软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(7):2078−2102 [doi: 10.13328/j.cnki.jos.006269]

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

面向分布式图计算作业的容错技术研究综述*

张程博[1], 李 影[1,2], 贾 统[3]

[1](北京大学 软件与微电子学院,北京 102600)

[2](北京大学 软件工程国家工程研究中心,北京 100871)

[3](北京大学 信息科学技术学院,北京 100871)

通讯作者: 李影, E-mail: li.ying@pku.edu.cn

摘 要: 随着图数据规模的日益庞大和图计算作业的日益复杂,图计算的分布化成为必然趋势.然而图计算作业在运行过程中面临着分布式图计算系统内外各种来源的非确定性所带来的严峻的可靠性问题.首先分析了分布式图计算框架中不确定性因素和不同类型图计算作业的鲁棒性,并提出了基于成本、效率和质量 3 个维度的面向分布式图计算作业的容错技术评估框架,然后分别对分布式图计算的 4 种容错机制——基于检查点的容错、基于日志的容错、基于复制的容错、基于算法补偿的容错等机制结合国内外相关工作做了深入的分析、评估和比较.最后对未来的研究方向进行了展望.

关键词: 图数据;故障和失效;分布式图计算;容错机制;非确定性软件系统

中图法分类号: TP311

中文引用格式: 张程博,李影,贾统.面向分布式图计算作业的容错技术研究综述.软件学报,2021,32(7):2078−2102. http://www.jos.org.cn/1000-9825/6269.htm

英文引用格式: Zhang CB, Li Y, Jia T. Survey of state-of-the-art fault tolerance for distributed graph processing jobs. Ruan Jian Xue Bao/Journal of Software, 2021,32(7):2078−2102 (in Chinese). http://www.jos.org.cn/1000-9825/6269.htm

Survey of State-of-the-art Fault Tolerance for Distributed Graph Processing Jobs

ZHANG Cheng-Bo[1], LI Ying[1,2], JIA Tong[3]

[1](School of Software and Microelectronics, Peking University, Beijing 102600, China)

[2](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China)

[3](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract: As the growth of graph data scale and complexity of graph processing, the trend of distributed graph processing shall be inevitable. However, graph processing jobs run with severe reliability problems caused by the uncertainty originated from inside and outside the distributed graph processing system. This study first analyzes the uncertainty factors of the distributed graph processing frameworks and the robustness of different types of graph processing jobs; then proposes an evaluation framework of fault tolerance for distributed graph processing based on cost, efficiency, and quality of fault tolerance. This study also analyzes, evaluates, and compares the four fault-tolerant mechanisms of distributed graph processing—checkpointing based fault tolerance, logging based fault tolerance, replication based fault tolerance, and algorithm compensation based fault tolerance—combining related researches. Finally, the direction of future researches is prospected.

Key words: graph data; fault and failure; distributed graph processing; fault tolerance; uncertainty software system

* 基金项目: 广东省重点领域研发计划(2020B010164003)

Foundation item: Key-area Research and Development Program of Guangdong Province, China (2020B010164003)

本文由“面向非确定性的软件质量保障方法与技术”专题特约编辑陈俊洁副教授、汤恩义副教授、何啸副教授以及马晓星教授推荐.

收稿时间: 2020-09-15; 修改时间: 2020-10-26; 采用时间: 2020-12-14; jos 在线出版时间: 2021-01-22

作为一种建模对象间关系的常用数据结构,图(graph)通常由顶点 V(即对象)、边 E(即对象间关系)和标签 L(即顶点或边的属性)这 3 个要素构成,其数据种类丰富、分布广泛.一方面,天然的图数据广泛存在于社会生活的方方面面,如社交网络、交易网络、物联网、交通运输网和供电网络等;另一方面,图数据也从各种任务的建模中产生,如商品推荐中的矩阵分解(消费者、商品及之间的购买关系)和文本分类中的 LDA(latent Dirichlet allocation)主题模型(文档、单词及之间的包含关系)等.对这些图数据的处理分析,即图计算(graph processing)任务,广泛应用于欺诈和威胁检测、社交网络分析、个性化推荐和自然语言处理等诸多领域,创造了重要的社会和经济价值.

然而,一方面,图计算要处理的图数据规模十分庞大,大图普遍存在并被广泛使用.一项针对企业从业者和科研人员的调研报告[1,2]指出,有 22.5%的任务需使用和处理超过十亿边的大图,有 4.5%的任务甚至使用超过一万亿边的超大图;另一方面,图计算任务种类丰富且计算复杂,统计数据[1,2]表明,除了常用的 SSSP(单源最短路径)和 PageRank 等图算法以外,有超过 68.5%的任务在图上应用了各种机器学习算法.因此,单机往往无法满足图计算作业的存储和计算需求,图计算的分布化成为目前的主流趋势[3].

