基于消息传递机制的 MapReduce 图算法研究

潘 巍[1)] 李战怀[1)] 伍 赛[2)] 陈 群[1)]

[1)](西北工业大学计算机学院 西安 710072)

[2)](新加坡国立大学计算机学院 新加坡 119077)

摘 要 单机运行环境难以满足基于海量数据的大图算法对时空开销的需求,如何设计高效的面向云计算环境的分布式大图算法越来越受到人们的关注,MapReduce 作为云计算的核心计算模式受限于易并行(EP)计算模型的制约不易表达图算法.文中突破了 MapReduce 基于易并行计算的假设,增强了 MapReduce 既有的编程规范,新的大同步(BSP)计算模型既能保证兼容旧的 MapReduce 作业可以无改动的运行,同时引入消息传递机制允许变化的状态数据在并行任务的超级步间进行交互.系统提供高度灵活的消息自定义接口,针对不同应用需求设计了轻量级和重量级两种自适应的消息传递机制,更高效地支持有数据交互需求的包含迭代处理的一大类图算法.在真实大规模图数据集上的实验结果表明,相比于原始的 MapReduce 作业外部链式处理,该文提出的 BSP 模型下的内部超级步迭代计算模式大幅降低了大图算法的处理时间.

关键词 云计算;MapReduce;大同步模型;消息传递;图算法;PageRank

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01768

Evaluating Large Graph Processing in MapReduce Based on Message Passing

PAN Wei[1)] LI Zhan-Huai[1)] WU Sai[2)] CHEN Qun[1)]

[1)](*School of Computer Science and Technology, Northwestern Polytechnical University, Xi'an* 710072)

[2)](*School of Computing, National University of Singapore, Singapore* 119077)

Abstract Since analyzing large-scale graph is usually difficult to be implemented on a single machine, how to design efficient parallel large-scale graph algorithms is receiving more and more attention. Constrained by embarrassingly parallel assumption, parallel graph algorithms are not easy to express in MapReduce. Inspired by Bulk Synchronous Parallel model, we propose a message-enhanced version of Hadoop MapReduce that breaks its key assumption. Enhanced implementation is compatible with original Hadoop MapReduce, existing Hadoop MapReduce programs can run directly on this platform without modification, and uses message passing mechanisms to facilitate interactive data communication between supersteps of tasks. It also provides a highly flexible self-defined message passing interface and two adaptive message passing mechanisms to support efficient implementation of graph algorithms with data transition and iterative computation. The experimental results on the real Stanford large network dataset collection demonstrate the superiority of enhanced version over original Hadoop MapReduce on PageRank algorithm.

Keywords cloud computing; MapReduce; BSP model; message passing; graph algorithms; PageRank

---

收稿日期:2011-07-18;最终修改稿收到日期:2011-09-26.本课题得到国家自然科学基金(61033007,60970070)、国家"八六三"高技术研究发展计划重大项目(2009AA01A404)、NSFC-JST 重大国际(地区)合作项目(60720106001)资助.**潘 巍**,男,1977 年生,博士研究生,主要研究方向为云数据管理、RFID 数据管理、数据挖掘技术.**李战怀**(通信作者),男,1961 年生,博士,教授,博士生导师,主要研究领域为数据库理论与技术.E-mail: lizhh@nwpu.edu.cn.**伍 赛**,男,1980 年生,博士,主要研究方向为 P2P 技术、云数据管理.**陈 群**,男,1976 年生,博士,教授,博士生导师,主要研究领域为云数据管理、RFID 数据管理、XML 数据库技术.

1 引　言

云计算、物联网和社交网络等技术的飞速发展，极大丰富了各种海量异构数据的产生渠道，高可扩展的海量数据并行处理是其中关键性的技术之一. MapReduce[1]是 Google 提出的一种处理超大规模数据集的分布式并行编程模型，也是云计算目前的核心计算模式. MapReduce 编程规范(Paradigm)借用函数式语言的映射(Map)和规约(Reduce)原语，通过自动切分输入数据集，在独立的数据切片(split)上应用 Map 操作产生中间结果的键值对(key/value pair)集合，然后通过分区操作(partition)确保具有同样键的数据映射到同一分区中并借助混洗操作(shuffle)在无共享(shared-nothing)的集群网络中传递中间结果，最后在不同的中间结果分区上应用 Reduce 操作产生最终的规约结果. 也就是说整个 MapReduce 作业的执行主要分为 Map 和 Reduce 两个处理键值对集合的并行运算阶段，Map 阶段同时存在多个异步执行的处理输入键值对的 Map 操作(下文称之为 Mapper)，Reduce 阶段同时存在多个异步执行的处理中间键值对的 Reduce 操作(下文称之为 Reducer)，Map 和 Reduce 阶段之间需要同步交互数据. 利用这种方式，MapReduce 屏蔽了底层复杂的并行处理细节，极大简化了并行程序的设计，应用开发者只需要关注与具体应用相关的 Map 和 Reduce 的处理逻辑本身，而将其余复杂的并行事务交与系统完成.

近年来，不少科研机构和公司团体都研发了自己的基于 MapReduce 设计规范的海量数据并行处理系统，其中 Apache 的 Hadoop 是 MapReduce 的一种开源实现，也是目前学术界和业界事实上的海量数据并行处理标准. Hadoop 可以方便地部署在通用的商用机集群中，为简化用户的并行编程环境，高抽象度的 Hadoop 仅为使用者提供了有限的执行策略，因此在某些应用上(特别是图的算法)只能采取高通用性低效率的方法，意图在易用性和执行性能上进行折衷. 大量的分布式图算法都包含明显的迭代过程并且在数据内部存在一定的依赖关系，在原始的 MapReduce 中，只能通过多趟的外部链式调用 MapReduce 作业[2-3]来支持迭代和数据交互，这不但需要开发者主动干预执行的过程，还不可避免会引入大量不必要的重复代价. 因为对于外部链式调用的作业，每轮迭代都不可避免会产生作业启动开销(包括作业分发、输入划分、任务划分等一些初始化操作，根据实验数据分析这部分 warm-up 代价占到整个作业执行代价的 7～10%左右)、不变数据的序列化和网络传输开销，迭代中间结果的 HDFS 持久化开销等. 某些进化的基于 Hadoop 的系统或类 MapReduce 系统，如 HaLoop[4]、Twister[5]等试图在作业内部完成迭代，以期减少多轮中间结果的持久化代价，但是在实现上多采用分布式内存和本地缓存来存储图的拓扑，这种设计策略对具有海量数据的大图处理存在一定的局限性.

这种并不优雅的解决问题的方式源于MapReduce 编程规范中一个很重要的假设：Mapper 或 Reducer 间不存在任何依赖，可以无交互的在不同的数据切片上独立执行. 这是一种称之为“易并行计算”[6](Embarrassingly Parallel Computation，EPC)的“理想”的并行计算模式. 基于此模式可以解决的并行问题都可以分解为多个完全独立的部分且他们能够异步独立执行，这种理想模式下异步并行的 Mapper 之间(或 Reducer 之间)不存在通信，数据交互仅依赖于 Mapper 和 Reducer 之间的 Shuffle 处理. 因此一些有中间数据交互需求的包含迭代过程的并行算法只能借助 MapReduce 作业的链式调用来满足多轮迭代的数据交互需求，并根据应用的迭代收敛条件决定何时终止链式的调用.

本文突破了 MapReduce 基于易并行计算的假设，设计了基于 Hadoop 的支持大同步编程规范[7](Bulk Synchronous Programming，BSP)的并行计算框架. 改进的框架增强了 MapReduce 既有的编程规范，既能保证兼容旧的 MapReduce 作业可以无改动的运行在新的并行运行环境中，同时利用有效的消息传递机制允许变化的中间状态数据在任务间进行交互. 新框架中将 Map(Reduce)阶段分解为多个同步的超级步，超级步内任务异步高度并行，超级步间利用消息传递机制完成任务间的数据交互. 新框架利用图节点驱动的方式更高效地支持有信息交互需求和包含迭代处理过程的一大类图算法，极大地减少了图算法在 MapReduce 既往处理模式中不必要的代价. 本文主要贡献包括：

(1)抽象了支持大同步编程规范的改进的并行计算框架的编程模型，利用易并行计算的形式化定义描述了 MapReduce 的框架实现，通过引入大同步模型增强了 MapReduce 的编程规范，为解决包含迭代过程且有交互需求的分布式大图算法提供了一种高效的实现途径.

(2)定义了改进的并行计算框架中图的通用表示格式，抽象了以节点为驱动的分布式图并行计算模型.

(3)设计了障栅消息格式,实现了自适应的基于 Hadoop 原有消息传递结构的轻量级消息传递机制和独立消息服务器模式的重量级消息传递机制,系统利用对使用者透明的隐式同步完成超级步间的同步和消息传递.

(4)通过在 72 节点的集群环境下使用标准的 Stanford 大型网络数据集设计并完成了相关实验,实验结果表明在给定的数据集和实验环境下,改进版的 Hadoop 集群中 PageRank 算法的执行性能相比于原始的 Hadoop 集群最高可提升 51%.

本文第 2 节对相关工作进行介绍;第 3 节描述易并行计算和大同步计算的并行编程模型并分析了改进的并行计算框架下作业执行的代价;第 4 节抽象大同步模型下图算法的计算模型,并以 PageRank 为典型应用给出具体的示例;第 5 节给出障栅消息的定义和格式,并详细描述自适应的轻量级和重量级消息传递机制;第 6 节通过实验结果验证改进的并行计算框架对 PageRank 算法的高效性;最后一节对全文进行总结并给出未来研究工作的展望.

