计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20150619
53(11):2530-2541,2016

面向磁盘驻留的类 Pregel 系统的多级容错处理机制

毕亚辉[1] 姜苏洋[1] 王志刚[1] 冷芳玲[1] 鲍玉斌[1] 于 戈[1] 钱 岭[2]
[1](东北大学计算机科学与工程学院 沈阳 110819)
[2](中国移动(苏州)软件技术有限公司 江苏苏州 215163)
(biyahui1990@163.com)

A Multi-Level Fault Tolerance Mechanism for Disk-Resident Pregel-Like Systems

Bi Yahui[1], Jiang Suyang[1], Wang Zhigang[1], Leng Fangling[1], Bao Yubin[1], Yu Ge[1], and Qian Ling[2]
[1](*College of Computer Science and Engineering, Northeastern University, Shenyang* 110819)
[2](*China Mobile (Suzhou) Software Technology Co, Ltd, Suzhou, Jiangsu* 215163)

Abstract The BSP-based distributed frameworks, such as Pregel, are becoming a powerful tool for handling large-scale graphs, especially for applications with iterative computing frequently. Distributed systems can guarantee a flexible processing capacity by adding computing nodes, however, they also increase the probability of failures. Therefore, an efficient fault-tolerance mechanism is essential. Existing work mainly focuses on the checkpoint policy, including backup and recovery. The former usually backups all graph data, which leads to the cost of writing redundant data since some data are static during iterations. The latter always loads backup data from remote machines to recovery iterations, ignoring the usage of data in the local disk in special scenarios, which incurs network costs. It proposes a multi-level fault tolerant mechanism, which distinguishes failures into computing task failures and node failures, and then designs different strategies for backup and recovery. For the latter, considering that the volume of data involved in computation varies with iterations, a complete backup policy and an adaptive log-based policy are presented to reduce the cost of writing redundant data. After that, at the stages of recovery, we utilize the local graph data and the remote message data to handle the recovery for task failures, but the remote data are used for node failures. Finally, extensive experiments on real datasets validate the efficiency of our solutions.

Key words fault tolerance; large-scale graph; iterative computing; BSP model; checkpoint

摘 要 基于 BSP 模型的分布式框架已经成为大规模图高频迭代处理的有效工具.分布式系统可以通过增加集群节点数量的方式提供弹性的处理能力,但同时也增加了故障发生的概率,因此亟需开发高效的容错处理机制.现有工作主要是基于检查点机制展开研究,包括数据备份和故障恢复 2 部分:前者没有考虑迭代过程中参与计算的数据规模的动态变化,而是备份所有图数据,因此引入了冗余数据的写开销;后者通常是从远程存储节点上读取备份数据进行故障恢复,而没有考虑利用本地磁盘数据恢复某些

收稿日期:2015-06-30;**修回日期**:2015-10-29
基金项目:国家自然科学基金重点项目(61433008);国家自然科学基金项目(61173028,61272179);中央高校基本科研业务费专项基金项目(N100704001);教育部-中国移动科研基金项目(MCM20125021)
This work was supported by the Key Program of the National Natural Science Foundation of China (61433008), the National Natural Science Foundation of China (61173028, 61272179), the Fundamental Research Funds for the Central Universities (N100704001), and Chinese Ministry of Education-China Mobile Communications Corporation Research Funds (MCM20125021).

场景下的故障,引入额外的网络开销.因此提出了一种多级容错处理机制,将故障分为计算任务故障和计算节点故障 2 类,并设计了不同的备份和恢复策略. 备份阶段利用了某些应用在迭代计算过程中参与计算的数据规模的动态变化特性,设计了完全备份和写变化 log 自适应选择的策略,可以显著减少冗余数据的写开销.故障恢复阶段,对任务故障,利用本地磁盘上保留的图数据和远程的消息数据完成恢复;而对节点故障,则利用备份在远程信息进行恢复.最后,通过在真实数据集上的大量实验,验证了提出的多级容错机制的有效性.

关键词 容错;大规模图;迭代计算;BSP 模型;检查点

中图法分类号 TP311.13

随着图数据规模的快速增长和分析复杂性的不断增加,大量支持大规模图迭代计算的分布式处理系统被开发[1-3],其中,Giraph[1]和 BC-BSP[2]在迭代计算过程中提供了基于磁盘辅助的数据和中间消息存储.分布式系统可以通过增加计算节点数量的方式提高处理的能力和效率.然而,系统在迭代过程中发生故障的概率与节点规模成正比[4].对于长时间迭代计算的图处理应用,需要设计高效的容错处理机制.

目前分布式图处理系统采取的处理故障方法一般是基于检查点的方法[2-3].检查点机制包括数据备份与数据恢复 2 部分.各个任务周期性地将图数据和有关信息备份到分布式文件系统(如 HDFS)中.当系统发生故障时,各任务从分布式文件系统中读取检查点备份的数据和有关信息来完成故障恢复.基于检查点的方法原理简单明了,容易实现.然而,现有的基于检查点的方法存在 2 方面的不足:

1) 在数据备份时,并没有区分迭代过程中数据是否发生动态变化,而是将所有的图数据信息进行备份,因此导致写检查点时产生了大量冗余数据的写操作;

2) 在故障恢复阶段,通常是从远程存储节点上读取备份的信息进行故障恢复,而没有考虑利用本地磁盘数据恢复某些场景下的故障,尤其是在面向磁盘驻留的计算系统中,例如任务故障的情形,使得在某些类型的故障恢复过程中需要远程读取检查点数据,存在"远程读"问题,引入了网络开销.

针对上述 2 个问题,本文提出了一种面向磁盘驻留的类 Pregel 系统的多级容错处理机制.所谓的多级是针对任务故障和节点故障的数据备份与恢复策略而言的.首先,对于数据备份策略,根据数据备份的位置和规模,可以分为 3 个级别:第 1 级别,被处理的图数据有本地备份和消息数据在 HDFS 上备份;第 2 级别,静态数据(顶点的出度邻接表,并假设在迭代计算过程中不改变)、动态数据(迭代过程中动态变化,如 PageRank 的 PR 值)和消息都备份在 HDFS 上;第 3 级别,HDFS 上备份有图的动态数据加日志(log)信息、静态数据和消息.对应于数据备份的 3 个级别,故障的恢复也有 3 个级别:第 1 级别,读取本地的数据和 HDFS 上的消息;第 2 级别,读取 HDFS 上的静态数据、动态数据和消息;第 3 级别,读取 HDFS 上的静态数据、启用 log 机制(记录变化的动态数据)后的动态数据和消息.对于任务故障的备份与恢复采用的是第 1 级别的容错处理机制,对于节点故障备份与恢复采用的是第 2 级别和第 3 级别的容错处理机制.本文所提出的多级容错处理机制能够有效地处理分布式图处理系统出现的任务故障以及节点故障,该机制适用于采用磁盘辅助的类 Pregel 系统.

