额外的计算成本和数据加载开销.此外,X-Stream 也不支持顶点的选择调度策略.GraphChi 将图数据在磁盘中组织成若干个分片,每个分片包含一系列顶点信息和以及与这些顶点相关的入边和出边信息,GraphChi 要求每个分片都能够装进内存.GraphChi 在计算之前必须将整个分片全部载入内存,这种约束阻碍了计算和 IO 的并行性.在相同内存限制下,GraphChi 相对 S-Maiter 会产生更多的分片,随之也产生了更多的数据传输和随机 I/O.S-Maiter 可以不间断地流式计算分片里的数据,并且将可变的顶点值数据缓存到内存中,减少了随机 I/O,最小化 I/O 代价.TurboGraph[12]虽然可以无延迟地处理图数据,但是 TurboGraph 是针对 SSD 设计的,S-Maiter 不仅可以应用在 SSD,还可应用在廉价的机械硬盘.

6 结 论

本文提出了流式处理的异步计算方案 ASP,一种用基于单机大容量磁盘的方法解决大图计算的方案,包括适用于异步累加迭代的图存储模型和计算模型.并基于该模型实现了基于外存的图计算框架 S-Maiter,有效解决了频繁更新和读取磁盘数据导致大量 I/O 的问题.实现了对磁盘数据的完全顺序访问,有效利用了内存和 CPU 资源,能并行化图数据的流式载入和更新函数的异步执行.通过一系列的实验,其结果表明:S-Maiter 比 Graphchi 和 X-Stream 更快,比 Maiter 性价比更高.

References:

[1] Yu G, Gu Y, Bao YB, Wang ZG. Large scale graph data processing on cloud computing environments. Chinese Journal of Computers, 2011,34(10):1753−1767 (in Chinese with English abstract).

[2] Li XT, Li JZ, Gao H. An efficient frequent subgraph mining algorithm. Ruan Jian Xue Bao/Journal of Software, 2007,18(10): 2469−2480 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/2469.htm [doi: 10.1360/jos182469]

[3] Pan W, Li ZH, Wu S, Chen Q. Evaluating large graph processing in MapReduce based on message passing. Chinese Journal of Computers, 2011,34(10):1768−1784 (in Chinese with English abstract).

[4] Zhang Y, Gao Q, Gao L, Wang C. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Trans. on Parallel & Distributed Systems, 2014,25(8):2091−2100. [doi: 10.1109/TPDS.2013.235]

[5] Malewicz G, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Cazjkowski G. Pregel: A system for large-scale graph processing. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2010. 135−146. [doi: 10.1145/1582716.1582723]

[6] Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proc. of the VLDB Endowment, 2012,5(8):716−727. [doi: 10.14778/2212351.2212354]

[7] Kyrola A, Blelloch GE, Guestrin C. GraphChi: Large-Scale graph computation on just a PC. In: Proc. of the 10th USENIX Symp. on OSDI. 2012. 31−46.

[8] Kang U, Tsourakakis CE, Faloutsos C. PEGASUS: A peta-scale graph mining system implementation and observations. In: Proc. of the 9th IEEE Int'l Conf. on Data Mining. IEEE Computer Society, 2009. 229−238. [doi: 10.1109/ICDM.2009.14]

[9] Chen R, Weng X, He B, Yang M. Large graph processing in the cloud. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2010. 1123−1126. [doi: 10.1145/1807167.1807297]

[10] Krepska E, Kielmann T, Fokkink W, Bal H. HIPG: Parallel processing of large-scale graphs. ACM SIGOPS Operating Systems Review, 2011,45(2):3−13. [doi: 10.1145/2007183.2007185]

[11] Shao B, Wang H, Li Y. Trinity: A distributed graph engine on a memory cloud. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2013. 505−516. [doi: 10.1145/2463676.2467799]

[12] Han WS, Lee S, Park K, Lee JH, Kim MS, Kim J, Yu H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In: Proc. of the 19th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. ACM Press, 2013. 77−85. [doi: 10.1145/2487575.2487581]

[13] Roy A, Mihailovic I, Zwaenepoel W. X-Stream: Edge-Centric graph processing using streaming partitions. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. ACM Press, 2013. 472−488. [doi: 10.1145/2517349.2522740]

[14] Cheng J, Liu Q, Li Z, Fan W, Lui JCS, He C. VENUS: Vertex-Centric streamlined graph computation on a single PC. In: Proc. of the 2015 IEEE 31st Int'l Conf. on Data Engineering (ICDE). IEEE, 2015. 1131−1142. [doi: 10.1109/ICDE.2015.7113362]

[15] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. HotCloud, 2010,10(10-10):95.