2 相关工作

MapReduce 已经受到了学术界和工业界的广泛关注和讨论[8-13],目前具有代表性的基于 MapReduce 的海量数据并行处理系统有 Google 的 Sawzall[14] 系统,微软的 Dryad[15] 系统和 SCOPE[16] 系统、Yahoo 的 Pig[3] 系统以及 Apache 的 Hive[2] 系统. 其中 Sawzall 是一种用于极大规模数据集合的平行分析语言,其采用 filter-aggregator(过滤器-聚合器)两阶段式的执行方式,通过限制编程模式来保证高并发和扩展能力,它在语言级别保证了平行处理的任务间不存在相互依赖. Dryad 提供了能够在 Windows 或者. Net 平台上编写大规模的并行应用程序的分布式并行计算基础平台,利用过程式高级语言接口 DryadLINQ 使得无并行编程经验的程序员可以轻松完成大规模的分布式计算任务. SCOPE 系统是建立在 Dryad 之上的用于大规模数据并行分析的声明式语言,允许用户自定义函数来实现更丰富的计算. Pig 和 Hive 基于 Hadoop 提供了高层的语言支持,Pig 引入了一种 SQL-Like 语言 Pig Latin,借助该语言编写的脚本可以被自动转化为 MapReduce 作业;Hive 是一个开源的数据仓库解决方案,提供了 SQL-Like 的陈述性语言 HiveQL,支持类似 SQL 的海量数据查询方式,查询被编译成 MapReduce 作业在 Hadoop 上执行. 两者都简化了编写 MapReduce 作业的代价,但是这种简便是以牺牲执行性能为代价的. 此外还有 Hyracks[17]、Spark[18]、Nephele[19] 等多个受 MapReduce 启发的海量数据处理系统,上述并行处理系统都是高通用性的并行平台,并没有针对分布式并行图算法的特点设计高效的支持方法,因此处理效率都不理想.

针对传统的 MapReduce 不易于表达迭代式操作的问题,Bu 等人[4] 设计了 HaLoop 一种基于 Hadoop 的支持内部迭代的数据并行处理系统,有效减少了迭代过程中数据重载以及迭代中间结果持久化的开销,但是无法消除中间数据排序以及重复任务启动的代价. 相比于 Hadoop,Twister[5,20] 是一个研究性实验项目,其基于 MapReduce 思想设计了支持迭代的并行编程模型,但该系统假设待处理数据不需要分布式文件系统支持可完全加载于分布式内存,导致其不满足海量数据处理的实际应用需求,且其输入数据手动切分的策略也增加了开发者设计并行应用的难度. Apache 的开源项目 Mahout[21] 其设计目标是基于 Hadoop 创建高可伸缩的机器学习算法,几乎所有的机器学习算法都涉及迭代的过程,Mahout 专门设计了外部驱动程序来控制迭代的执行,每轮迭代都需要启动新的 MapReduce 作业,如引言中所述,这种外部迭代的方式会引入很多不必要的执行代价.

目前专门面向大规模图算法的分布式并行编程模型及相关优化的研究工作也有很多成果. Google 的 Pregel[22] 编程框架能够为图算法提供并行支持,其根据图的特点是设计了顶点传递信息的多轮迭代处理模式. 但该平台并不基于 Hadoop,在开放性和通用性上均有所限制,并且其所有的计算状态均保存于内存,因此也缺乏对大规模数据的有效支持. Avery Ching 等人提出了一种基于 Hadoop 的大规模图处理框架 Giraph①,该框架支持动态资源管理,利用高可用容错的 ZooKeeper 实现系统工作单元的分布式协调,并提供了支持迭代的图形处理库,其和本文设计的编程框架都已期在 Hadoop 平台上利用大同步编程规范提升迭代式图算法的处理能力,但在设计策略和实现细节(特别是消息处理)上有诸多不同,相对于 Giraph 构建于 Hadoop 之上的多线程架构,本文设计的框架采用了侵入式的设计模式,利用通信实现工作节点间的分布式协调,无需手工切分和分布输入数据等,因此在兼容性、易用性和通用性等特性上更具优势. Surfer[23] 系统提供了

① Giraph. http://incubator.apache.org/giraph/

MapReduce原语和 Propagation 原语，并利用基于原语的构建块来支持在云上的大图算法，其主要目标在于提供运行期的可视化监控，并没有涉及针对图特征的具体实现细节. Lin 等人在文献[24-25]中对 MapReduce 的图算法实现进行局部优化，提出了 Mapper 内合并、避免图拓扑重复传递以及范围分区等优化技术，但其还是基于多轮的 MapReduce 作业调度. 还有很多并行处理框架，像 Apache 的 HAMA[26] 和 CMU 的 GraphLab[27] 等也都支持迭代，但是这些平台均面向特定的问题领域.

此外还有很多研究[28-29]是希望借鉴并行计算中的成熟的消息传递接口(Message Passing Interface，MPI)技术提升 MapReduce 的处理能力，但这些研究都没有给出基于 Hadoop 平台的实现，在容错、可扩展、鲁棒性等特性上都存在缺失. 本文旨在继承 Hadoop 原有的诸多特性的基础上通过引入大同步模型，利用消息传递机制和超级步同步来更高效地支持分布式图算法.

3　并行编程模型抽象和代价分析

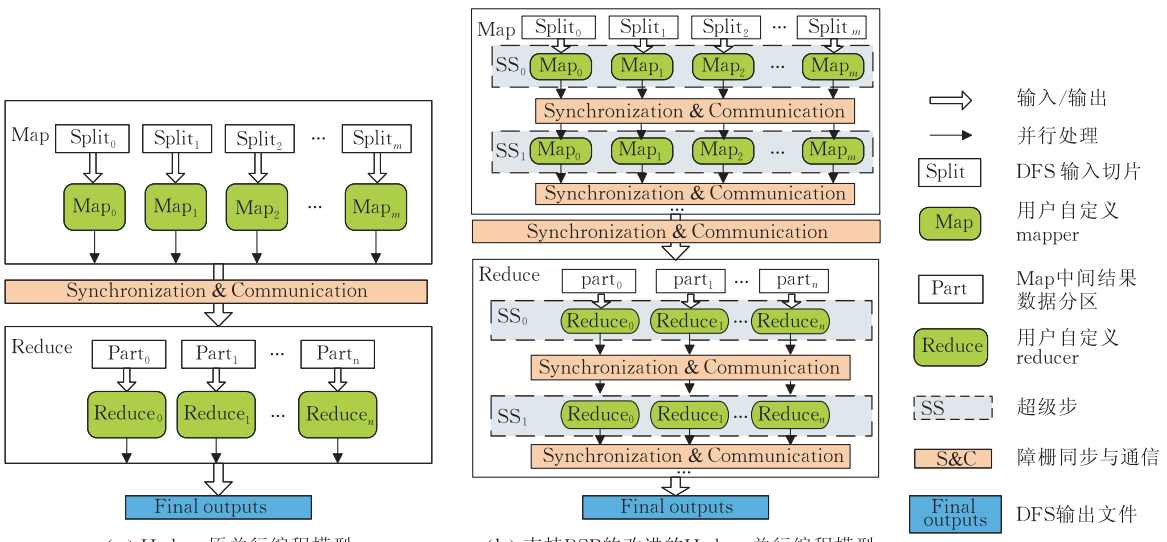
本节将介绍 MapReduce 支持的既有计算模式和本文改进的计算模式，并分析了改进模式的相关代价. 同时抽象出基于稀疏有向图的并行算法在 MapReduce 并行编程环境下迭代执行的设计模式.

3.1　易并行(EP)计算

并行计算可以用多种不同的并行编程模型表示和实现，每一种模型都有与其相适应的一类计算应用. 为简化并行应用开发者的开发工作，使无并行开发经验的程序员也可以正确快速地编写并行应用，MapReduce 对编程模型进行了限制，使 MapReduce 编程模型主要针对易并行(Embarrassingly Parallel，EP)计算抽象. 正是利用这种限制性的模式，MapReduce实现了并行程序的自动并发处理，并在内部提供了诸如输入划分、并行任务调度及通信、容错、负载均衡等并行细节的自动支持，实现了高可扩展性和高度并行性. 下面先给出易并行计算的形式化定义.

定义 1(易并行计算)　并行计算中，假定给定一个并行作业 J，其可以分解为一系列可异步并行执行的任务 T，同时给定一个输入数据集 W 和并行任务间的通信代价 C，如果对输入数据集 W 的任一划分 $\forall P=\{p_1,p_2,\cdots,p_m\}$，若其满足 $p_i\cap p_j=\varnothing$，$\bigcup\limits_i p_i=W$ 以及 $C(T(p_i),T(p_j))=0(1<m<|W|,1<i,j<m)$ 条件时，并行作业 J 可表示为 $J=\bigcup\limits_{i=1}^{m}T(p_i)$，称满足以上条件的并行计算为易并行计算.

也就是说，若干并行任务可以在互斥的输入集划分上无通信代价地独立执行. 下面利用易并行的形式化定义分析 MapReduce 的编程模型. 从图 1(a)中可以看到，MapReduce 作业执行计划分为 Map 和 Reduce 两个阶段，也称之为一次 MR 过程，MR 过程中每个阶段内部的异步并行任务($Map_0\sim Map_m$或者 $Reduce_0\sim Reduce_n$)都运行在易并行的理想模式下.



(a) Hadoop原并行编程模型　　(b) 支持BSP的改进的Hadoop并行编程模型

图 1　并行编程模型

在 Map 阶段,输入数据被自动切分成等大小的独立输入片段 $Split_0 \sim Split_m$(Split 的默认值是 64 MB,和底层分布式系统存储块等大小,这种策略避免了 Split 跨越块边界而可能引发的数据传输导致的网络代价),输入片段是若干键值对构成的集合. MapReduce 并行处理框架会依据数据本地化优化策略将 $Map_0 \sim Map_m$ 分布到输入片段所在的执行节点上运行. 执行过程中 $Map_0 \sim Map_m$ 之间不存在任何依赖关系无需通信交互,符合易并行的形式化定义,其产生的中间结果是新的键值对集合.

在 Reduce 阶段,由 $Map_0 \sim Map_m$ 产生的中间结果经过按输出键分区操作后(默认是采用散列分区的方式,在输入数据集的分布不偏斜的情况下,散列能得到分散均匀的中间结果分区)产生的中间结果分区 $Part_0 \sim Part_n$ 作为 $Reduce_0 \sim Reduce_n$ 的输入. 执行过程中 $Reduce_0 \sim Reduce_n$ 之间不存在任何依赖关系无需通信交互,符合易并行的形式化定义,最终 Reducer 的输出结果会自动被持久化到底层存储介质上(默认是 HDFS).