图计算作业在运行过程中面临着分布式图计算系统内外各种来源的非确定性所带来的严峻的可靠性问题:在系统内,(1) 分布式图计算系统的复杂性给图计算作业的运行带来更多的不确定性.为了帮助用户专注于作业逻辑,现有的分布式图计算框架(distributed graph processing framework)——专用分布式图计算框架[4−8]和基于通用分布式数据流框架(general-purpose distributed dataflow framework)的高层图计算库[9,10],均具有较高的抽象层次,如图 1 所示,系统的复杂性高,故障可能在硬件、操作系统、数据流框架等底层发生;(2) 图计算作业数据驱动的计算过程导致集群节点间负载不均衡,增加了作业运行的不确定性.数据驱动的计算过程导致难以在作业提交前通过对数据和计算任务的均衡切分来均衡各节点的计算负载和节点间的通信负载,并且图计算作业运行过程中负载动态变化,进一步加剧了负载均衡的实现难度,而集群中负载的不均衡是导致故障发生的重要原因之一;在系统外,(3) 分布式图计算作业运行集群规模庞大,集群可靠性低.即使集群中单节点平均无失效时间(mean time between failure,简称 MTBF)可达 100 万小时,随着集群规模的增加,集群整体 MTBF 则呈指数级降低[11].极端情况下,集群中每时每刻都会有节点失效;(4) 云计算环境的异构、多租户等特性也威胁着作业运行的可靠性.异构性加剧了图计算作业负载难以均衡的问题,多租户特性使得图计算作业的性能受到环境中其他作业的干扰[12−14],甚至由其他作业产生的故障会导致图计算作业的失效.而分布式图计算作业任务间的依赖较为复杂,故障传播快、影响大:在分布式图计算作业运行期间,每个任务都需要从其他顶点(通常是邻居顶点)接收或读取数据来更新其自身的顶点值,这导致任务间的依赖关系复杂化,一旦有任务发生故障,错误会迅速在任务间传播,进而导致整个作业运行失败,或者当集群中一个节点失效,幸存节点因为无法读取失效节点的数据,使得幸存节点上的任务无法继续进行,导致整个作业运行中断,严重影响作业性能.上述原因导致分布式图计算作业面临着故障发生频率高、传播快、影响大等技术挑战,因此,如何对分布式图计算作业和系统进行有效的容错,使其在实际应用场景下仍能保持稳定的性能是亟待解决的核心问题.

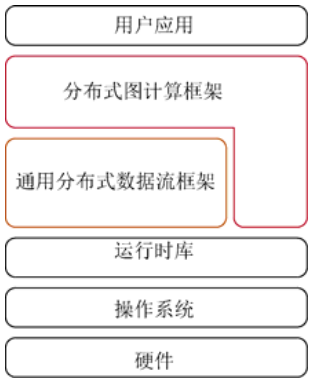

Fig.1 Abstract level of distributed graph processing framework

图 1 分布式图计算框架所处的抽象层

随着分布式图计算作业处理的图数据日益庞大、计算日益复杂,其对容错机制的性能开销、恢复效率等提出了更高的要求.传统的基于硬件[15,16]、操作系统[17−20]、运行时库[21−24]的容错机制只能对作业进行粗粒度的状态管理,如进程复制、计算节点备份等,性能开销极大,无法满足分布式图计算作业对性能的需求.基于此,面向分布式图计算作业的容错技术应运而生.面向分布式图计算作业的容错技术主要从两方面进行优化,一方面通过与分布式图计算框架、分布式图计算作业进行紧密耦合,实现容错质量更好、恢复效率更高且容错成本更低

的容错机制;另一方面通过深入分析理解不同的分布式图计算作业的容错需求,实现容错机制在容错的“不可能三角”——成本、效率和质量之间更为灵活、准确地权衡和取舍.例如,针对仅需近似准确结果的作业而言,在失效恢复时将其恢复至全局一致状态是不必要的,可以通过快速地近似恢复实现高效率.

本文第 1 节介绍分布式图计算框架、分布式图计算作业以及分布式图计算中的故障和失效模型等背景知识.第 2 节分析面向分布式图计算作业的容错技术的优化目标和面临的挑战,进而提出对容错机制进行评估的框架和指标.第 3 节从分布式计算框架发展的角度梳理分布式图计算容错技术的发展阶段.第 4 节总结归纳 4 种分布式图计算容错机制,分别基于检查点、日志、复制和算法补偿,并结合相关工作进行详细的分析、比较和评估.最后,在第 5 节总结现有工作的不足并对未来的研究方向加以展望.