本文的主要贡献如下:

1) 任务故障的恢复直接读取本地磁盘的静态数据与动态数据和 HDFS 上的消息,避免了加载 HDFS 上的静态和动态数据的开销,加快了任务故障的处理过程.

2) 提出了 log 机制,当参与计算的图数据规模小于指定阈值时,使用 log 方式记录数据变化而不是全部备份动态数据,以减少备份数据的写开销.此外,本文还给出了 log 启动阈值设置的理论分析.

3) 在大量的实验基础上,对比了传统的检查点(checkpoint)机制和本文的多级容错处理机制,验证了本文的多级容错处理机制的有效性.

1 相关工作

设计高效的容错方案始终是分布式图处理系统重点解决的问题.因此,已有许多关于分布式(图)处理系统的容错机制的研究工作.已有的方法可以分为基于检查点的方法、基于日志的方法和混合方法 3 类.

目前大多数知名的分布式图处理系统如Giraph[1], GraphLab[5], PowerGraph[6], GPS[7], Mizan[8]系统采用的都是基于传统检查点的方法;GraphX[9]采用基于日志的方法. Pregel[3]系统中提供了2种容错机制:1)基本的写检查点机制;2)受限的恢复机制,即采用的是一种基于检查点和日志相结合的混合方法.

Pregel的基于基本写检查点机制实现的容错机制是周期性地备份顶点的状态和消息以实现容错.当一个或多个节点发生故障,主节点重新分配这些图的分区到当前可用的工作节点集合上,这些节点会从最近记录检查点的超步 S 开始重新加载分区状态.

Pregel提出的另一种受限的容错恢复机制是一种基于检查点和基于日志相结合的方法.除了基本的检查点,工作节点同时将图数据加载和迭代计算期间从这个节点上分区发出去的消息记录到日志中,这样故障恢复就会被限制在丢失的分区上.这种方法的优点是:只重新计算丢失的分区,节省了恢复时的计算资源,同时由于每个工作节点需要恢复的分区很少,减少了恢复的延迟;缺点是对发送出去的消息进行保存会产生一定的存储开销,降低了作业正常运行时的效率.本文的恢复机制虽然还是要重新计算所有分区,但通过日志记录发生变化的动态数据可以减少检查点的存储开销及网络IO开销.Pregel的受限恢复机制可以与本文的工作互补.

Spark系统[10]将图数据信息分为动态数据和静态数据.写检查点只记录动态变化的部分.对于绝大部分真实图,静态数据的规模远大于动态数据,因此这种方式极大减少了写检查点的开销.本文的多级容错机制借鉴了Spark的这种处理方式,即第2级别.进一步地,对于某些算法,如单源最短路径(SSSP),迭代过程中仅有部分顶点参与计算,即参与更新计算的动态数据的规模是变化的.针对这种情形,本文提出了第3级容错方案——写日志机制(即log机制)来进一步减少IO开销.此外,Spark对于任务故障和节点故障的恢复都是加载存储在分布式文件系统的检查点数据,没有利用本地磁盘数据.

GraphX[4]采用的是基于日志(血统)的恢复方法,它利用弹性分布式数据集(RDD)加速故障恢复.然而,当一个节点发生故障时,这个节点上的图数据仍然需要恢复.

文献[11]则针对传统检查点性能低下的问题提出了基于内存缓存的异步检查点容错方法.其主要思想是将检查点临时缓存在节点的内存中,然后由另一个辅助任务将缓存在内存中的检查点数据写到分布式文件系统.但是这种异步的检查点容错方法并不适用于类Pregel系统,因为类Pregel系统需要在写检查点时进行全局同步才能进入下一个超步.

2 多级容错处理机制概述

本节首先介绍BC-BSP系统及其现有的检查点机制,然后介绍本文的备份与恢复框架.

2.1 BC-BSP系统简介

BC-BSP系统[2]是基于BSP模型的开源大图迭代处理系统,支持多种数据输入方式和使用磁盘辅助暂存数据(简称磁盘操作),具有良好的容错控制能力和可伸缩性.图1给出了BC-BSP的系统结构图.它包括客户端(Client)、BSP Controller端、Worker端、Staff端和完成同步协调的ZooKeeper.

客户端是用户与BC-BSP系统交互的实体,作业的提交和运行状态的监控均需要通过客户端平台实现.Controller端是BC-BSP系统的中枢控制系统,负责调控整个集群,包括作业调度、故障恢复等.Worker端是工作节点的控制中心,隶属于Controller端,负责本节点的整体运行调控.Staff端是工作实体,完成具体的工作任务,从逻辑上讲,按照用户提交的作业进行组织,但是在集群中受Worker端的直接管理.全局同步、消息通信和容错控制,是作业运行过程中的重要环节,需要Controller端、Worker端和Staff端的协同工作来实现.其中的ZooKeeper作为第三方插件,在BC-BSP系统的任务调度模块、高可用(HA)管理模块、全局同步模块以及聚集计算功能的实现中具有重要作用.

鉴于数据量的不断激增和硬件资源的相对缺乏,BC-BSP系统支持使用磁盘作为迭代计算过程中的辅助存储介质,暂存图数据和中间消息数据等,而不是假设所有数据(包括中间的消息数据)都在内存.因此,系统中实现了磁盘缓存模块,它负责暂存系统计算时内存无法容纳的图数据和消息数据.其基本思路是:对于图数据,在迭代计算过程中常驻磁盘,在图处理系统的数据加载阶段,每个计算任务从原始数据所在的存储系统(通常为HDFS或HBase)按照数据分片记录的位置信息加载数据,数据加载程序每读取一个顶点的数据,就按照该顶点的ID值,根据系统设定的映射规则,将其写入到对应节点的磁盘块中.图数据在本地磁盘的存储是按照Hash