Map 阶段和 Reduce 阶段之间是串行同步的,存在一个相对用户透明的隐式同步和通信过程. Reducer 必须等到最后一个 Mapper 执行完毕后才开始执行(但是,其中 Mapper 产生的中间数据的 Shuffle 过程是与 Mapper 以重叠方式执行的,即任一个 Mapper 结束后,Reducer 就可以 Shuffle 中间结果,这样可以缩短并行流水处理的长度,提高处理的效率). 如果一个并行应用需要使用多趟的 MR 过程,那么一次 MR 过程与下一次 MR 过程之间也是链式串行同步的. 通信和数据交互也仅发生在一次 MR 过程中的 Map 阶段和 Reduce 阶段之间以及多趟 MR 过程之间.

大量实际应用属于易并行计算模式,如分布式的 Grep、倒排索引以及分布式排序等,这类问题非常适合用易并行计算模式处理. 然而,易并行计算是整个并行计算设计体系中最理想的一类计算问题. Sutter 等人[30]将并行计算模式分为 3 种类型:无依赖并行(Independent parallelism)、规则并行(Regular parallelism)和无结构并行(Unstructured parallelism). 其中无依赖并行即是前文讨论的易并行,规则并行是比易并行更高级的并行模式. 规则并行适用于并行计算工作存在一定依赖、在并行操作间有通信或同步需求的并行应用类型. 本文希望通过引入消息传递机制扩展 MapReduce 支持的并行模式,使 MapReduce 以更优雅更高效的方式支持诸如 PageRank 等大量的规则并行应用.

3.2 大同步(BSP)计算

引入消息传递机制的主要挑战在于在现有的 MapReduce 并行计算框架中,Mapper 之间或 Reducer 之间不支持消息传递. 本文中 Map(或 Reduce)阶段内部支持消息传递的灵感来源于并行计算的 BSP 模型,BSP 模型相对现有的 MapReduce 提供了更高级别的并行抽象,利用障栅实现同步控制,使用消息传递机制完成并行任务间数据的交互. 下面先给出 BSP 模型的定义.

定义 2(大同步计算). 大同步计算模型中,一个并行作业由一系列的超级步(Supersteps)组成,每个超级步构成一个相并行(Phase Parallel). 大同步计算主要由 3 个有序的部分构成:(1)易并行计算. 超级步内各任务独立的异步并行执行;(2)通信. 并行任务在超级步结束前利用消息传递机制完成数据的交互;(3)障栅同步. 同步等待同一个超级步内所有并行任务的交互全部完成,则整个并行才可以向下一个超级步移动,进入下一轮的相并行.

从图 1(b)中可以看到本文设计的支持 BSP 模型的 MapReduce 并行处理框架和原始的 MapReduce 并行处理框架在整体框架逻辑上保持一致. 从宏观的角度观察,如果将一次 MR 过程的整个 Map 阶段视为一个超级步,整个 Reduce 阶段视为另一个超级步,两个阶段间的 Shuffle 过程视为超级步间的障栅同步和通信过程,那么其实 MapReduce 本身也是符合大同步计算的,这也是为什么通过链式的 MapReduce 作业调度可以满足有交互需求的迭代运算的实现依据.

BSP 模型中,利用消息传递机制并行任务间可以交互变化的中间状态数据,并行任务间的消息传递不是任务间单独分散的行为,而是被视为一个整体并约束在相邻超级步之间. 任务发送的消息需要在一个超级步内进行汇集,上一个超级步汇集的消息需要等到下一个超级步内并行任务执行时才能使用,消息不能跨越不连续的超级步使用. 利用消息传递,同样的并行逻辑可以使用交互的数据在一个新的状态下继续运行. 迭代的计算都可以抽象为相同的处理逻辑,在新的状态数据集中期望收敛地连续执行,因此这种模式非常适合存在迭代需求的分布式并行图算法. 并且这种不跨越超级步的消息传递模式也有效简化了大量并行任务间消息的维护代价.

要保证消息能以整体汇集的形式有序地在超级步间传递,就必须依赖超级步间有效的同步机制. 在

BSP 模型中，超级步之间的同步等待是借助障栅[31]来实现的. 障栅是一种可控的粗粒度级的全局同步机制，利用障栅可以将一趟并行任务划分为多个连续的松散同步的超级步，如图 1(b)中的 SS_0、SS_1 等. 其保证了消息仅在一个超级步之内汇集，并在相邻的后继超级步间传递，关于障栅的概念在第 5.1 节中有更详细的介绍.

本文改进的并行处理框架试图在 Map 或 Reduce 阶段内部支持超级步，基于这种设计模式，以往需要通过多趟 MapReduce 作业外部链式调用才能实现的迭代式计算，现在可以在一次 MR 过程中，利用 Map 阶段内部(或 Reduce 阶段内部)的多个超级步的同步执行就可以完成. 复杂的消息传递控制由新的运行时系统处理(具体实现细节参看第 5 节)，并行程序开发者只需要利用原有的 MapReduce 程序开发经验就可以在改进的并行框架下编写更高效的并行应用. 改进的并行框架有效减少了占据大量处理时间的外部迭代引入的代价. 但是，相对于原有的 Hadoop 并行计算框架的代价[32]，改进的支持 BSP 模型的并行计算框架也引入了一些新的代价：首先，粗粒度的障栅同步使得单个超级步的总体执行时间对单个最慢完成任务是敏感的，针对非一致性状态下任务完成时间异常的问题可以借助 Hadoop 的推测执行(Speculative Execution)机制[1]，利用冗余执行的备用任务有效缓解，并且由于一个超级步内并发的多个任务的计算和通信是重叠执行的，所以此代价还能进一步地被一个超级步内异步并行的多个任务摊销；其次，新代价模型中的障栅同步是一个潜在的可能会造成性能下降的瓶颈，但是，实际上改进模型中的障栅同步只是将原模型中多次 MR 过程间 Map 和 Reduce 阶段的隐式障栅同步转移成一次 MR 过程中 Map 和 Reduce 阶段内的多次障栅同步，所以本质上并没有增加任何新的同步代价.

4　BSP 模型下图算法的计算模型

本节根据新引入的 BSP 模型，介绍如何在此模型下进行高效的图算法的设计和实现，包括图如何在改进的并行计算框架中表示，以及如何建立以节点为驱动的图并行计算模型，并以 PageRank 为典型应用给出具体的应用示例.

4.1　图在并行计算框架中的表示

基于 BSP 模型的并行计算框架主要意图在于利用图的特点以更高效的方式支持分布式图算法. 众所周知，在单机运行环境中往往难以满足图运算(特别是大图)对时空开销的需求，MapReduce 并行计算框架虽然可以满足分布式图运算对可扩展性的需求. 但是由于其无状态并行的任务没有利用节点的依赖关系，所以不易直接表达图算法. 把图运算放入并行计算框架中执行首先需要把图剖分成适合 MapReduce 处理的输入键值对集合. 邻接矩阵和邻接链表是两种最常用图的表示形式. 因为实际应用中的大图常呈现出典型的稀疏特性[22]，如社交网络图、网页链接关系图等，所以相对于 $O(n^2)$ 空间需求的邻接矩阵形式，邻接链表更适合稀疏大图的表示需求. 下面以邻接链表为图的基本表示形式，进一步构造适合改进框架的图的键值对表示形式.

假设有向图 $G=(V,E)$(不失一般性的情况下，下文讨论均以有向图为例，无向图可以用一对节点间互相指向的有向边表示成有向图)，其由顶点集 $V(G)=\{v_1,v_2,\cdots,v_n\}$，和连接两个顶点的边集 $E(G)=\{(v_i,v_j)\mid i,j=1,2,\cdots,n\}$ 构成. 其中与一个顶点直接相邻的图的局部拓扑结构可以用该节点的直接前驱集合和直接后继集合表示，先给出直接前驱集合和直接后继集合的形式化定义.

定义 3(直接前驱集合).　设节点 $v_i\in V$ 是有向图 $G=(V,E)$ 的一个节点，若图中有节点 v_j 满足 $(v_j,v_i)\in E$，则集合 $\chi^p(v_i)=\{v_j\mid(v_j,v_i)\in E\}$ 就是节点 v_i 的直接前驱集合.

定义 4(直接后继集合).　设节点 $v_i\in V$ 是有向图 $G=(V,E)$ 的一个节点，若图中有节点 v_j 满足 $(v_i,v_j)\in E$，则集合 $\chi^s(v_i)=\{v_j\mid(v_i,v_j)\in E\}$ 就是节点 v_i 的直接后继集合.

直接前驱集合和直接后继集合代表了基于节点的并行计算任务间的依赖，也是消息传递的路径. 我们在基于邻接链表的基础上通过适当扩充变形，抽象出在改进框架下使用的有向图的输入键值对表示格式，如图 2 所示.

Key	Value				
键名	节点标识符 (v_i)	直接前驱集合 ($\chi^p(v_i)$)	直接后继集合 ($\chi^s(v_i)$)	元数据 (md)	节点状态 ($sv(v_i)$)

图 2　改进并行计算框架下图的键值对表示

图 2 中输入键可以是任意内容(默认的键是文本行起始处的偏移量)，输入值分为 5 个基本部分：节点标识符、直接前驱集合、直接后继集合、元数据和当前节点状态. 其中元数据是图元素代表的实体

关键信息,包含节点的元数据和边的元数据两部分.例如,在交通网络图中,节点代表道路交叉口,其中节点元数据可能包括位置坐标、所属区域以及节点名称等信息;边代表节点间连通路段,其中边元数据可能包括路段长度、断面通行能力、单向通行与否等信息.当前节点状态是随着迭代运算不断更新的当前状态值,其代表本次迭代的节点中间状态,当迭代收敛时,中间状态值就成为图运算的结果状态值.