1 背景概述

分布式图计算框架的异构性和复杂性一方面给图计算作业的运行带来了大量的不确定性,另一方面也为容错机制的设计提供了优化空间,同时图计算作业本身的鲁棒性和作业可能面临的失效类型对容错机制的设计也有重要影响.本节首先对分布式图计算框架作了总结介绍,然后从失效及恢复的角度对分布式图计算作业进行了分类,最后对分布式图计算中的故障和失效类型以及容错框架进行了分析和总结.

1.1 分布式图计算框架

自 2010 年第一个分布式图计算框架 Pregel[4]提出以来,分布式图计算框架在近 10 年来被不断地研究、发展和提出.据不完全统计[25],截止 2017 年,先后有不下 40 种分布式图计算框架被提出.大体上,分布式图计算框架由 4 个相互依赖的组件构成,分别是编程抽象(programming abstraction)、图分区(partitioning)、任务调度(scheduling)和通信(communication).这 4 个组件共同驱动了图计算作业的分布式执行,并共同决定了作业执行的性能[26],如图 2 所示.

Fig.2 Distributed graph processing framework components and their types

图 2 分布式图计算框架组件及其类型

首先,编程抽象(programming abstraction)往往是用户感知最为敏感的组件,它向用户提供了一个输入图数据的局部视图(顶点视图应用最为广泛)和一组在此视图上进行操作的方法接口(即编程模型).例如,Pregel[4]向用户提供了一个顶点视图和在此顶点上进行操作的 compute()方法接口,用户在 compute()内实现接收消息、更新顶点值和发送消息的计算逻辑,而图上的每个顶点都将按照该逻辑进行计算.从容错的角度来看,顶点视图为容错提供了更细粒度的容错单元.相较于节点备份、进程复制、基于数据块的 task 重试(MapReduce)[27]、基于 RDD 分区的重播(Spark)[28]等粗粒度的容错方法,基于顶点的细粒度容错方法天然地有更小的性能开销.此外,编程抽象中除了有采用 compute()方法接口的单段 Vertex-Centric 编程模型以外,应用较为广泛的还有两段的

scatter-gather(SG)模型(也称为 signal/collect 模型[29])以及 3 段的 gather-apply-scatter(GAS)模型[7]等.在 SG 模型中,用户需要在 Scatter 接口方法中实现顶点产生要发往其他顶点的消息的计算逻辑,在 Gather 接口方法中实现利用接收到的消息更新自身顶点值的计算逻辑;在 GAS 模型中,用户需要在 Gather 接口方法中实现在顶点的边和邻居顶点的子集上产生局部值的计算逻辑,在 Apply 接口方法中实现累加 Gather 阶段并行产生的多个局部值并更新自身顶点值的计算逻辑,最后在 Scatter 接口方法中实现依据更新后的顶点值更新边值的计算逻辑.虽然不同的编程模型有不同的适用场景,但总的来说,友好的编程模型大大简化了用户对分布式图计算作业的实现.亦受益于此,相较于在需要用户自己处理计算并行、数据切分、节点通信等问题的专用程序上实现用户应用程序级的容错机制[30,31],基于采用以上编程模型的用户应用程序加以实现更为简单和便利.

其次,在分布式图计算作业提交后需要加载数据进行计算,而图分区(partitioning)则是数据加载的第 1 步,选择将图划分成多少个分区,并决定哪些边或顶点属于一个分区.而如何将图切开(cut),则是图分区面临的第 1 个问题,目前图分区主要基于 3 种图切分(cut)方法,分别是边切分(edge-cut)、顶点切分(vertex-cut)和混合切分(hybrid-cut).其中,边切分是指在分区过程中把边“剪断”,从而可以将顶点分配在不同的机器上,如图 3(b)所示,图中的顶点 A、B 和 C 与顶点 D、E 和 F 分别分配在了机器 $M1$ 和 $M2$ 上.然而边切分对高度(high-degree)顶点是不合适的.在计算过程中,高度顶点需要和大量分布在其他机器上的邻居节点通信,这会造成网络负载的不均衡,从而影响计算性能,而顶点切分则能改善这一问题.如图 3(d)所示,在分区过程中将顶点“切开”,将边分配在不同的机器上,并从被切成多份的顶点中选择一份作为 master,其余作为 mirror.搭配 GAS 编程模型,在计算过程中,mirror 在各自机器上完成局部计算,将结果汇总至 master 进行更新后,再将更新结果同步至 mirror 上.但是,采用顶点切分对低度(low-degree)顶点是不合适的,master 与 mirror 之间的汇总-同步过程反而带来了额外的开销,因此低度顶点更适合边切分,于是混合切分方法应运而生.通过设置合理的阈值将图中顶点分为高度和低度两类,混合切分方法对高度顶点采用顶点切分,对低度顶点采用边切分,综合了边切分和顶点切分的优势,从而提高了作业的整体性能.从容错的角度看,图切分的重要意义在于其在系统中为大量顶点创建了副本,实现了状态冗余,虽然图切分方法和图分区策略通常会极力减少冗余,但这并不容易,充分利用已有的状态冗余是面向分布式图计算作业的容错技术的一个重要的优化方向.