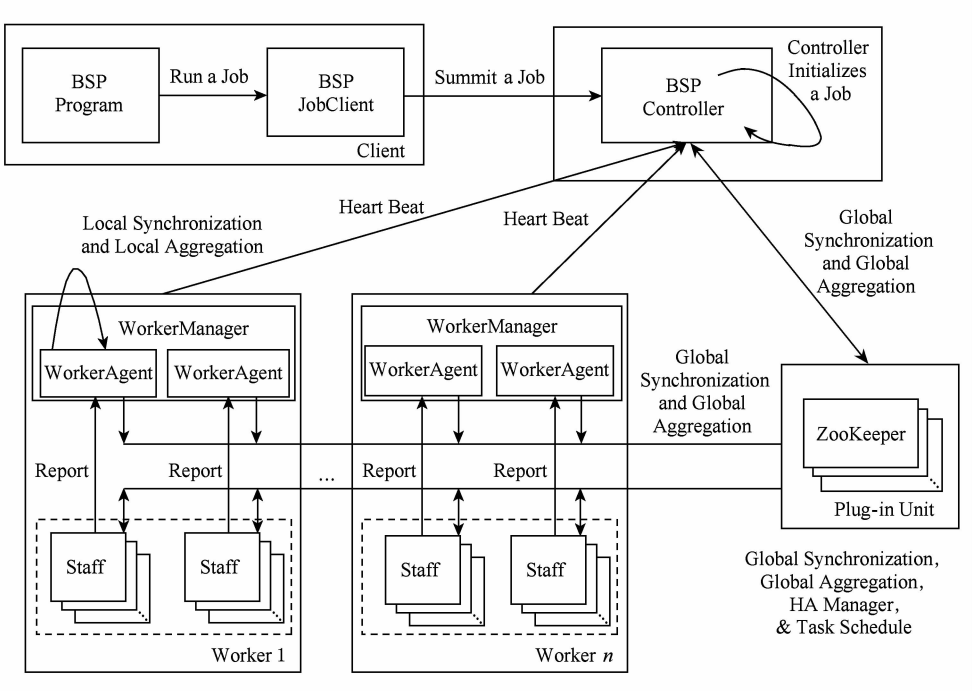


Fig. 1 The system structure of BC-BSP.

图 1 BC-BSP 系统结构关系图

分桶组织,且每个任务的数据被分成图顶点(动态数据)、边(静态数据)和消息 3 个部分,每一部分都分为若干个 Hash 桶存放到本地磁盘上,桶的数量可由用户自行设定.进入迭代计算阶段,每个超步结束后,将动态变化的顶点数据写回本地磁盘,而不发生变化的静态数据只在需要处理时才从本地磁盘读入内存,处理结束后并不需要写回磁盘,因为它没有变化.而对消息数据,则尽可能地存储在内存中,如消息发送时内存缓冲区中的数据量超出用户设置的缓冲区上限,计算等待发送;在消息接收时,如果所占用缓冲区的大小也超出用户设置的接收消息缓冲区上限,则接收过程要同步等待数据块写入磁盘.

2.2 BC-BSP 现有的容错机制

BC-BSP 当前版本的数据备份就是对作业本地计算的中间结果按照一定的频率(比如每隔 k 个超步)记录检查点.分布式文件系统中记录的检查点由 3 部分信息组成:1)原始的图数据信息,该部分数据在作业完成之前一直存在;2)每次以增量方式(即只记录顶点动态数据而不记录顶点的出边信息)记录的检查点信息;3)各个分区收到的、在下个超步处理的消息.为了节省存储资源,当新的检查点记录成功之后则删除历史检查点.数据恢复即从分布式文件系统加载最后记录的检查点信息,加载时要同时读取原始图数据信息和最近的增量检查点信息,以及备份的消息这样才能还原到最近检查点记录时图处理作业继续运行的上下文状态.BC-BSP 系统写检查点的流程如图 2 所示.我们称这种容错机制为"增量检查点"机制.

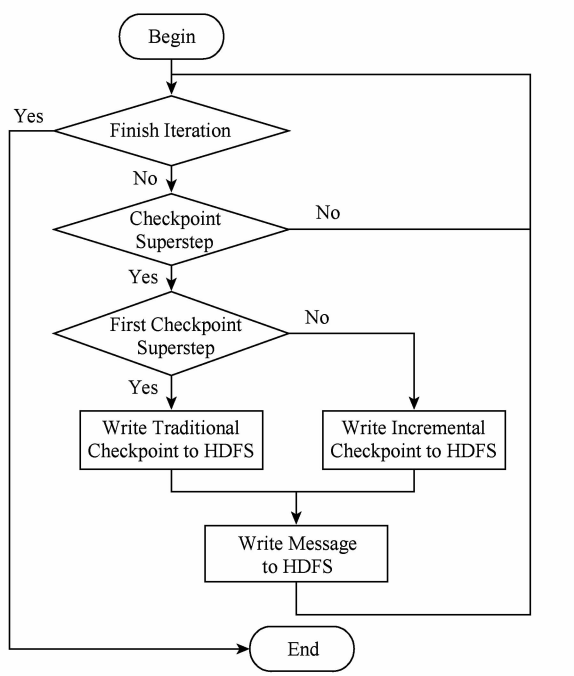


Fig. 2 The flowchart of write checkpoint.

图 2 写检查点流程图

BC-BSP 系统对故障的检测是通过心跳机制完成的.当主节点在一定的时间内没有收到工作节点的心跳信息,就把该节点标记为故障节点.

2.3 多级容错机制的备份与恢复框架

系统运行过程中各个任务加载分区数据到该任务本地的磁盘上,动态数据每个迭代步都写回本地磁盘,静态数据在每次迭代计算中是只读的.进入迭代计算阶段,如果没有发生故障,无论动态数据或是静态数据的访问都是针对本地磁盘的.迭代过程中,系统按照配置文件中设置的检查点频率周期性地记录检查点.在增量检查点机制中,除了第1次写检查点时需要记录完整的图数据(包括顶点 Id、$value$ 值和出边)之外,其后的每个检查点只需记录图的动态数据(顶点 Id 与 $value$ 值)即可.若计算过程中存在节点间交互,则这种交互的信息都以消息的形式备份到 HDFS 上.故障恢复时会读取检查点及备份的消息进行恢复.但是,通过对某些应用的运行特征进行观察,我们发现图的动态部分也不是全部变化的,例如在单源最短路径计算中每次参与计算的点很少.因此,当动态数据变化的规模小于一定阈值时,启用 log 机制来记录变化的动态数据,这样需要备份的数据量就小于完整的动态数据部分.这里的关键是阈值的确定问题,3.1 节将详细讨论.多级容错机制备份算法如算法 1 所示.

算法 1. *computeFramework*().

输入:log 机制启用标志 *logFlag*.

① While *flag*=true /* *flag*:本地循环计算标志 */
② For each vertex *v*
③ *compute*();
④ If *logFlag*=true
⑤ 将*v* 放到 *c* 中;/* *c*:值发生变化的顶集合 */
⑥ End If
⑦ End For
⑧ 将动态数据写回本地磁盘文件;
⑨ If *commandType*.*equals*("CHECKPOINT") /* *commandType*:超步命令类型 */
⑩ 将消息写到 HDFS;
⑪ If *isFirstCheckpoint*()=true
⑫ 将整个子图写到 HDFS;
⑬ Else If *logFlag*=true
⑭ 将 *c* 写到 HDFS;
⑮ Else
⑯ 将增量检查点写到 HDFS;
⑰ End If
⑱ End If
⑲ End While

当故障发生时,针对不同的故障类型采取不同的恢复策略.故障恢复过程的框架见算法 2 所示.