4.2 图的计算模式

下面抽象出在改进框架下,以节点为驱动的包含迭代过程的稀疏有向图的计算模式.每一次迭代过程包含相同的处理逻辑,其包括以下主要处理步骤:

1. 节点驱动的功能函数启动处理.如前所述,图经过预处理被剖分为输入键值对集合,每一个键值对即代表以节点为中心的计算元,用户设计 Map 处理逻辑根据应用需求利用键值对中包含的信息计算首轮迭代中节点的中间状态值;

2. 节点间状态的交互处理.利用消息传递机制,将节点的中间状态值依据节点在图中的邻接关系进行传递,邻接关系即用节点的直接前驱集合和直接后继集合完全表示;

3. 节点驱动的功能函数迭代处理.接受并解析上一个超级步传递的消息获取邻接节点的新的中间状态值,在新的邻接节点中间状态值集合和代表图拓扑的原始输入键值对集合上,应用用户设计的 Map 处理逻辑计算本轮迭代中节点的中间状态值;

4. 迭代终止检测.根据具体应用的迭代终止条件决定是返回步 2 继续执行迭代处理,还是停止迭代返回计算结果.系统可以指定两种迭代终止条件,一是比较相邻超级步间的结果误差是否小于指定阈值,二是迭代次数是否达到设定上限.

由上述描述可知,基于 MapReduce 的图的计算模式的核心内容是基于节点的异步并行计算和基于邻边的同步消息传递.

4.3 实例:PageRank 的实现

现实中的很多问题都可以转化为图来处理,如以社交网络为代表的诸多应用,下面就以 PageRank 为改进框架中的典型应用展开介绍.PageRank[33]是一种用于搜索引擎的基于超连接结构测度网页质量的算法,其是有交互和迭代需求的图算法中最具代表性的例子.假定 Web 网表示为图 $G=(V,E)$,其中节点 $v_i \in V$ 代表网页,$PR(v_i)$ 是该节点代表的网页的 PageRank 值,其表示浏览到 Web 图中页面 v_i 的可能性,这个可能性与 Web 图的拓扑结构高度相关,例如指向它的页面的状况(即直接前驱集合),也就是说一个页面的 PR 值是由其它指向页面的 PR 值计算得到的,一个基本的 PageRank 计算公式可表示为

$$PR(v_i)^r=\begin{cases} s_0, & r=0 \\ \dfrac{1-q}{|V|}+q\sum\limits_{v_j\in\chi^p(v_i)}\dfrac{PR(v_j)^{r-1}}{|\chi^s(v_j)|}, & r>0 \end{cases} \quad (1)$$

其中,r 表示迭代的轮次,s_0 是节点的 PageRank 初始值,q 表示阻尼系数(详细的介绍可参阅文献[33]).在给定每个节点一个随机的 PageRank 初始值 s_0 的情况下,经过多轮迭代计算,节点的 PR 值会趋向收敛.

在 MapReduce 并行处理框架下计算 PageRank,首先需要将图转化为键值对集合,图最简单的表示形式形如二元组〈$FromVertexID$, $ToVertexID$〉,元组元素表示有向边关联的端节点.这种最原始的图表示形式仅需通过两次简单的 MR 过程就可以转化为图 2 所示的通用形式,其中状态值代表节点的 PR 值.

算法 1. 基于原始 MapReduce 的 PageRank 算法.

```
Map 阶段:
//图分解为图 2 所示的键值对 G=(V,E)→[(key, value)_1…]
//parse: 解析键值对的函数
//countPageRank: 计算节点 PR 值的函数
1. MAP (Key key, Value value)
2.   v_i=parse(value); χ^s(v_i)=parse(value);
3.   PR(v_i)=countPageRank(value);
4.   foreach v_j∈χ^s(v_i) do
5.     output (key v_j, value PR(v_i));
6.   end foreach
7.   output (key v_i, value value);//传递原图拓扑
Reduce 阶段:
1. REDUCE (Key v_j, Value [w_1,w_2,…])
2.   newPagerankSet←∅;
3.   foreach w_i∈[w_1,w_2,…] do
4.     if w_i∈χ^p(v_j) then
5.       newPagerankSet=newPagerankSet+w_i;
6.     else
7.       value=w_i
8.     end if
9.   end foreach
10. PR(v_j)=countPageRank(value,newPagerankSet);
11. value=update(value,PR(v_j), newPagerankSet);
12. output (key v_j, Value value).
```

上述是一次 MR 的处理过程,运算需要多次 MR 过程迭代执行,并且需要在两次 MR 过程间增

加收敛判断来确定何时终止迭代过程.

值得关注的是，除了更新的节点中间状态值(本例中即 PR 值)需要借助 Shuffle 机制在不同执行节点间传递，为保证下一次迭代的执行，运算过程中并没有改变的整个图的拓扑(如 $\chi^p(v_i)$、$\chi^s(v_i)$ 以及 md 等)也需要在每次 MR 过程中反复传递和持久化. 实际图运算过程中未改变的部分(图 2 中浅色部分)比例远大于变化的中间状态值集(图 2 中深色部分)，如式(2)所示. 所以，多次 MR 过程中这个代价相对于整个计算过程而言是非常大的.

$$\frac{\left(\sum_{v_i\in V}(\chi^s(v_i)+\chi^p(v_i)+md(v_i))+\sum_{e_i\in E}md(e_i)\right)}{\sum_{v_i\in V}sv(v_i)}\gg 1 \tag{2}$$

所以，MapReduce 虽然可以执行图的运算但是并不易于表达这种有数据交互依赖和迭代需求的计算. 改进的并行框架下图的计算则充分利用图本身的拓扑特征和运算特点，根据图计算内在的迭代需求，利用 Map 和 Reduce 阶段内的多个同步的超级步完成迭代. 改变的中间状态值在超级步间通过消息传递，未变化的图的拓扑无需传递和持久化. 对于 PageRank 算法甚至仅需 Map-only 的方式即可完成. 下面给出新并行计算框架下 PageRank 分布式算法的实现伪代码.

算法 2.　基于支持 BSP 模型的 MapReduce 的 PageRank 算法.

```
Map 阶段-setup 操作：
1. i=superstepCounter++; //超级步计数器
2. Msg_i=new userdefineMessage();
                                    //新建空自定义消息体
3. resetInputDataOffset(); //重置输入数据访问偏移
Map 阶段-map 操作：
1. MAP (Key key, Value value)
2.   v_j=parse(value); χ^s(v_j)=parse(value);
3.   if (stopflag==false)
4.     if (i==0) //首次迭代，执行启动处理
5.       PR(v_j)=countPageRank(value);
6.     else //执行迭代处理
7.       PR(v_j)=countPageRank(value,newValueSet_{i-1});
8.     end if
9.     Msg_i.setStateValue (PR(v_j));
10. else //终止迭代，输出
11.     value=update (value, newValueSet_{i-1});
12.     output (Key v_j, Value value);
13. end if
Map 阶段-cleanup 操作：
1. if (compare(Msg_{i-1}, Msg_i)>threshold)
                                    //迭代终止检测
2.   header_i=createMessageHeader(i,timestamp);
3.   barrierMsg_i=assembleMessage(header_i, Msg_i);
4.   send(barrierMsg_i); //发送消息，执行交互处理
5.   sleep(); //陷入等待状态
6.   MsgSet_i=receive(); //被唤醒后接受汇集消息
7.   newValueSet_i=getNewValue(MsgSet_i);
                                    //获取消息携带的新 PR 值集合
8. else
9.   stopflag=true; //设置迭代终止标志
10. end if
```

setup 和 cleanup 操作仅在每次超级步的任务运算前和运算后执行，map 操作则循环处理输入数据切片上的键值对. 此外，新系统还提供了继承 Mapper 类的新超类 MsgMapper 和若干方法来支持在框架内部完成迭代处理，迭代控制过程对用户而言是透明的.

5　基于消息传递的系统框架设计与实现

本节介绍基于 BSP 模型的改进版并行计算框架所支持的通信协议和基于开源 Hadoop 0.20.2 版本的具体实现细节. 系统引入的两种自适应的消息传递机制可以高效地处理图运算的中间状态信息的交互.

5.1　障栅消息

障栅(Barrier)是可用于消息传递系统的一种有效的同步机制. 对于计算过程中有数据依赖且不能完全独立执行的计算任务，其需要借助消息传递数据. 为保证消息整体有序地传递，同一超级步内的异步并行的计算任务在发送消息后需插入障栅同步等待. 插入障栅的任务会陷入等待状态，直到该超级步内所有任务通过消息完成数据的交互，然后此超级步内所有被障栅阻隔并陷入等待状态的任务才能被重新唤醒并利用接收到的汇集消息中的新数据在下一个超级步内继续运行.

支持 BSP 模型的 MapReduce 并行计算框架，需要利用障栅将 MR 过程中的 Map 阶段和 Reduce 阶段分割成同步执行的若干超级步. 一个超级步内异步并行的多个 Mapper 或者 Reducer 在发送消息后借助障栅互相等待，本文称这种在一个超级步内被障栅分隔的由任务发送的消息为障栅消息. 障栅

消息仅能在相邻的超级步间进行传递，超级步 i 中任务发送的消息经过汇集处理可以被超级步 $i+1$ 中的任务使用. 也就是说，超级步内任务的执行完全异步，超级步间所有任务通过障栅同步，并借助障栅消息交互数据.

为支持消息的传递，系统新增了预置的消息接口，用户可以根据应用的需求灵活地设计实现该接口的自定义消息类型. 这种可插入式的消息设计模式保证了运行的 Hadoop 集群可以在不重启的情况下动态支持任意类型的用户消息，只要用户实现了系统预置的消息接口. 预置的消息接口中主要设置了系统用于维护障栅消息所需要的消息元数据，而用户自定义消息的具体内容则依赖于应用的需求. 障栅消息格式主要由两部分组成：消息头和消息体，每一部分的具体内容如图 3 所示.

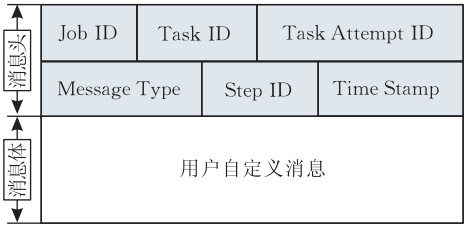


图 3 障栅消息格式

消息头包含的是障栅消息的元数据，其由 6 部分构成(图中灰色部分). 其中 Job ID、Task ID 和 Task Attempt ID 三者构成了具体任务(Mapper 或 Reducer)在集群中的唯一标识. 用户编写的并行应用以作业(Job)为单元提交到 Hadoop 集群中执行，Job ID 即集群中该作业的唯一标识，Task ID 即任务在作业内的唯一标识，由于推测执行机制，同一个输入片段可能同时会在不同工作节点上存在多个完全相同的冗余任务，Task Attempt ID 就是用以区分冗余任务的唯一标识. 所以，集群内一个具体执行计算的任务可以用〈*Job-ID*，*Task-ID*，*Task-Attempt-ID*〉三元标识组合来唯一确定. 障栅消息和发送消息的任务绑定，因此该三元标识组合同时也可被系统用于区分超级步内的障栅消息. Message Type 用于区分发送消息的任务类别，包含 Map 任务和 Reduce 任务两大类. Step ID 用于区分消息隶属的超级步，因为 Map 或 Reduce 阶段被分为一系列同步的超级步，每一个超级步都伴随着消息的传递和交互，一个超级步中的消息仅在此超级步内进行汇集，不同超级步内的消息不能混和汇集，这种同步方式简化了消息维护的代价.