接着,在数据加载进内存后,用户编写的计算逻辑结合输入数据实例化成一组互相依赖的任务.作业的执行过程就是对这些任务进行迭代、反复调度执行的过程.任务调度(scheduling)组件采用不同的调度机制将任务按照一定的顺序调度到 CPU 上进行计算,目前主流的任务调度机制有同步调度机制和异步调度机制.同步调度机制一般基于整体同步并行(bulk synchronous parallel,简称 BSP)计算模型实现,作业计算的整个过程按照迭代轮次被全局同步栅划分成多个超步,每个超步开始的时候,所有任务均处于同一个迭代轮次.同步调度机制设计简单、可扩展性强,但在并行计算中全局同步栅引入了相当的性能开销.而在异步调度中,每个任务所能使用的数据可以是其他任务迭代产生的最新结果,这样计算快的任务无需等待落后任务即可继续进行计算,从而提升了作业整体的执行效率.然而,异步调度机制为了保证数据的一致性,避免读写冲突,需要引入额外的控制机制,如版本控制、分布式锁机制等,这增加了分布式图计算框架设计的复杂性,也为计算引入了额外开销.从容错的角度来看,调度机制是面向分布式图计算作业容错技术优化时的重要考量,一方面,调度机制会影响容错机制对作业状态的管理,例如借助同步调度机制的全局同步栅可以方便地获得作业的全局一致状态;另一方面,调度机制也一定程度地反映了作业对结果准确度的需求程度.例如,一般基于异步调度机制的作业仅需要近似准确的结果,容错机制可以借此实现更高效的近似恢复.

最后,通信(communication)组件负责任务间的数据交换,目前,主流的通信机制包括消息传递(massage-passing)、共享内存(shared-memory)和数据流(dataflow)这 3 种.其中,消息传递机制主要利用网络通信协议在任务间交换消息完成数据的交换,如图 3(b)所示,其天然地契合边切分;共享内存机制则是通过将每个任务所需要的数据以共享变量的方式保存在本地机器上,从而实现直接的本地读写以避免远程读写,其结合异步调度机制可以实现更高的计算效率,如图 3(d)所示,其天然地契合顶点切分;当结合边切分使用时,如图 3(c)所示,需要在本地机器上为远程邻居顶点创建 ghost 顶点实现本地读写;数据流机制主要是应用在基于通用分布式数据流框架的高层图计

算库.在这类框架内,虽然高层图计算库,如 GraphX 和 Gelly 可以提供不同的编程模型、支持不同的图切分方式以及调度机制,但是底层的数据交换仍然是基于数据流的——图计算的迭代过程被抽象成 join-groupBy-aggregation 的数据流模版(pattern),而数据交换则发生在 shuffle 过程中.相较消息传递和共享内存机制,基于数据流机制的分布式图计算框架在灵活性和可扩展性上较差,但具有更好的通用性,帮助用户在同一个框架内完成多种数据处理任务,如预处理、模型构建、图计算等.而从容错的角度看,通信机制对容错机制的设计和实现有着重要的影响,例如基于日志的容错技术就更适合应用于采用消息传递机制的作业,一方面,在实现逻辑上更为简单和直接,保存传递的消息即可;另一方面,消息传递机制的众多优化功能,如为了提高网络吞吐率的 Combiner,同样也能提高基于日志容错技术的恢复效率.

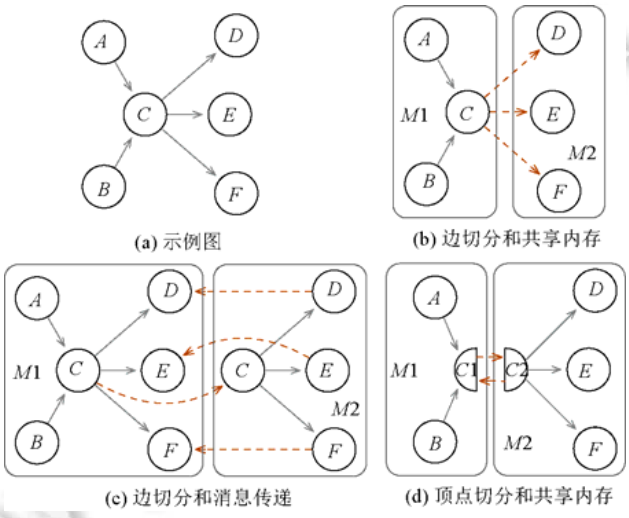

Fig.3 Examples of graph-cut methods and communication modes

图 3 图切分方法和通信方式示意图

以上介绍了分布式图计算框架的四大组件及其各自的技术类型,并简要介绍了各种组件特点能够为容错机制优化提供的可能性,详细内容将在第 4 节结合具体内容进一步加以论述.