算法 2. *FaultRecovery*(*faultType*).

输入:故障类型 *faultType*.

① If *faultType*.*equals*("任务故障")
② 加载本地静态和动态数据及 HDFS 上的消息;
③ Else If *logFlag*=true /* *logFlag*:log 启用的标志 */
④ 从 HDFS 加载检查点数据、日志和消息;
⑤ Else
⑥ 从 HDFS 加载检查点数据和消息;
⑦ End If

对于任务故障,系统直接在本地重启故障的任务,各个任务(包括重启的恢复任务)直接利用本地保存的图数据以及远程的消息数据恢复到最近的检查点,因为图数据在本地有完整的信息,且任务故障不会造成本地的数据不可用(除了极少数文件损坏的情况外).这就避免了加载 HDFS 上的检查点图数据,从一定程度上加快了任务恢复的过程.而对于节点故障,系统首先利用故障恢复调度机制对在这个节点上的所有任务进行迁移操作,因为节点发生故障就不能再使用存储在本地的图数据进行恢复了.此时,如果发生故障的任务没有启用 log 机制,那么迁移后的任务通过读取 HDFS 上的静态数据、动态数据和消息进行恢复;如故障任务启用了 log 机制,通过读取 HDFS 上的静态数据、log 机制记录的动态数据和消息进行恢复.

3 多级容错机制的数据备份策略

写检查点是常用的容错数据备份机制:按照一定的频率或超步间隔将各个任务处理的数据和顶点所收到的消息写入分布式存储介质(如 HDFS).因为假设内存不足,系统所处理的数据常驻磁盘,需要时才加载到内存,所以各任务处理的数据每个超步结束后都保存到本地磁盘,静态部分常驻磁盘,动态变化部分每个超步都写回本地磁盘.

这样,利用本地的静态数据、动态数据和 HDFS 上备份的消息就可以完成任务故障的恢复.

3.1 log 机制及其启用条件

在现有的增量备份策略中,是将动态部分数据全备份到远程,但实际上有些应用每次迭代计算,甚至在写检查点间隔期间,并不会更新分区上所有的

状态或者值,因此为了减少写入检查点的冗余数据,当顶点值发生变化的比例低于一定的阈值时,就开启 log 机制.所谓的 log 机制就是在迭代过程中只有一小部分顶点的值发生改变时,记录这些发生改变的顶点的信息.log 机制的实现可以有 2 种方式:作业级的实现和任务级的实现.作业级的实现就是当作业满足开启 log 机制的条件时,对这个作业的所有任务都启用 log 机制;任务级实现是针对某个任务而言的,如某个任务满足启用 log 机制的条件,对这个任务本身启用 log 机制.log 机制的作业级实现的优点是:当启用 log 机制时能够加快整个作业的运行速度,降低存储开销;而任务级实现的优点是:启用 log 机制的任务能够加快该任务本身的运行,减少存储开销,更加灵活,当所有任务都开启 log 机制时也能加快作业的运行.本文的 log 机制是在任务级实现的,以任务为单位开启 log 机制.采用任务级的 log 机制,虽然作业的整体运行时间要受到没有开启 log 机制的任务运行时间的影响,但是对于启用 log 机制的任务大大减少了检查点写入 HDFS 的数据量,减少了存储开销.而当所有的任务都开启 log 机制时,作业的整体运行时间也会得到很大的改善.

如果开启了 log 机制,那么在 2 个检查点之间,每个超步都要将变化的日志写到远程,或者暂存在本地.这样的话,如果这些超步累计记录的信息大于全部动态数据(本部分增量写只写这么多),那么记日志就没有优势可言了.对于顶点值发生变化的比例阈值,本文选取为检查点频率(记为 c)的倒数,即 $1/c$,此时满足式(1):

$$\sum_{i=1}^{c} P(S_i) < 1, \tag{1}$$

其中,$P(S_i)$为第 i 超步某任务顶点值发生变化的比例,S_i 为第 i 个超步.此时开启 log 机制能够保证在 2 个检查点之间所记录的顶点不会大于原来检查点所记录的顶点规模.另外对每个任务设置一个 log 机制的标志位,用于判断该任务的 log 机制是否开启.对于 log 机制开启条件的判断,本文采用了一种预测式的判断,如图 3 所示.

对提交的作业从 S_1 开始(S_0 为任务的初始化超步,不进行记录)对 2 个检查点之间的每一个超步内顶点值变化的顶点比例进行收集,设 S_k 为第 1 个检查点的超步数,一个任务在 $S_1, S_2, \cdots, S_k$ 满足式(2):

$$\max\{P(S_1), P(S_2), \cdots, P(S_k)\} < 1/c, \tag{2}$$

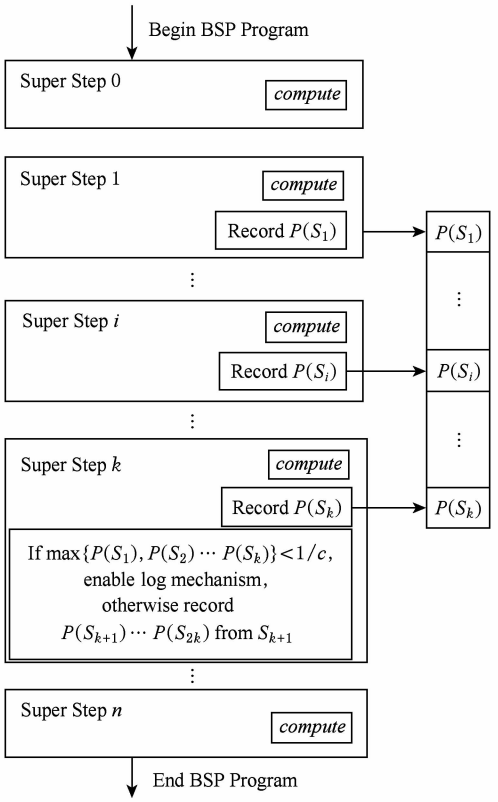


Fig. 3 The decision of enabling log mechanism.