消息体是由用户根据具体应用需求所构造的可序列化的消息实体，其可以是基本数据类型(如整型或实型)，也可以是复杂的复合类型(如容器类型或构造类型)，用户可以完全控制消息内容的格式. 因为消息需在集群中通过网络传递，所以必须强制要求用户按照 Hadoop 集群序列化的要求实现自定义消息内容的序列和反序列化逻辑. 借助 Hadoop 提供的各种序列化操作基元，可以很容易地实现自定义消息体的序列和反序列化.

消息头是由系统自动生成和维护的，消息体发送时系统会自动为其添加相应的消息头，然后完整的消息再被序列化为可在网络中传递的字符序列，在集群的工作节点间传递. 此外借助动态加载技术可以让运行中的集群系统在不停机的情况下自动加载用户创建的自定义消息类型.

5.2 轻量级消息传递机制

轻量级的消息传递机制旨在利用 Hadoop 现有的消息传递机制，在不影响既有功能的情况下，通过修改和新增部分通信协议以实现小体量的消息传递. 图 4 描述了轻量级消息传递机制的实现框架.

先介绍 Hadoop 既有的通信机制以提供技术实

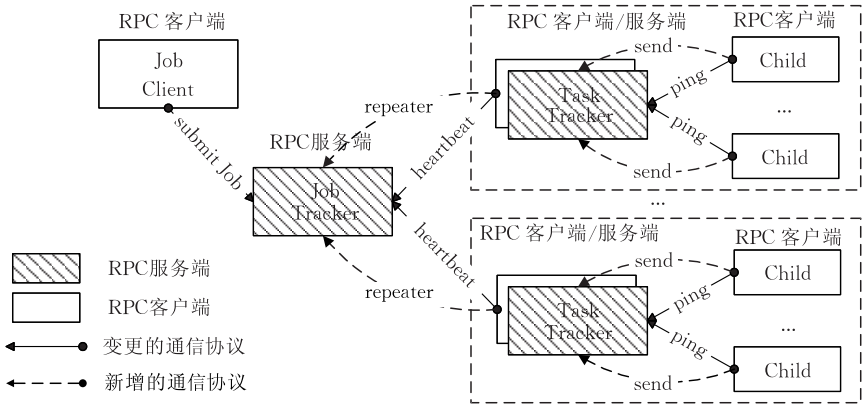


图 4 轻量级消息传递实现框架

现的背景. Hadoop 分布式并行计算系统是典型的主从(Master/Slave)架构，如图 4 中所示集群由一个 JobTracker 节点和若干个 TaskTracker 节点组成. 用户作业通过 JobClient 向 MapReduce 集群投交，JobTracker 负责接受作业的投交请求，将作业自动分解为多个并行的任务，并依据数据本地化原则尽可能的将任务调度到输入数据所在的 TaskTracker 节点上执行. TaskTracker 节点会根据分配任务的类型启动独立的子进程 Child 执行 Map 任务或者 Reduce 任务. JobTracker 同时还负责监控和维护整个集群中所有 TaskTracker 节点的运行状态以及投入作业的执行情况.

状态信息及任务调度等信息主要籍由节点间的通信来传递，集群中节点间的通信是使用 Hadoop 的 RPC(远程过程调用)机制来实现的，各节点利用 RPC 协调各自的运行状态确保集群流畅的运作. TaskTracker 节点通过 RPC 协议向 JobTracker 发送周期性的心跳(图 4 中所示的 heartbeat)来通知自己的当前状态，并根据自身的负载能力请求新的任务；执行具体任务的 Child 子进程也会通过 RPC 协议向 TaskTracker 节点发送周期性的连通指令(图 4 中所示的 ping)，利用该指令 Child 子进程查看作为父进程存在的 TaskTracker 节点的当前状态来确定继续还是终止任务的执行；JobClient 则通过 RPC 协议向 JobTracker 发送投交指令(图 4 中所示的 submitJob)提交用户作业.

发起 RPC 请求的作为 RPC 客户端，而接受 RPC 请求并执行处理逻辑的作为 RPC 服务端. 如图 4 所示，JobTracker 和 TaskTracker 都实现了 RPC 服务端接受不同的 RPC 请求，JobClient 和 Child 作为 PRC 客户端利用动态通信代理发送 RPC 请求，Child 对于 JobTracker 而言是透明的，没有直接的信息交互. 另外，TaskTracker 同时也作为 JobTracker 的 RPC 客户端与 JobTracker 进行信息交互.

为在集群原有的通信框架内支持任务间的障栅消息的传递，必须修改 Hadoop 既有的部分通信协议和新增个别专用于障栅消息传递的通信协议. 系统新增了两个通信协议：send 和 repeater. 其中 send 协议用于 Child 发送障栅消息给 TaskTracker，repeater协议则用于 TaskTracker 把局部汇集的障栅消息转发给 JobTracker. 系统修改和增强了原有的两个通信协议：heartbeat 和 ping. 其中增强后的 heartbeat 在保持原有功能的基础上可以通过心跳返回值定期的将 JobTracker 处理后的汇集障栅消息回带给发送心跳的 TaskTracker，ping 经过增强后可以将 TaskTracker 从 JobTracker 接受并缓存的汇集障栅消息转发给 Child. 接收到消息后，被障栅陷入等待状态的 Child 可以被唤醒，通过解析携带交互数据的信息进入下一个超级步的执行态.

轻量级消息传递机制的设计原则是：执行具体任务的 Child 之间不建立直接的通信，以转发和分发的形式建立通信层级，与任务绑定的障栅消息经 TaskTracker 转发后，统一在 JobTracker 中汇集，然后再分发给具体任务. 这样设计的原因在于：(1)首先 JobTracker 中维护有基于整个集群的全局的作业与任务、任务与执行节点等诸多映射信息，所以 JobTracker 可以利用这些全局信息维护和管理不同作业间、同一作业不同任务间、同一任务不同超级步间的障栅消息；(2)其次分布式系统中一个很重要的假设是节点失效是常态，也就是说集群中任务的执行有较大的不稳定性，直接维护 Child 之间的通信状态过于复杂，在高度并行的情况下也不可行. 所以这种层级的通信体系能确保障栅消息有序高效地传递.

5.3　重量级消息传递机制

利用 Hadoop 现有的消息传递机制实现的轻量级通信机制存在几个潜在的问题：(1) JobTracker 节点是整个 Hadoop 集群资源管理和任务调度的唯一主控节点，其也是整个 Hadoop 集群中潜在的瓶颈. 特别当作业密集投交时 JobTracker 的管理负担变得非常繁重，其计算和内存资源会更加紧缺，如果障栅消息不是小体量级的，那么维护汇集障栅消息的代价会加重 JobTracker 的工作负载，并且受限于轻量级通信机制的设计策略，消息的两层传递不可避免地会增加通信开销，进一步会导致集群整体吞吐量、作业完成时间等重要技术指标的下降；(2)其次利用原有的周期性心跳和连通指令回带汇集障栅消息的模式可能会产生一定的作业延迟. 因为集群规模的变化会影响心跳和连通指令的工作周期，因此当集群规模增长到一定程度时，即使集群中可用于执行任务的空闲工作节点增多，需要传递消息的作业执行时间反而会变得更长. 因此需要一种专用的模式能把障栅消息的维护工作从 JobTracker 节点中剥离出来，以更高效的方式维护整个集群中的障栅消息.

5.3.1　主要组件和功能

重量级消息传递机制就是为应对这些可能的问题而设计出来的工作模式. 系统新增了一个专门用

于维护障栅消息的障栅消息服务器(Message Tracker,MT),如图5所示.障栅消息服务运行于集群的独立节点中,该新增节点可以随Hadoop集群一起启动和关闭,也可以单独地启动和关闭而不影响集群既有的功能.借助新增的障栅消息通信协议,MT可以与原始的Hadoop集群高度协作,在完全兼容既有作业类型的基础上,同时提供对新的可传递消息的作业类型的支持.重量级消息传递机制的主要组件和功能如下(下面仅针对支持障栅消息的新作业类型展开讨论).

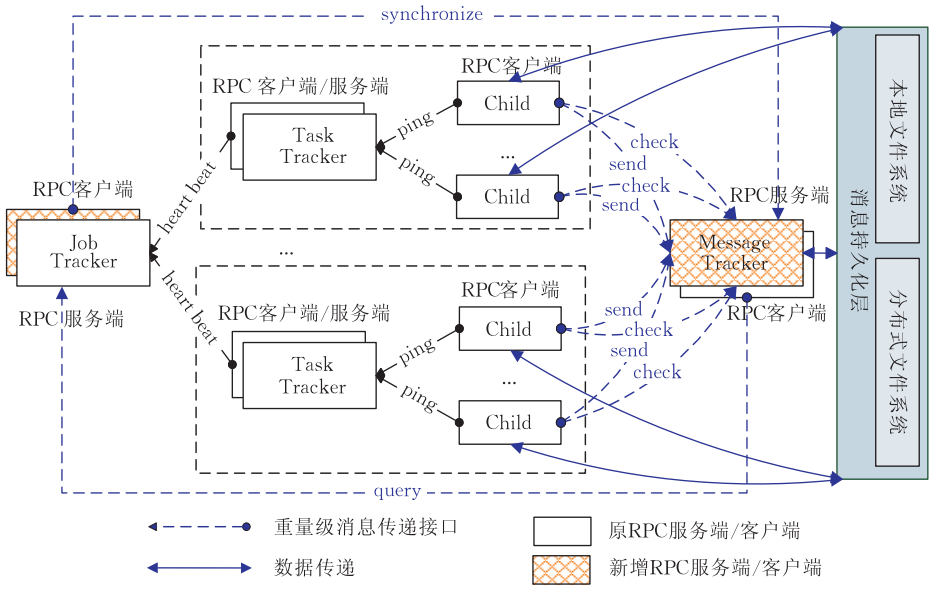


图5 重量级消息传递实现框架

(1)MT中建立了一个RPC服务器用于接收任务进程Child发送的障栅消息以及接受JobTracker发送的作业初始化以及用户自定义消息类型等同步数据;

(2)包含一系列的维护障栅消息的数据结构和技术,提供障栅消息的汇集、过滤和裁剪等核心功能的处理;

(3)MT中拥有JobTracker的RPC客户端用于与JobTracker建立通信,MT利用动态通信代理获取JobTracker所维护的各种与集群节点和作业相关的映射信息,从而保证MT与JobTracker所持有的集群状态的一致性;

(4)JobTracker中建立一个连接MT的RPC客户端,将用户投入作业的信息,特别是使用了障栅消息接口的新作业的用户自定义消息类型等信息主动同步到MT中.因为作业分配等主要的工作仍然由JobTracker完成,这部分操作产生的各种映射关系是维护障栅消息时所必须的.