1.2 分布式图计算作业

分布式图计算作业由输入、计算过程和输出构成,其输入通常是一个由顶点、边及其值构成的有向图,其计算过程可被描述成一个基于用户定义的计算逻辑(user defined function,简称 UDF)在图上多次迭代直至达到终止条件(如收敛)的过程,而其输出则视作业情况而定,可以是更新后的图、顶点及其值的集合或者统计数字等,不一而足.以 PageRank 作业为例,输入是一个顶点带有 rank 值的有向图;用户定义的计算逻辑是每个顶点接收其入边方向邻居顶点传来的 rank 值后,更新自身的 rank 值,然后再沿出边方向,向邻居顶点发送更新后的 rank 值;而作业的计算过程就是在每个迭代中使图中的每个顶点都执行上述逻辑,然后多次迭代,直到图中所有顶点的 rank 值更新幅度均小于某一阈值后,迭代收敛,计算过程结束;输出是图中的所有顶点及其更新后的 rank 值.

上述过程可以建模成一个由图中顶点值和边值构成的全局状态的演变过程.在同步调度机制中,作业基于当前全局状态在一个超步内对全部的顶点或边完成值的更新,随后进入到下一个全局状态;而在异步调度机制中,作业基于当前全局状态往往仅能依据具体的调度策略对一部分的顶点或边完成更新,即进入到下一个全局状态.在作业运行过程中,失效的发生将导致全局状态部分损坏或丢失,作业面临状态不一致的问题.容错中的失效恢复就是对全局状态进行“修复”的过程.因为分布式图计算作业不同的运行特点和对运行结果不同的准确度需求,容错机制可以在失效恢复中对全局状态进行不同程度的“修复”,如图 4 所示.基于此,分布式图计算作业可分为 4 类,分别是自稳定作业、可矫正作业、全局一致作业和近似作业.

为了方便后续对各类作业进行详细的解释,这里对图 4 中的相关状态进行了定义.首先,图中顶点和边的值

在某一刻所构成的全局状态可以表示为状态向量 s,向量的元素即顶点或边的值.其中,作业开始运行时由顶点和边的初始值所构成的全局状态记作初始状态(initial state)向量 s_i;当失效发生后,顶点和边的值所构成的全局状态记作故障状态(faulty state)向量 s_e;当作业运行结束得到完全准确的结果时,顶点和边的值所构成的全局状态记作最终状态(final state)向量 s_f;当运行结束得到近似准确的结果时,顶点和边的值所构成的全局状态则记作近似最终状态(approximate final state)向量 s_a.其次,图中顶点和边的所有可取值的笛卡尔积所构成的状态空间,记作全部状态(all states)集合 S;作业从初始状态 s_i 开始正常执行所能达到的所有状态的集合,记作全局一致状态(globally consistent states)集合 S_{GC};介于这两者之间的包括强有效状态(strong valid states)集合 S_{SV} 和弱有效状态(weak valid states)集合 S_{WV},当作业从某一强有效状态开始正常执行,最终能够产生完全准确的结果[32],而当作业从某一弱有效状态开始正常执行时,最终仅能产生近似准确的结果.最后,全部状态、弱有效状态、强有效状态和全局一致状态依次是包含关系,即 $S \supseteq S_{WV} \supseteq S_{SV} \supseteq S_{GC}$.