图 3 log 机制启用判定

其中,$P(S_i)$为第 i 步某任务顶点值变化的比例,那么该任务将会在 S_{k+1} 步开启 log 机制,该任务的 log 机制标志位置为 true,否则从 S_{k+1} 开始重新收集变化的顶点比例.开启 log 机制后,从 S_{k+1} 继续开始记录每个超步内变化顶点最新的 $value$ 值,到 S_{2k} 时将这些顶点的变化记录到 HDFS 的一个文件中,这就是 log 机制记录的 log 文件.系统在 $S_{2k}, S_{3k}, S_{4k}\cdots$ 不再记录完整的图顶点信息,而是记录这些 log 信息,log 的存储规模远小于所有顶点值的存储规模.

3.2 log 文件生成及优化

log 机制开启后,如果顶点在参与计算之后其值发生改变,那么该顶点的信息将会被暂时记录在内存中.每记录一个顶点之前首先要查找内存中是否存在该顶点,如不存在直接记录,否则记录顶点的最新的值.在写检查点时,内存中的所有记录将会以 log 文件记录到 HDFS.

开启 log 机制后,每到一个检查点就会记录一个 log 文件.因此,当一个作业运行的超步数比较多时,就会在 HDFS 上产生很多 log 文件,这些 log 文件会影响节点故障的恢复.为了避免在发生节点故障时合并大量的 log 文件,任务每产生 n 个 log 文件(n 可由配置文件读入)就会启动一个线程在后台

合并 log 文件,从而减少发生节点故障时要合并的 log 文件的数量.后台的线程独立于作业的执行,因此不会影响作业运行时间.

当大部分的顶点都发生变化时,启用 log 机制的开销过大,此时检查点记录的数据量不会有明显减少,反而会因 log 文件的合并增大节点故障的恢复开销,这时就不适合启用 log 机制.采用这种策略,在启用 log 机制之后只备份发生变化的动态数据,明显减少了检查点备份的数据.而在发生节点故障后也能根据 log 信息、动态数据、静态数据及消息进行恢复.

开启 log 机制对作业运行的收益为

$$Benefit = (1-p) \times Benefit_{\mathrm{N}} + p \times Benefit_{\mathrm{R}}, \tag{3}$$

其中,p 为系统发生节点故障的概率,则 $1-p$ 为系统正常运行至结束或发生任务故障恢复的概率;$Benefit_{\mathrm{N}}$ 为启用 log 机制相比于没有启用 log 机制的作业正常运行至结束或发生任务故障恢复的收益;$Benefit_{\mathrm{R}}$ 为启用 log 机制相比于没有启用 log 机制的作业进行节点故障恢复的收益.$Benefit_{\mathrm{N}}$ 和 $Benefit_{\mathrm{R}}$ 可分别由式(4)和式(5)表示:

$$Benefit_{\mathrm{N}} = \sum_{i=1}^{(s-s_{\log})/c+1} (CostW_{ck} - CostW(i)_{\log}), \tag{4}$$

$$Benefit_{\mathrm{R}} = Benefit_{\mathrm{N}} - \sum_{i=1}^{(s_{\mathrm{f}}-s_{\log})/c} CostR(i)_{\log}. \tag{5}$$

式(4)中,$s_{\log}$ 为第 1 次记录 log 文件的超步数,$(s-s_{\log})/c+1$ 为总共记录的 log 文件的个数,$CostW_{ck}$ 为记录一次检查点的代价,$CostW(i)_{\log}$ 为第 i 次记录 log 的代价.

式(5)中,s_{f} 表示发生节点故障的超步数,则 $(s_{\mathrm{f}}-s_{\log})/c$ 为故障任务需要读取的 log 文件的数量;$CostR(i)_{\log}$ 为读取第 i 个 log 文件的代价.

为简化问题,我们忽略在内存中记录变化的顶点信息的开销及后台进程对 log 文件的合并.由式(4)和式(5)可以看出,开启 log 机制获得的收益和检查点频率、发生节点故障的超步数有密切的关系.检查点频率设置得越小,发生节点故障的超步数越小,开启 log 机制相对于 BC-BSP 的增量检查点获得的收益可能越大.

4 多级容错机制的故障恢复策略

4.1 任务故障的恢复

任务运行过程中,会因为运行环境的影响,例如出现异常、文件读写错误等,导致任务不能正常运行.这种任务故障一般不会造成本地数据的损坏(极少数的任务故障由文件的磁盘故障引起,造成文件损坏,本文忽略此种情况),所以系统对于任务故障的恢复策略是直接在原来的节点上重新启动故障任务,所有任务加载检查点进行故障恢复.

根据本文提出的多级容错处理机制的第 1 级容错处理机制的数据备份策略,本地磁盘保存了任务故障恢复所需的静态数据和动态数据.因此,故障任务可以直接加载本地保存的静态数据和检查点时刻的动态数据以及 HDFS 上备份的消息,回滚到距离故障超步最近的检查点进行故障恢复.本文的第 1 级容错处理机制避免了加载 HDFS 保存的静态数据和动态数据,直接利用本地磁盘保存的静态数据和动态数据进行恢复,加快了任务故障恢复时加载检查点所需的时间,同时也减轻了网络传输的压力.

4.2 节点故障的恢复

节点故障一般是由分布式系统中的物理机宕机或网络原因导致.这种故障一般会造成故障节点不可用.系统对于节点故障的处理流程是:首先利用故障恢复调度机制对在这个节点上所有的任务进行迁移操作,因为节点发生故障就不能再使用该节点进行本地恢复,故障任务将会被迁移到正常的节点上重启;然后加载检查点进行恢复.迁移到其他节点的任务由于缺少该任务以前的本地的动态数据与静态数据,因此系统通过读取 HDFS 记录的静态数据与动态数据及消息进行节点故障的恢复.

如果故障任务没有开启 log 机制,可以利用第 2 级容错处理机制(即 BC-BSP 的增量检查点机制)恢复策略,加载检查点上的静态数据、动态数据及消息进行故障恢复.如果故障任务开启 log 机制,根据第 3 级容错处理机制的恢复策略,需要加载检查点记录的静态数据、动态数据、log 文件及消息进行节点故障恢复.第 2 级容错处理机制已在 2.2 节详细介绍,这里不再赘述.

第 3 级容错处理机制的恢复策略具体为:按照 log 文件生成的先后顺序,首先读取最晚生成的 log 文件的每一条记录到内存中;然后依次读取较早生成的 log 文件,较早记录的 log 文件中的顶点 ID 如在内存中已记录就无需再记录,而较早记录的 log 文件中的顶点在内存中不存在时便记录此顶点信息.扫描完所有的 log 文件之后,再读取记录有全图信息的检查点记录(可能存在增量检查点,也可能只存在第 1 个检查点记录),将内存中的记录与检查点

按照上述方法再次合并,最终生成故障发生前最近的检查点时刻的完整动态数据.该策略完整描述如算法 3 所示.