(5)Child中建立一个连接MT的RPC客户端,用于在超级步内发送障栅消息以及以定时轮询的方式获取汇集障栅消息反馈.

5.3.2 主要执行流程

基于独立障栅消息服务器的重量级消息传递机制的主要执行流程如下(以MT与集群同步启动为例,主要描述与障栅消息相关的执行步骤,其他与原始作业相同的执行处理被省略):

(1)用户根据需求在配置文件中新增MT入口随后启动集群.

```
〈property〉
    〈name〉mapred.barriermsg.tracker〈/name〉
  〈value〉hdfs://host:port〈/value〉
〈/property〉
```

(2)开发者可使用新增接口setMTServerMode在编写并行应用作业时显式开启障栅消息服务器模式,默认设置是关闭.

(3)JobTracker和MT在等待各自的RPC服务端启动后,利用动态代理互相建立连接到对方的RPC客户端.

(4)JobTracker接受用户投交的支持障栅消息的新类型作业,利用新的synchronize协议把作业包所在的位置信息、作业的任务划分情况以及用户自定义障栅消息类型同步推送到MT中.

(5)任务执行子进程Child启动后利用动态代理建立连接到MT的RPC客户端,当任务执行到障栅插入点时,利用send协议将携带交互数据的用户自定义障栅消息发送给MT,同时任务执行线程陷

入到等待状态.

(6) MT 汇集任务发送的障栅消息，同时利用 query 协议保持与 JobTracker 任务执行状态的一致性.

(7) Child 子进程利用周期性的 check 协议询问 MT 当前障栅消息的处理状况，当同一作业的同一超级步内的所有任务障栅消息汇集完毕，则借助 check 协议回带处理后的汇集障栅消息给 Child 子进程.

(8) Child 收到障栅消息反馈后，唤醒处于等待状态的任务线程，被唤醒的任务线程解析收到的障栅消息，获取其需要的新数据同时结合原输入数据切片，推动任务在下一个超级步中继续迭代执行.

(9) 任务在进入下一次障栅点前执行用户定义的收敛检查确定任务是否满足终止条件.

5.3.3　一致性、容错处理及可扩展性

为维护障栅消息，MT 中需要持有 JobTracker 中所维护的若干映射和状态信息的一个副本，系统采用推拉协作的更新模式确保这些信息在 JobTracker 和 MT 中的最终一致性. 信息的更新一是采用客户端(MT)的按需请求模型利用 query 协议从 JobTracker 中用拉模式(pull)获取所需信息的最新副本；或者是采用服务端(JobTracker)急切更新(Eager Update)模型利用 synchronize 协议以推模式(push)将 JobTracker 中的关键更新及时同步到 MT 中，以期最小化不一致的时间窗口. 同时副本信息以只读的形式在 MT 中被访问也有效消除了资源竞争的问题. 一些重要的副本更新时机包括：作业投交、作业的任务切分、任务推测执行、作业完成、工作节点失效等.

新作业类型执行中的容错需利用原 Hadoop 集群的容错机制并辅以适当的增强. Master 节点会定期轮询集群中 slave 节点的状态，若存在无响应节点则重新调度分配该失效节点的任务到其他活动节点上重新执行，失效节点已完成的任务也需要重新执行因为任务产生的中间结果仅持久化到失效节点的本地存储中，当节点失效时执行结果也无法访问，若失效节点中包含有支持障栅消息的新作业类型的任务，则同时需要通知 MT 清除与这些任务绑定的障栅消息，等待重新执行的任务发送的新消息. 在重量级消息传递机制中，MT 也作为一个特殊的工作节点受控于 JobTracker，当 MT 失效时，JobTracker 会暂时迁移到轻量级消息传递工作模式下，所有支持障栅消息的运行中的任务都需要重新执行，因为汇集消息在 MT 中维护，当 MT 失效时这些汇集消息也无法访问(更细粒度的重做机制也可以定制实现，如指定从某一个超级步开始重做，因为篇幅原因不再详述).

扩展性有多方面的衡量指标，主要包括集群工作节点规模、处理数据规模、消息规模等. 本文在 6.3 节的实验环节中对改进框架在工作节点个数和原始数据规模的扩展性指标上进行了验证，实验结果表明框架在这些指标上具有较好的可扩展性. 目前本框架适用于传递消息的频度和量都比较小的稀疏图问题，因为在实际应用中稠密图是比较少的[22]，而且大多数图算法的迭代过程中仅需要交互一些信息量不大的状态数据. 当消息量和频度激增时，系统的工作性能会退化到原始的 Hadoop 链式调用水平，但是本文设计的系统具有足够的弹性可以进行有针对性的优化，例如采用传递消息元、压缩消息体等优化技术，这部分优化工作是我们下一步重点研究的内容.

5.4　障栅消息处理技术

所有的障栅消息维护工作都基于图 6 所示的障栅消息树(Barrier Message Tree，BMT)，BMT 是逻辑多叉树状结构，其由 4 个层次组成，包含：任务类型层、作业层、超级步层、任务层.

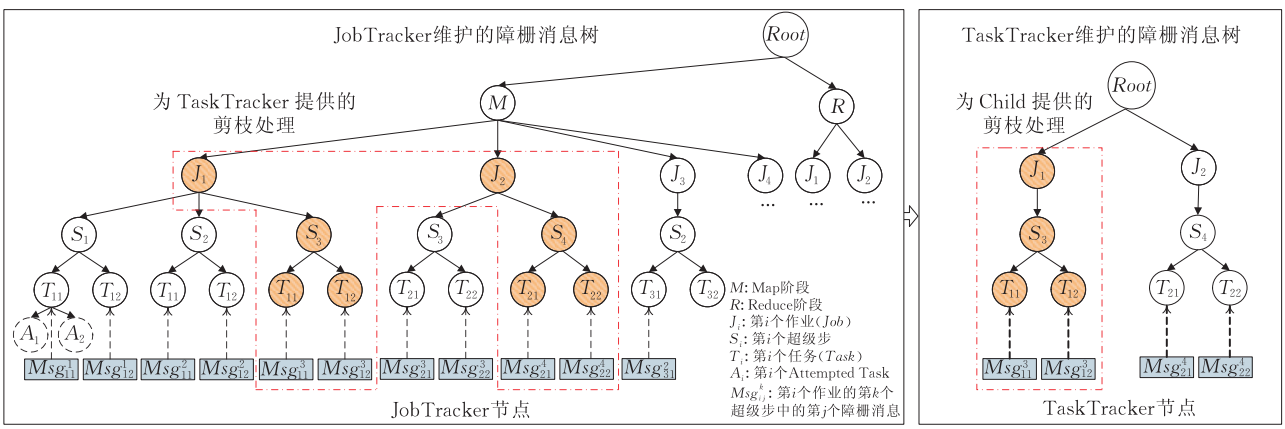


图 6　障栅消息树及剪枝示意图

先建立只有根节点的空树,当障栅消息维护节点收到任务发送的障栅消息时,用算法 3 建立 BMT,同时在叶节点上关联障栅消息,概要逻辑如下.

算法 3. BMT 的插入算法.

输入:障栅消息 barrierMsg

1. 解析障栅消息头 barrierMsg.header,获取 MessageType,JobID,StepID,TaskID,TimeStamp 以及 TaskAttemptID 等各部分的值;

2. 先根据 MessageType 定位到 BMT 中正确的任务类型层,然后根据 JobID 创建或者定位障栅消息所属的作业层,再根据 StepID 创建或者定位障栅消息所属的超级步层,并将 TimeStamp 关联到超级步节点上,最后依据 TaskID 建立叶子节点;

3. 将障栅消息中的消息体 barrierMsg.Msg 关联到叶节点上.

值得注意的是,具有相同 TaskID 不同 TaskAttempt ID 的消息只建立一个 TaskID 叶节点,因为冗余的推测执行任务产生的障栅消息是完全一样的,所以实质上障栅消息是和 TaskID 唯一对应的.

当作业执行完成、作业失效或者时间窗口过期时会自动触发 BMT 的删除逻辑,删除算法如下.

算法 4. BMT 的删除算法.

输入:JobID、过期时间窗口 w

```
1. if (作业完成 or 作业失效)
2.    依据 JobID 在 BMT 中进行定位;
3.    删除 JobID 为根的子树;
4. else //时间窗口过期
5.    foreach JobID 节点 in BMT
6.       if (不是最新的 StepID 节点
                and (now-TimeStamp)>w))
7.          删除 StepID 为根的子树;
8.       end if
9.    end foreach
10. end if
```

算法 4 中时间窗口过期是基于维护障栅消息的节点的内存容量和维护复杂性的限制.作业可能在不同超级步中迭代执行多次,当下一个超级步所属的消息被接收时,同时也表明之前超级步中汇集完成的消息已经过期,那么根据用户设定的过期时间窗口,可以删除以既往的超级步节点为根的子树.之所以不在新超级步节点创建时立刻删除旧超级步节点为根的子树,是基于细粒度容错的考虑,当任务执行发生问题时,可以基于超级步的粒度重新投入执行.