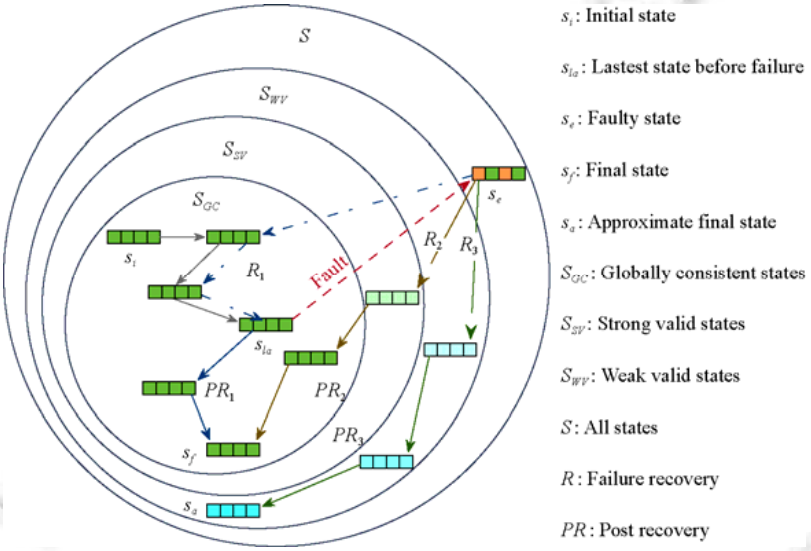

Fig.4 Changes of global states in distributed graph processing jobs during failure-free execution and after failure

图 4 分布式图计算作业正常运行时和失效发生后全局状态的变化

当失效发生后,自稳定作业从任意状态开始重新计算均可到达最终状态 s_f 得到完全准确的结果,典型的有信念传播作业、图着色作业等.该类作业一般采用随机初始化图中顶点值或边值的方式开始计算,并且计算的结果与顶点值和边值无关,因此对这类作业来说,任何状态都是强有效状态,即 $S_{GC} \subset S_{SV}=S_{WV}=S$.可矫正作业要求容错机制将全局状态从失效后的故障状态 s_e“修复”至某一强有效状态(图 4 中 R_2 过程)后继续计算才能最终到达最终状态 s_f(图 4 中 PR_2 过程),并得到完全准确的结果,典型的有广度优先遍历作业、单源最短路径作业和隐式狄利克雷分布作业等.该类作业重新开始计算的强有效状态虽然不属于全局一致状态,即 $S_{GC} \subset S_{SV} \subset S_{WV} \subseteq S$,但作业自身的计算逻辑能够保证最终达到最终状态 s_f.此外,在分布式集群环境中,可矫正作业又被分为局部可矫正作业和全局可矫正作业,前者在“修复”至强有效状态的过程中仅修改失效节点上的状态,而后者则涉及修改幸存节点上的状态.全局一致作业则要求容错机制必须将全局状态从失效后的故障状态 s_e“修复”至某一全局一致状态(图 4 中 R_1 过程)后继续计算才能最终到达最终状态 s_f(图 4 中 PR_1 过程),进而得到完全准确的结果,例如,中间中心度(betweenness centrality)[33]作业.该类作业的任一强有效状态都是全局一致状态,即 $S_{GC}=S_{SV} \subset S_{WV} \subseteq S$.最后,近似作业仅需要得到近似的结果即可,例如图模式挖掘作业[34]、基于增量累积迭代的 PageRank 作业[35−37]等.该类作业不要求得到每个顶点或边的准确值,只需要一个近似的统计值或者排序,所以这类作业发生失效后仅需要容错机制将故障状态 s_e“修复”至某一弱有效状态(图 4 中 R_3 过程)后继续计算即可最终到达近似最终状态 s_a(图 4 中 PR_3 过程),并得到近似准确的结果.一般来说,自稳定作业、近似作业、局部可矫正作业、全

局可矫正作业和全局一致作业的容错难度依次增加,并且能够支持高难度作业,如全局一致作业的容错机制往往也能支持低难度作业,如自稳定作业.因此,可支持的最高难度作业类型成为衡量面向分布式图计算作业容错机制质量的重要指标之一.

1.3 失效模型和容错框架

故障(fault)是软件或硬件中被激活缺陷的最低级别抽象,错误(error)是指故障导致的系统内部状态的改变或丢失,失效(failure)则是指由于错误导致的系统行为与预期行为之间的偏差[38−40].这三者的关系如图 5 所示.

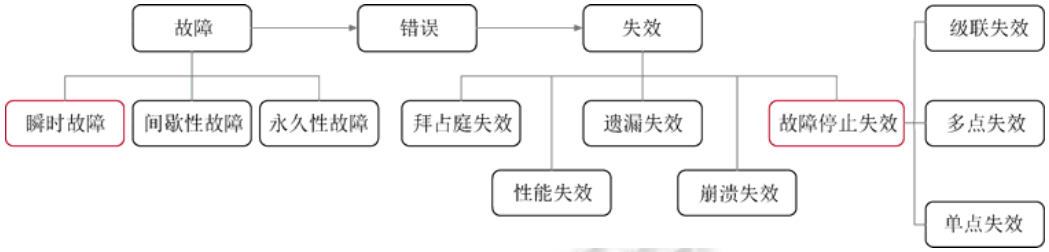

Fig.5 Failure model

图 5 失效模型

具体而言,故障按照其发生的时机通常可以分为瞬时故障(transient fault)、间歇性故障(intermittent fault)和永久性故障(permanent fault)这 3 类,其中,瞬时故障是由于某些影响系统运行的条件临时改变而导致的,如网络波动、线程竞争等,故障只会导致当前任务失效,通常无需修改代码,当任务重新运行时,故障很可能不会再次发生;间歇性故障则以不规则间隔随机发生,其极难诊断和永久解决,如由于温度波动而无法正常工作的硬盘会在某个阶段又恢复正常;永久性故障在其软件缺陷未解决之前将一直存在,如代码的逻辑 bug 等.在面向分布式图计算作业的容错中主要处理的是由瞬时故障引起的失效.