算法 3. *readLogCheckPoint*(*s*,*ck*).

输入:记录第 1 个 log 文件的超步数 *s*、检查点频率 *ck*;

输出:图数据对象 *graphData*.

① /* 读取未合并 log 文件集合 */
② For each log file *lfr* in *Nr* /* *Nr*:未合并 log 文件集合 */
③ For each 顶点 *v* in *lfr*
④ 将*v* 放入 *c* /* *c*:值发生变化的顶点集合 */
⑤ End For
⑥ End For
⑦ /* 读取已合并 log 文件集 */
⑧ For each log file *lfm* in *Nm* /* *Nm*:已合并 log 文件集合 */
⑨ For each 顶点 *v* in *lfm*
⑩ If 不存在 *v*
⑪ 记录 *v*;
⑫ End If
⑬ End For
⑭ End For
⑮ /* 合并 log 与记录有全图信息的检查点 */
⑯ If *s*=2×*ck* /* 未记录增量检查点 */
⑰ For each 顶点 *v* 在正常的检查点中
⑱ If 不存在 *v*
⑲ 记录 *v*;
⑳ End If
㉑ *graphData*.*add*(*v*); /* 将顶点添加到 *graphData* */
㉒ End For
㉓ Else
㉔ For each 顶点 *v* 在增量检查点中
㉕ If 不存在 *v*
㉖ 记录 *v*;
㉗ End If
㉘ *graphData*.*add*(*v*); /* 将顶点添加到 *graphData* */
㉙ End For
㉚ End If
㉛ Return *graphData*.

5 实验结果与分析

5.1 数据集与实验设置

本文使用 2 个真实图数据集进行实验,包括 Wiki[12] 和 USA-Road[13],具体描述如表 1 所示.测试使用的应用包括计算图的连通分量(CC)和单源最短路径(SSSP).

Table 1 Description of Real-World Graphs

表 1 真实数据集描述

Dataset	Vertex Num	Edge Num	Avg Out-degree	Dataset Size/GB
USA-Road	23 947 132	58 332 307	2.436	1.01
Wiki	5 716 808	135 877 199	23.768	1.56

本文在 BC-BSP 系统上实现了多级容错处理机制.本文实验的对比分析包括:第 1 级容错处理机制与 BC-BSP 的增量检查点机制的对比,即任务故障恢复时加载 HDFS 与加载本地数据的对比;不同写检查点频率下开启 log 机制与关闭 log 机制的作业运行时间、写检查点 I/O 开销(不包括消息)的对比;第 3 级容错处理机制与 BC-BSP 增量检查点的对比,不同写检查点频率下节点故障的恢复;顶点值变化比例的阈值对作业运行的影响.第 2 级容错处理机制即为原 BC-BSP 系统节点故障的恢复机制.实验所用集群由 15 个节点构成,且由 1 台 Gigabit 以太网交换机连接,每个计算节点配置酷睿 i3-2100 双核处理器、8 GB 内存、1TB 的 7200RPM 硬盘.每个节点最大任务槽数设为 2,测试时启动 10 个任务,多余的节点是为了发生节点故障时有可用的节点进行故障迁移.节点的心跳间隔设为 1 s,心跳超时时间设为 3 s.测试的参数为检查点频率和故障超步数,测试的指标为作业运行时间和写检查点 I/O 开销.

5.2 任务故障恢复

我们在 2 个真实数据集上使用 SSSP 和 CC 测试了 BC-BSP 增量检查点机制加载 HDFS 与多级容错处理机制的第 1 级容错处理机制加载本地磁盘进行任务故障恢复的时间.这里我们设置检查点频率为 6,运行至第 8 步发生任务故障,共运行 10 个超步.

图 4 为不同的应用分别在 BC-BSP 的增量检查点机制与本文的多级容错机制的第 1 级容错机制下进行任务故障恢复的运行时间,图 5 统计了平均每

个任务在进行任务故障恢复时加载检查点所花费的时间.

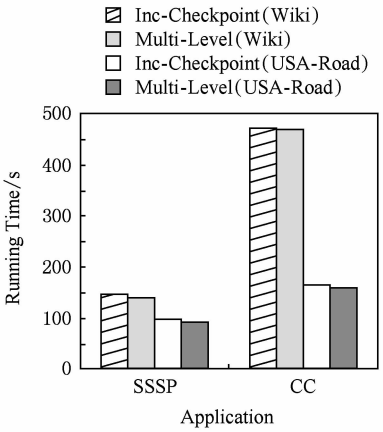


Fig. 4 The recovery time of task failure.

图 4 任务故障恢复时间

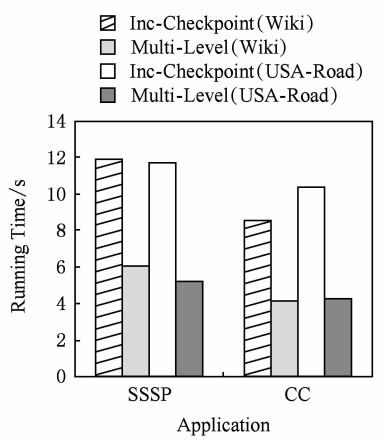


Fig. 5 The checkpoint time of task failure recovery.

图 5 任务故障恢复加载检查点时间

综合图 4 和图 5 的实验结果可以发现,对于不同应用(SSSP 和 CC),加载本地数据的时间比加载 HDFS 的时间快了 1 倍多,作业恢复运行的总时间也有所改善.此外,由于加载本地数据不需要网络传输,因此也降低了网络传输的开销.本节的实验证明了本文多级容错处理机制的第 1 级容错处理机制的高效性.

BC-BSP 进入迭代计算阶段,每个超步结束后将动态变化的顶点数据写回本地磁盘,不发生变化的静态数据只在需要处理时再从本地磁盘读入内存,而在处理结束时并不写回磁盘,因为它没有变化.任务故障的恢复只需额外备份检查点写到磁盘上的动态数据,所以存储代价为一步的动态数据的大小.经实验测得,对于 2 种应用在 USA-Road 数据集上,每台机器的存储代价均为 21 MB,而 Wiki 数据集的存储代价均为每台机器 5MB.因此,存储代价很低.

5.3 log 机制对正常运行的作业的影响

在 2 个真实数据集上,使用 SSSP 测试了 log 机制,以证明开启 log 机制能够加速作业正常运行.

图 6 给出了 SSSP 在数据集 Wiki 和 USA-Road 上以不同的检查点频率正常运行 40 个超步时,BC-BSP 的增量检查点机制与 log 机制的运行时间的对比.