何时释放障栅以及回传 BMT 上哪些汇集的障栅消息是两个很重要的问题,这些问题在剪枝算法中解决.

算法 5. BMT 的剪枝算法.

输入:JobID,TaskTrackerID

```
1. S←∅,保存 TaskTrackerID 标识的 TaskTracker
   节点上当前运行的 JobID
2. O←∅,保存可反馈消息的 JobID 的集合
3. 获取发送心跳请求的 TaskTracker 节点当前运行
   中的作业集 S',S←S';
4. foreach job∈S do
5.    releaseFlag=true; //可反馈(释放障栅)标志
6.    在 BMT 中找到 job 节点下最新的超级步节点;
7.    foreach task∈job do
8.       if (叶子节未关联障栅消息) then
9.          releaseFlag=flase;
10.         break;
11.      end if
12.   end foreach
13.   if (releaseFlag==true)
14.      O=O+job; //所有叶节点都关联消息后该作业
                   所属任务的障栅可被释放
15.   end if
16. end foreach
17. foreach job∈O
18.   以此 job 的 JobID 为根,在 BMT 中剪下最新超级
      步节点下的子 BMT.
19. end foreach
```

假设 TaskTracker 节点中正在运行的作业包括 J_1、J_2 和 J_3,因为 J_3 不满足障栅释放条件,所以只有满足障栅释放条件的 J_1 和 J_2 的最新超级步节点被裁剪,图 6 中虚线框内显示了剪枝处理的结果.算法 5 描述的是 JobTracker 维护的 BMT 的剪枝算法,裁剪的子 BMT 回传给 TaskTracker,同理如图所示 TaskTracker 也会裁剪其维护的 BMT 回传给 Task.障栅消息的发送源会周期性地通过 ping 或者 check 协议向障栅消息维护节点发送请求,接收到请求后,BMT 的剪枝算法就会被触发来决定是否反馈以及反馈哪些消息.

整个消息的汇集、删除、裁剪等过程对于用户而言是完全透明的,由系统框架在后台自动处理,用户只需要关注自定义消息的内容和格式.

6 实验结果与分析

6.1 集群环境

实验在 72 节点构成的 Hadoop 集群[11]中进行评估,集群节点的硬件环境如表 1 所示.

表 1　集群硬件环境

	CPU	Memory/GB	Hard Disk	Network Interface	OS
Master Node	E5620 4(8) @ 2.4GHz	48	2×146GB SAS 15k 2×500GB SAS 7.2k	Gigabit Ethernet	CentOS 5.5
Slave Node	X3430 4(4) @ 2.4GHz	8	2×500GB SATA 7.2k	Gigabit Ethernet	CentOS 5.5

节点间通过 3 个交换机和千兆比特网卡连接，24 个节点为一组共置一个机架中，节点配置 Hadoop 0.20.2 版本以及本文支持 BSP 模型的基于该版本的开源代码实现的改进版.

6.2　数据集

实验主要采用标准的 StanfordLarge Network Dataset Collection 中的 soc-LiveJournal 作为测试 PageRank 的数据集，该数据集的主要特征如表 2 所示.

表 2　数据集的主要特征

DataSet	Nodes	Links	Meta-Data	Volume
soc-LiveJournal	4847571	68993773	0G	1G
LiveJournal-V1	4847571	68993773	14G	15G
LiveJournal-V2	4847571	68993773	29G	30G
LiveJournal-V3	4847571	68993773	44G	45G

soc-LiveJournal 是 StanfordLarge Network Dataset Collection 数据集中的有关社交网络的图数据子集. LiveJournal 是一个具有强大社交网络功能的博客站点，LiveJournal 数据集以〈*FromNodeId ToNodeId*〉二元组集合的形式提供该网络的高度抽象的有向图拓扑，其节点表示用户，边是用户间的好友关系. 由于其省略了图 2 中提及的节点或边的元数据，原始数据集的大小约为 1G，为保证实际应用时处理的合理性，实验以该数据集为基准，在不改变图拓扑结构的条件下以字符串形式增加了节点和边的元数据，形成了 3 个扩展的 LiveJournal 数据集 LiveJournal-V1～V3.

在实验的预处理阶段，需要将二元组形式的图转化为文中图 2 所描述的并行框架下的图的标准键值对形式，使用两个简单的原始 MapReduce 作业即可完成所有的转换工作，实验中并行处理框架的输入和输出都基于分布式文件系统 HDFS.

6.3　评估及结果分析

实验首先在固定规模的 LiveJournal-V1 数据集和 24 个节点上测试原始 Hadoop 和基于轻量级消息传递以及基于重量级消息传递的改进版 Hadoop 的总体执行性能，并用线性增加的迭代次数验证并行执行框架的可扩展性.

如图 7 所示，PageRank 分布式图算法在基于超级步内部迭代的改进版 Hadoop 上的总体执行性能明显优于基于链式调用外部迭代的原始 Hadoop. 并且随着迭代次数的增多，改善的性能幅度也随之增大，在第 20 轮迭代时，基于重量级消息传递机制的改进版 Hadoop 执行性能相比于原始 Hadoop 提高近 51%. 因为对于链式调用的 Hadoop 作业，每轮迭代都不可避免地产生作业启动开销、不变数据的序列化和网络传输开销、迭代中间结果的 HDFS 持久化开销. 相比之下，改进版的 Hadoop，利用障栅消息在超级步间仅传递在整个图数据中所占比例很小的一部分随迭代操作变化的状态数据，执行过程中未改变的图拓扑数据和节点与边的元数据作为任务的输入，仍驻留在执行任务的工作节点中不需要重新加载，在任务执行到下一个超级步时只需要重置输入数据的访问偏移即可. 并且由图 7 可以观察，随着迭代次数的增多，并行执行框架的整体执行性能表现出良好的线性可扩展性，这得益于 Hadoop 本身的高可扩展特性，且改进版基于 Hadoop 实现并继承了该并行计算框架的可扩展性.

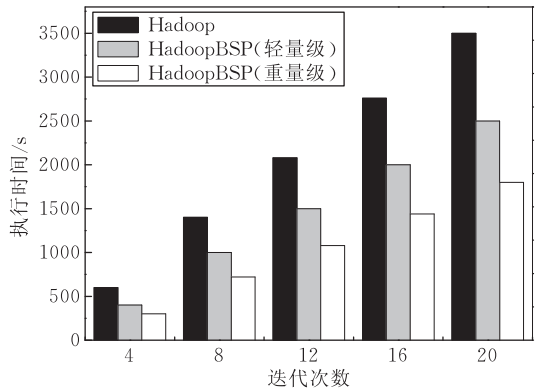


图 7　PageRank 总体执行性能(LiveJournal-V1，24 nodes)

同时还可以观测到重量级消息传递机制在整体执行性能指标上是优于轻量级消息传递机制的，这是由于轻量级消息传递机制受限于集群心跳和连通指令发送的间隔，为保证 JobTracker 节点的工作性能，指令发送不能过于频繁，最差情况是在消息汇集完毕后仍需等待整个间隔时间该汇集消息才能被回传到消息接收节点. 因此，随着迭代轮次的增加，这个间隔延时也会同步累积，并导致性能下降. 图 8 通过增加节点规模进一步验证了这个问题.

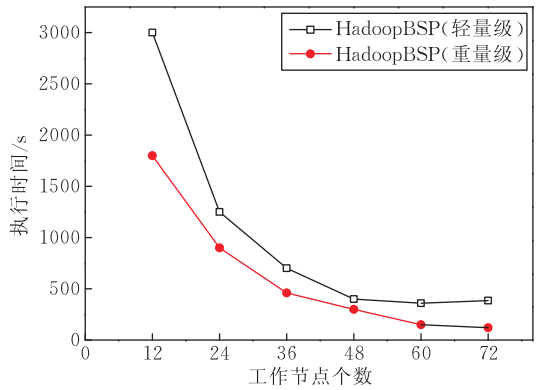


图 8 轻量级模式和重量级模式的可扩展性
(LiveJournal-V1,10 次迭代处理)

MapReduce 本身是一个线性可扩展的计算模型,也就是说 Hadoop 集群的处理能力理论上可随着集群节点数的增加实现近线性扩展.但是由图 8 中可以看出,当节点初始增加时总体执行时间线性下降,但是当集群节点规模增加到一定程度时,两种模式的总体执行时间的提升幅度都趋于减小,甚至对于轻量级消息传递机制,其总体执行时间在节点增加到 60 以后还有部分回升.其原因在于同时最大并发的任务数是由输入数据的切片个数决定的,当节点增加到一定程度时已经能满足特定大小的测试数据集最大并发性的要求,此后即使在集群中投入更多的空闲工作节点也无法再提升作业处理的并发度,总体执行时间也不会再进一步缩短.其次,对于轻量级消息传递机制的执行时间回升现象是因为,当集群节点数增加并且 JobTracker 的压力增大时心跳周期也会平滑地增大,因此借助轮询的心跳指令回带汇集消息的轻量级消息传递机制会受到影响,进而会增加任务移入下一个超级步的延迟.因此轻量级消息传递机制适用于非巨量待处理数据集且集群节点规模和运行压力适中的应用环境中.

图 9 测试的是当数据集规模增大时并行处理框架对于 PageRank 算法表现的可扩展性.在 24 节点集群中通过设定的 10 次迭代可以看到当数据规模的增加时并行框架显示了良好的线性扩展处理能力.同时可以发现重量级消息传递机制的算法总体执行时间增加幅度最小,也说明该机制具有很好的应对海量数据规模的可扩展能力.

最后图 10 表示的是新并行框架的兼容性测试结果,实验在原始的 Hadoop 集群中和支持 BSP 模型的改进型 Hadoop 集群中分别以相同的规模线性增加的数据集在 24 个节点上进行原始 MapReduce 作业的运行测试(改进型 Hadoop 运行时对于原始的 MapReduce 作业不会触发消息传递机能),实验选择最具代表型的 WordCount 应用作为原始的 MapReduce 作业.实验表明改进型 Hadoop 集群可以无损地兼容原始的 MapReduce 作业,不同类型的作业的调度由系统运行时自动处理.