外部观察者观察到系统行为的偏差后,如未得到计算结果或得到了错误的计算结果之后,对系统失效原因进行猜想假设.根据假设的失效原因的不同,从复杂到简单,失效可被分为拜占庭失效(Byzantine failure)、性能失效(performance failure)、遗漏失效(omission failure)、崩溃失效(crash failure)和故障停止失效(fail-stop failure).在分布式图计算中,拜占庭失效的原因可以是任意错误,包括节点内部状态出错、传输数据损坏或网络路由错误等;性能失效的原因是集群中某计算节点 M_i 未按时从其他节点收到所需消息;而遗漏失效的原因则是该节点 M_i 根本未从其他节点收到所需的消息;崩溃失效则进一步简化,即该节点 M_i 未从某个其他节点 M_j 收到所需消息的原因是后者 M_j 崩溃了;而故障停止失效则是假设了当集群中节点 M_j 崩溃时,其他节点均知道其崩溃了,它强调了故障不会造成错误的系统状态改变,体现在两个方面:一是在节点 M_j 崩溃前,故障的发生未导致其遗漏须发送的消息或发送错误的消息;二是在节点 M_j 崩溃后,其他节点未在缺少部分消息的情况下继续运算,从而产生错误的作业状态.在故障停止失效中,系统只会丢失部分内部状态,而所有幸存的内部状态均是正确的.在分布式图计算作业中,故障停止失效被进一步细分,从复杂到简单,可以分为级联失效(cascading failure)、多点失效(multi-node failure)和单点失效(single-node failure).其中,单点失效是指同一时间只有一个节点崩溃了;多点失效则是指同时存在多个节点崩溃;而级联失效则是指在对崩溃节点的状态进行恢复的过程中,集群中又有节点崩溃了.容错机制的实现难度与失效的复杂度通常是正相关的,并且能够处理复杂失效的容错机制往往也能够处理较简单的失效,因此,可处理的最复杂失效类型成为衡量面向分布式图计算作业容错机制质量的重要指标之一.

面向分布式图计算作业的容错机制通常是针对预期的失效类型来设计相应的状态冗余方法和失效恢复方法,通常包括 3 个组件:状态冗余、失效检测和失效恢复.图 6 展示了作业全局状态和容错组件间的相互作用.当作业正常运行时,一方面,状态冗余组件对作业全局状态即正常状态进行冗余保存,生成冗余状态,并定期地对其进行更新;另一方面,失效检测组件也定期地对作业的运行状态进行检测,当检测到失效发生时,作业全局状态即成为失效状态.同时,失效恢复组件被启用.失效恢复组件利用失效状态和冗余状态对作业进行恢复,当恢

复成功后,作业全局状态即成为正常状态.

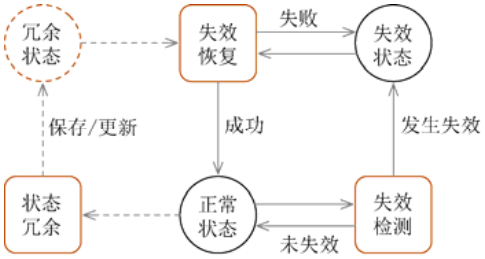

Fig.6 Global states in distributed graph processing jobs and fault tolerant framework

图 6 分布式图计算作业全局状态和容错框架

值得注意的是,在分布式图计算中,由于作业运行时所产生的天然冗余和作业本身鲁棒性的不同,容错中的状态冗余方法也多种多样,基本上可归为 3 类:检查点(checkpointing)、日志(logging)和复制(replication)方法,并且,还有一些作业的容错甚至不进行状态冗余.因此,依据容错所采用的作业状态冗余方法的不同,面向分布式图计算作业的容错可分为 4 类:(1) 基于检查点的容错;(2) 基于日志的容错;(3) 基于复制的容错;(4) 基于算法补偿的容错.具体各类容错的状态冗余方法和相应的失效恢复方法将在第 4 节结合相关工作进一步加以论述.