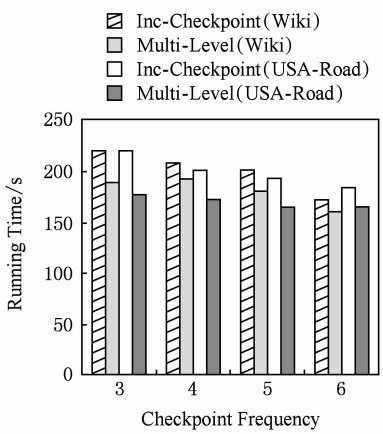


Fig. 6 Running time of job against the checkpoint frequency.

图 6 不同检查点频率作业运行时间

由图 6 可以看出,开启 log 机制后的作业运行时间明显减少,而写检查点频率设置越小,log 机制对于作业正常运行时间的收益越大.因为,开启 log 机制的任务在写检查点时备份的数据量比 BC-BSP 的增量检查点机制要少得多,在记录多个检查点后,log 机制明显地缩短了作业正常运行的时间.

图 7 给出了 SSSP 在数据集 Wiki 和 USA-Road

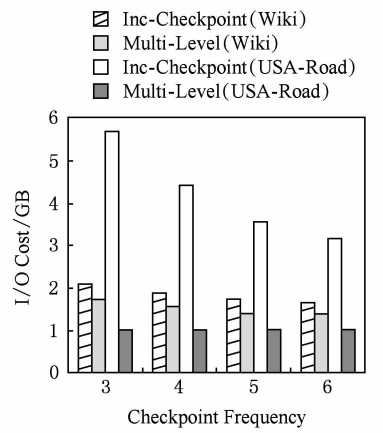


Fig. 7 I/O cost of backup against the checkpoint frequency.

图 7 不同检查点频率下备份的 I/O 开销

上以不同的检查点频率正常运行 40 个超步时,BC-BSP 的增量检查点机制与 log 机制的备份 I/O 开销对比.由图 7 可以看出,启用 log 机制后备份的 I/O 开销比 BC-BSP 的增量检查点机制的要小,特别是对 USA-Road 这种平均出度比较小的数据集效果更加显著.这是因为 SSSP 在 USA-Road 启用 log 机制的超步比在 Wiki 上早很多,因此 SSSP 在 USA-Road 上的 I/O 收益要远高于在 Wiki 上的 I/O 收益,在时间上的收益也高于 Wiki.本节从时间和备份的 I/O 开销的角度来对比 BC-BSP 的增量检查点机制与 log 机制,实验结果论证了本文的 log 机制提高了作业正常运行的效率.

5.4 log 机制对节点故障恢复的影响

我们在 2 个真实数据集使用 SSSP 测试了 log 机制对于节点故障恢复的影响,为了说明故障超步数和检查点频率对节点故障恢复的影响,我们在 Wiki 数据集上进行了测试.

图 8 和图 9 分别给出了以不同的检查点频率在第 17 步与第 33 步制造节点故障时运行 40 个超步 log 机制对于节点故障恢复时间的影响.

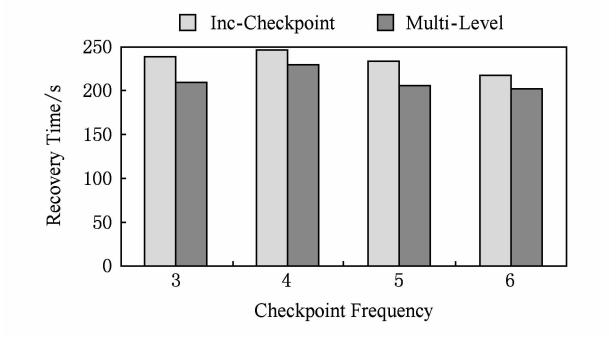


Fig. 8 Recovery time of job against the checkpoint frequency.

图 8 不同检查点频率作业恢复时间

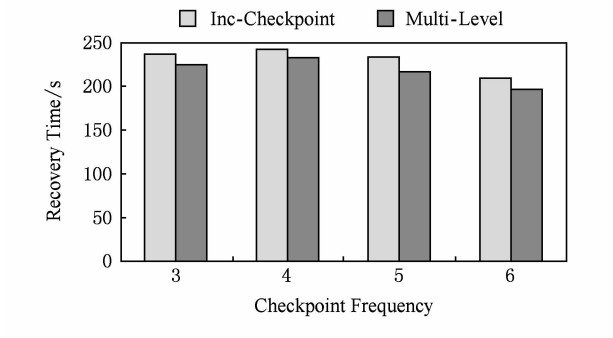


Fig. 9 Recovery time of job against the checkpoint frequency.

图 9 不同检查点频率作业恢复时间

对比图 8 和图 9 可以发现,故障步数越大,恢复时需要读取的 log 文件内容可能越多,合并 log 所花费的时间开销也有所增大,但是恢复的总时间仍小于没有开启 log 机制的总时间.这是因为,在启用 log 机制的情况下,写检查点的时间开销减少了,节省的时间足以抵消读取 log 文件所花费的时间.因此开启 log 机制在一定程度上加速了节点故障的恢复过程.本节实验说明第 3 级容错处理机制加速了节点故障的恢复.

5.5 log 启动阈值对 log 机制的影响

我们使用 SSSP 和 CC 在 Wiki 上测试不同的 log 启动阈值对于 log 机制的影响,以验证 3.1 节理论分析.本节实验中,阈值的定义为值发生变化的顶点占所有顶点的比例,而检查点频率设置为 5,运行 40 个超步.

图 10 和图 11 分别给出了在不同阈值下作业的运行时间与备份的 I/O 开销.该阈值的选取要对作业运行时间和写检查点的 I/O 开销优化相对多.因为阈值选取过大可能会造成在内存中记录的 log 信息过多,从而导致内存开销过大;图 11 可以看出,阈值选取为 10%时,启用 log 机制比较晚,导致备份的 I/O 开销比较大.通过权衡作业的运行时间和写检

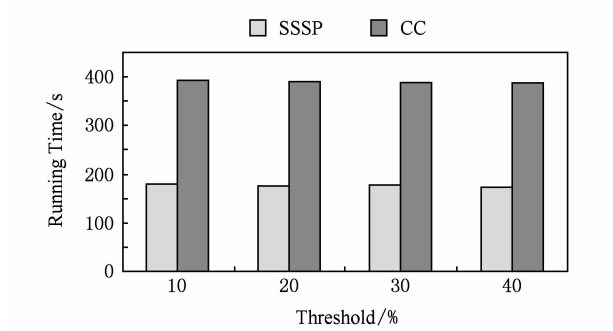


Fig. 10 Running time of job against the threshold of starting the log mechanism.