[16] Zhang Y, Gao Q, Gao L, Wang C. Imapreduce: A distributed computing framework for iterative computation. In: Proc. of the 2011 IEEE Int'l Symp. on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). IEEE, 2011. 1112−1121. [doi: 10.1109/IPDPS.2011.260]

[17] Zhang Y, Gao Q, Gao L, Wang C. PrIter: A distributed framework for prioritized iterative computations. In: Proc. of the 2nd ACM Symp. on Cloud Computing. ACM Press, 2011. 13. [doi: 10.1145/2038916.2038929]

[18] Engle C, Lupher A, Xin R, Zaharia M, Franklin MJ, Shenker S, Stoica I. Shark: Fast data analysis using coarse-grained distributed memory. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2012. 689−692. [doi: 10.1145/2213836.2213934]

[19] Kang U, Tong H, Sun J, Lin CY, Falontsos C. GBASE: A scalable and general graph management system. In: Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Diego: DBLP, 2011. 1091−1099. [doi: 10.1145/2020408.2020580]

[20] Bertsekas DP. Distributed asynchronous computation of fixed points. Mathematical Programming, 1983,27(1):107−120. [doi: 10.1007/BF02591967]

[21] Stanford dataset collection. http://snap.stanford.edu/data/

[22] Snap. http://github.com/snap-stanford/snap

[23] Gerbessiotis AV, Valiant LG. Direct bulk-synchronous parallel algorithms. Journal of Parallel Distributed Computing, 1994,22(2):251−267. [doi: 10.1006/jpdc.1994.1085]

[24] Yan D, Cheng J, Lu Y, Ng W. Blogel: A block-centric framework for distributed computation on real-world graphs. In: Proc. of the VLDB Endowment. 2014. 1981−1992. [doi: 10.14778/2733085.2733103]

[25] Xin RS, Crankshaw D, Dave A, Gonzalez JE, Franklin MJ, Stoica I. GraphX: Unifying data-parallel and graph-parallel analytics. In: Proc. of the Computer Science. 2014.

[26] Borkar V, Borkar V, Jia J, Carey MJ, Condie T. Pregelix: Big(ger) graph analytics on a dataflow engine. Proc. of the VLDB Endowment, 2014,8(2): 161−172. [doi: 10.14778/2735471.2735477]

[27] Borkar V, Carey M, Grover R, Onose N, Vernica R. Hyracks: A flexible and extensible foundation for data-intensive computing. In: Proc. of the IEEE Int'l Conf. on Data Engineering. IEEE Computer Society, 2011. 1151−1162. [doi: 10.1109/ICDE.2011.5767921]

[28] Hadoop. http://hadoop.apache.org/

[29] Wang G, Xie W, Demers A, Gehrke J. Asynchronous large-scale graph processing made easy. In: Proc. of the CIDR. 2013.

[30] Balmin A, Balmin A, Corsten SA, Tatikonda S, Mcpherson J. From "think like a vertex" to "think like a graph". Proc. of the VLDB Endowment, 2013, 7(3):193−204. [doi: 10.14778/2732232.2732238]

[31] Han M, Daudjee K. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. In: Proc. of the VLDB Endowment. 2015.

[32] Apache Giraph. http://giraph.apache.org

附中文参考文献:

[1] 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.

[2] 李先通,李建中,高宏.一种高效频繁子图挖掘算法.软件学报,2007,18(10):2469−2480. http://www.jos.org.cn/1000-9825/18/2469.htm [doi: 10.1360/jos182469]

[3] 潘巍,李战怀,伍赛,陈群.基于消息传递机制的 MapReduce 图算法研究.计算机学报,2011,34(10):1768−1784.

李金吉(1993－),男,吉林松原人,硕士生,主要研究领域为大数据处理,分布式系统.

于戈(1962－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为数据库,分布式系统,嵌入式系统.

张岩峰(1982－),男,博士,教授,CCF 专业会员主要研究领域为大数据处理,分布式系统,云计算.

高立新(1968－),女,博士,教授,博士生导师,主要研究领域为社交网络,路由策略,网络虚拟化,云计算.

巩树凤(1991－),男,博士生,主要研究领域为大数据处理,分布式系统.