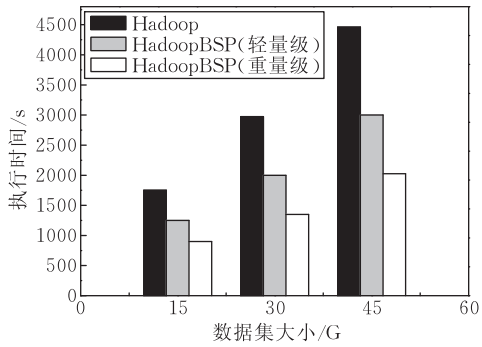


图 9 针对数据规模的总体执行性能和扩展性(24 nodes,10 次迭代处理)

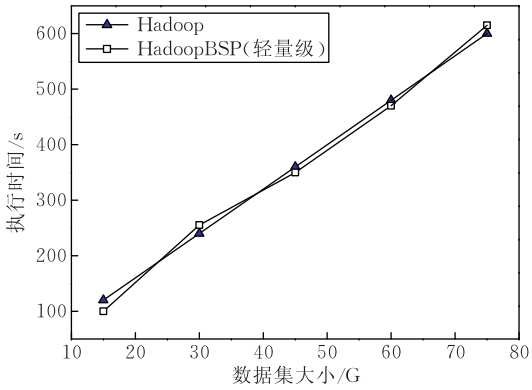


图 10 兼容性测试(WordCount,24 nodes,无迭代)

7 总结与展望

本文针对目前基于 MapReduce 的图算法执行性能低下的问题,在开源的 Hadoop 基础上通过引入大同步模型实现了一种支持障栅消息传递的改进型并行计算框架.通过将迭代过程内化到 Map 或 Reduce 阶段的超级步间,有效地减少了以往多轮作业调度的开销,为分布式大图算法的设计提供了一种高效的计算模式.实验证明相比于原始的 MapReduce 图算法,新计算框架下的分布式大图算法可行、高效.此外,如何解决信息交互频繁的稠密图性能退化的问题以及更广泛的机器学习、聚类等算法在该平台下的实现还有待于进一步研究.

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing

on large clusters//Proceedings of the Conference on Operating System Design and Implementation (OSDI'04). San Francisco, USA, 2004: 137-150

[2] Thusoo A, Sarma J S, JainN, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A warehousing solution over a map-reduce framework//Proceedings of the Conference on Very Large Databases (VLDB'09). Lyon, France, 2009: 1626-1629

[3] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: A not-so-foreign language for data processing//Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08). Vancouver, BC, Canada, 2008: 1099-1110

[4] Bu Y, Howe B, Balazinska M, Ernst M D. HaLoop: Efficient iterative data processing on large clusters//Proceedings of the Conference on Very Large Databases (VLDB'10). Singapore, 2010: 285-296

[5] Ekanayake J, Li H, Zhang B, Gunarathne T, Bae S-H, Qiu J, Fox G. Twister: A runtime for iterative MapReduce//Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. Chicago, Illinois, USA, 2010: 810-818

[6] Wilson G V. Practical Parallel Programming. Cambridge, MA: MIT Press, 1995

[7] Valiant L G. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103-111

[8] Dean J, Ghemawat S. MapReduce: A flexible data processing tool. Communications of the ACM, 2010, 53(1): 72-77

[9] Pavlo A, Paulson E, Rasin A, Abadi D J, DeWitt D J, Madden S, Stonebraker M. A comparison of approaches to large-scale data//Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09). New York, USA, 2009: 165-178

[10] Stonebraker M, Abadi D J, DeWitt D J, Madden S, Paulson E, Pavlo A, Rasin A. MapReduce and parallel DBMSs: Friends or foes? Communications of the ACM, 2010, 53(1): 64-71

[11] Cao Y, Chen C, Guo F, Jiang D, Lin Y, Ooi B C, Vo H T, Wu S, Xu Q. ES^2: A cloud data storage system for supporting both OLTP and OLAP//Proceedings of the IEEE International Conference on Data Engineering (ICDE'11). Hannover, Germany, 2011: 291-302

[12] Jiang D, Ooi B C, Shi L, Wu S. The performance of MapReduce: An in-depth study//Proceedings of the Conference on Very Large Databases (VLDB'10). Singapore, 2010: 472-483

[13] LinY, Agrawal D, Chen C, Ooi B C, Wu S. Llama: Leveraging columnar storage for scalable join processing in the MapReduce//Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11). Athens, Greece, 2011: 961-972

[14] Pike R, Dorward S, Griesemer R, Quinlan S. Interpreting the data: Parallel analysis with sawzall. Scientific Programming Journal, 2005, 13(4): 277-298

[15] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks//Proceedings of the 2nd European Conference on Computer Systems (EuroSys'07). New York, USA, 2007: 59-72

[16] Chaiken R, Jenkins B, Larson P, Ramsey B, Shakib D, Weaver S, Zhou J. SCOPE: Easy and efficient parallel processing of massive data sets//Proceedings of the Conference on Very Large Databases (VLDB'08). Auckland, New Zealand, 2008, 1(2): 1265-1276

[17] Borkar V R, Carey M J, Grover R, Onose N, Vernica R. Hyracks: A flexible and extensible foundation for data-intensive computing//Proceedings of the IEEE International Conference on Data Engineering (ICDE'11). Hannover, Germany, 2011: 1151-1162

[18] Zaharia M, Chowdhury M, Franklin M, Shenker S, Stoica I. Spark: Cluster computing with working sets//Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10). Boston, USA, 2010: 10-10

[19] Battre D, Ewen S, Hueske F, Kao O, Markl V, Warneke D. Nephele/PACTs: A programming model and execution framework for web-scale analytical//Proceedings of the ACM Symposium on Cloud Computing (SoCC'10). Indianapolis, Indiana, USA, 2010: 119-130

[20] Ekanayake J, Pallickara S, Fox G. MapReduce for data intensive scientific analysis//Proceedings of the 4th IEEE International Conference on eScience. Indianapolis, Indiana, USA, 2008: 277-284

[21] Chu C T, Kim S K, Lin Y A, Yu Y, Bradski G, Ng A, Olukotun K. Map-Reduce for machine learning on multicore//Proceedings of the Neural Information Processing Systems Conference (NIPS). Vancouver, Canada, 2006: 281-288

[22] Malewicz G, Austern M H, Bik A, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing//Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10). Indianapolis, Indiana, USA, 2010: 135-146

[23] Chen R, Weng X, He B, Yang M. Large graph processing in the cloud//Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10). Indianapolis, Indiana, USA, 2010: 1123-1126

[24] Lin J, Schatz M. Design patterns for efficient graph algorithms in MapReduce//Proceedings of the 8th Workshop on Mining and Learning with Graphs (MLG'10). New York, USA, 2010: 78-85

[25] Lin J, Dyer C. Data-Intensive Text Processing with MapReduce. United States: Morgan & Claypool Publishers, 2010

[26] Seo S, Yoon E J, Kim J, Jin S, Kim J-S, Maeng S. HAMA: An efficient matrix computation with the MapReduce framework//Proceedings of the Cloud Computing Technology and Science Conference (CloudCom'10). Indianapolis, USA , 2010: 721-726

[27] Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein J. GraphLab: A new framework for parallel machine learning//Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence(UAI'10). Catalina Island, California, USA, 2010

[28] Hoefler T, Lumsdaine A, Dongarra J. Towards efficient MapReduce using MPI//Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Berlin, Heidelberg, 2009:240-249

[29] Plimpton Steven J, Devine Karen D. MapReduce in MPI for large-scale graph algorithms. Parallel Computing, 2011, 37(9): 610-632

[30] Sutter H, Larus J. Software and the concurrency revolution. ACM Queue, 2005, 3(7): 54-62

[31] Albrecht Jeannie, Tuttle Christopher, Snoeren Alex C, Vahdat Amin. Loose synchronization for large-scale networked systems//Proceedings of the Annual Conference on USENIX'06 Annual Technical Conference (ATEC'06). CA, USA, 2006: 28-28

[32] Karloff H, Suri S, Vassilvitskii S. A model of computation for MapReduce//Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10). Philadelphia, PA, USA, 2010: 938-948

[33] Page L, Brin S, Motwani R, Winograd T. The pagerank citation ranking: Bringing order to the web. California:Stanford InfoLab, Technical Report: 1999-66, 1999

PAN Wei, born in 1977, Ph. D. candidate, lecturer. His research interests include data management for cloud computing, RFID data management and data mining.

LI Zhan-Huai, born in 1961, professor, Ph. D. supervisor. His research interests include database theory and technology.

WU Sai, born in 1980, Ph. D.. His research interests include P2P systems, data management for cloud computing.

CHEN Qun, born in 1976, professor, Ph. D. supervisor. His research interests include data management for cloud computing, RFID data management, and XML database technologies.

Background

MapReduce has become a popular paradigm for parallel massive-scale data processing on large cluster of commodity PCs. However, restricted by embarrassingly parallel assumption, parallel graph algorithms, in which iterative computation and inter-vertex communication are intrinsic necessities, are not easy to express in MapReduce. Currently, many researchers have done meaningful work on graph algorithms with MapReduce. However, existing methods have limited use because of their poor compatibility and commonality.

In this paper, we propose a modified Hadoop-based framework to address this issue by introducing the Bulk Synchronous Parallel model. New platform possesses the build-in support for message-passing and replaces chains of multiple MapReduce job operations with supersteps within the map and reduce phases. Experimental results show the superiority of enhanced version over original Hadoop on large-scale graph algorithms. In future, we plan to expand this modified version to support more algorithmic scenarios such as machine learning and data clustering where message-passing and iteration are required.

This research was supported by the National Natural Science Foundation of China under Grant Nos. 61033007, 60970070 and the National High Technology Research and Development Program (863 Program) of China under grant No. 2009AA01A404 and Project supported by the Major International (Regional) Joint Research Program of China under Grant No. 60720106001.

MapReduce is one of core technologies of cloud computing and how to implement large-scale graph algorithms on MapReduce has become quite popular in recent years. Our research group has been working on database research for many years and has close cooperation with a number of prestigious universities and scientific research institutes abroad, such as National University of Singapore and University of Queensland. We have published many papers in top-ranked international conferences and transactions, such as VLDB, ICDE, SIGMOD and TKDE.