图 10 不同 log 机制启动阈值下作业运行时间

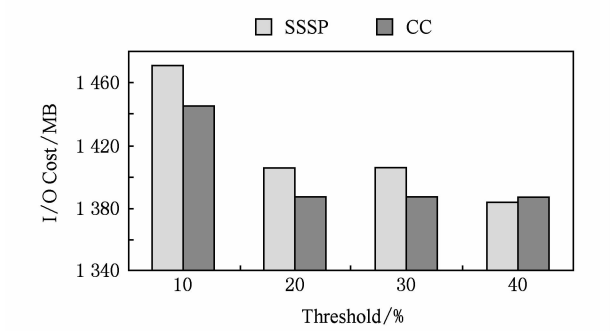


Fig. 11 I/O cost of backup against the threshold of starting the log mechanism.

图 11 不同 log 机制启动阈值下备份的 I/O 开销

率的倒数)是比较合适的. 当阈值为 20%时,写检查点的 I/O 相对较小,作业的运行时间也和其他 3 种阈值下的作业运行时间相当. 这说明了 3.1 节中对这个阈值的推导是正确的.

6 结 论

本文提出了多级容错处理机制,通过在 2 个真实数据集上大量的对比实验证明了多级容错机制的高效性与正确性. 第 1 级容错处理机制直接利用本地保存的动态数据、静态数据及 HDFS 上的消息进行恢复,避免了加载 HDFS 上动态数据、静态数据从而加快了其恢复过程. 第 3 级容错处理机制对于顶点值变化比例较低的应用,例如 SSSP 和 CC,通过 log 记录变化的顶点信息而极大减少了传统检查点机制所记录的数据量和存储开销(实验中也发现,对于每个超步顶点变化比例较高的应用,例如 PageRank,意义不大). 系统在所有任务都启用 log 机制后,整个作业的运行时间明显减少. 通过 log 信息进行节点故障的恢复,在节点恢复过程中虽然引入了合并 log 的过程,但由于作业运行过程中开启了 log 机制,作业的整体运行时间在一定程度上仍有所降低.

下一步的工作将探索日志合并频率对 log 机制的影响,即它的改变对于节点故障恢复时间的影响. 通过大量实验找出一个最合适节点故障恢复的日志合并频率.

参 考 文 献

[1] The Apache Software Foundation. Introduction to Giraph [EB/OL]. [2015-05-25]. http://giraph.apache.org/intro.html

[2] Bao Yubin, Wang Zhigang, Yu Gu, et al. BC-BSP: A BSP-based parallel iterative processing system for big data on cloud architecture [C] //Proc of the 1st Int DASFAA Workshop on Big Data Management and Analytics. Berlin: Springer, 2013: 31-45

[3] Malewicz G, Austern M H, Bik A J C, et al. Pregel: A system for large-scale graph processing [C] //Proc of the 2010 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2010: 135-146

[4] Shen Y, Chen G, Jagadish H V, et al. Fast failure recovery in distributed graph processing systems [J]. Proceedings of the VLDB Endowment, 2014, 8(4): 437-448

[5] Low Y, Gonzalez J E, Kyrola A, et al. GraphLab: A new framework for parallel machine learning [J/OL]. 2014[2015-05-25]. http://arxiv.org/abs/1408.2041

[6] Gonzalez J E, Low Y, Gu H, et al. PowerGraph: Distributed graph-parallel computation on natural graphs [C] //Proc of the 10th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 17-30

[7] Salihoglu S, Widom J. GPS: A graph processing system [C] //Proc of the 25th Int Conf on Scientific and Statistical Database Management. New York: ACM, 2013: 22

[8] Khayyat Z, Awara K, Alonazi A, et al. Mizan: A system for dynamic load balancing in large-scale graph processing [C] //Proc of the 8th ACM European Conf on Computer Systems. New York: ACM, 2013: 169-182

[9] Xin R S, Gonzalez J E, Franklin M J, et al. GraphX: A resilient distributed graph system on spark [C] //Proc of the 1st Int Workshop on Graph Data Management Experiences and Systems. New York: ACM, 2013: 1-6

[10] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C] //Proc of the 9th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 141-146

[11] Yi Huizhan, Wang Feng, Zuo Ke, et al. Asynchronous checkpoint/restart based on memory buffer [J]. Journal of Computer Research and Development, 2015, 52(6): 1229-1239 (in Chinese)
(易会战, 王锋, 左克, 等. 基于内存缓存的异步检查点容错技术[J]. 计算机研究与发展, 2015, 52(6): 1229-1239)

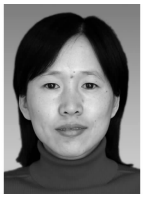
[12] Wikipedia. Using the Wikipedia Link [EB/OL]. [2015-05-25]. http://haselgrove.id.au/wikipedia.htm

[13] Sapienza University of Rome. Using the USA-Road Link [EB/OL]. [2015-05-25]. http://www.dis.uniroma1.it/challenge9/download.shtml

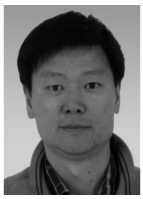
Bi Yahui, born in 1990. Master candidate at the College of Computer Science and Engineering, Northeastern University. His main research interests include cloud computing and graph management, etc.

Jiang Suyang, born in 1991. Master candidate at the College of Computer Science and Engineering, Northeastern University. Her main research interests include cloud computing and graph management, etc.

Wang Zhigang, born in 1987. PhD candidate at the College of Computer Science and Engineering, Northeastern University. His main research interests include cloud computing and graph data mining, etc.



Leng Fangling, born in 1978. Received her PhD degree in computer software and theory from Northeastern University in 2008. Lecturer at Northeastern University. Member of China Computer Federation. Her main research interests include data warehouse and online analytical processing (OLAP), etc.



Bao Yubin, born in 1968. Received his PhD degree in computer software and theory from Northeastern University in 2003. Professor at Northeastern University. Senior member of China Computer Federation. His main research interests include data warehouse, online analytical processing (OLAP), cloud computing and data intensive computing, etc.

Yu Ge, born in 1962. Received his PhD degree in computer science from Kyushu University of Japan in 1996. Professor and PhD supervisor at Northeastern University. His main research interests include database theory and technology, distributed system, parallel computing and cloud computing, etc.

Qian Ling, born in 1972. Received his PhD degree of engineering at the Department of Computer Science and Technology, Tsinghua University, in 2001. He joined Bell Labs Research China in 2001. He worked on IPv6 edge router, voice messaging, voip, instant messaging, LBS, mobile application and other related projects. In 2008, he joined China Mobile Research Institute and worked on mobiles ads, big data and cloud computing projects.