此外,绝大多数面向分布式图计算作业的容错是针对故障停止失效的,在这种情况下,简单的心跳机制就能对失效进行有效的检测,而极个别能够处理拜占庭失效的容错,其失效检测相对复杂,具体的将在第 4 节结合相关工作进行介绍.

2 目标和挑战

本节首先介绍面向分布式图计算作业容错技术的三大优化目标——更低的容错成本、更高的失效恢复效率和更好的容错质量,并分析了实现这 3 个优化目标所面临的难点和挑战,最后,提出基于成本、效率和质量这 3 个维度的面向分布式图计算作业的容错技术评估框架.

2.1 优化目标

如上所述,随着分布式图计算作业处理的图数据日益庞大、计算日益复杂,其对性能和可扩展性(scalability)的追求日益迫切,但同时,分布式图计算作业面临的故障发生频率高、传播快、影响大等问题,对其性能和可扩展性的提高也造成了巨大的威胁.为了缓解这一矛盾,更低的容错成本、更高的失效恢复效率和更好的容错质量是面向分布式图计算作业的容错技术不懈的追求之一.而这 3 点优化目标则分别涉及到设计容错机制时需要考虑的 3 个维度:成本、效率和质量.

(1) 成本是指容错机制的实现和启用对分布式图计算作业的实现和正常运行带来的负面影响,一般包括人力成本和资源成本.人力成本通过实现分布式图计算作业容错时用户的参与度来衡量,对用户不透明的容错机制往往比对用户透明的容错机制要消耗更多的人力成本;资源成本则通常是指容错机制在分布式图计算作业正常运行时所消耗的 CPU、内存、网络带宽和磁盘 I/O 等计算机资源,这通常会给分布式图计算作业的正常运行带来额外的性能开销,并常用因此增加的运行时间来衡量;

(2) 效率是指失效发生后容错机制对作业运行的正面影响,包括失效恢复(failure recovery)阶段和后恢复(post recovery)运行阶段,通常使用失效恢复时间和后恢复运行时间来衡量容错机制的效率.值得一提的是,在相当多的支持全局一致作业的容错工作中,将失效恢复看作是从故障状态 s_e 恢复至失效前最新状态 s_{la} 而非其他任一全局一致状态的过程,并且,由于失效前最新状态 s_{la} 至最终状态 s_a 的运行时间在发生失效的情况和正常(failure-free)情况下相同,故直接使用失效恢复时间来衡量容错机制的效率;

(3) 质量则是指面向分布式图计算作业的容错机制的容错能力.如上一节所述,可支持的作业类型和可处理的失效类型分别是衡量容错机制容错质量的两个重要指标.其中,可处理的失效类型越复杂,可支持的作业类

型容错恢复难度越高,则容错能力越强、质量越高.

2.2 难点和挑战

在实际实现面向分布式图计算作业容错机制时分别面临成本、效率和质量这 3 方面的难点和挑战.

(1) 在成本方面,图计算的并行计算过程由数据驱动,过程中节点工作负载动态变化,用户难以在作业提交前预知其负载情况;并且,图数据的局部性差,分布式图计算作业需要占用大量通信和计算资源,对资源竞争敏感.因此,在何时、以怎样的方式进行状态冗余备份,能够减少消耗的资源成本、减少容错机制给分布式图计算作业正常运行带来的额外性能开销,同时又不显著提高人力成本即保持对用户的高透明度成为一个挑战;

(2) 在效率方面,分布式图计算作业任务间的依赖关系复杂,导致失效发生后,幸存节点难以继续运行,需要挂起等待甚至回滚,造成单点恢复瓶颈以及重复计算等问题,增加了失效恢复过程的时间,大量资源被闲置和浪费;在非全局一致作业容错过程中,失效节点上的"修复"状态和原状态 s_{la} 的差异还可能会带来后恢复运行时间延长的副作用.因此,如何在失效后快速恢复分布式图计算作业的状态使其继续运行,同时不显著增加后恢复运行时长成为第 2 个挑战;

(3) 在质量方面,如前文所述,分布式集群规模庞大、图计算框架抽象层次高,加之图计算作业计算负载动态变化且运行周期长等因素,使得分布式图计算作业面临着分布式图计算系统内外各种来源的非确定性,故障和失效频繁、复杂且种类多样.因此,如何分辨不同的作业类型,在满足其个性化结果准确度需求的基础上支持处理更复杂的故障和失效类型成为第 3 个挑战.

2.3 评估框架

然而,即便与分布式图计算框架的功能结构、图计算作业的运行特点进行紧密耦合,充分利用其提供的优化空间,面向分布式图计算作业的容错技术依然难以同时在成本、效率和质量这 3 个维度优化到极致.如图 7 所示,在中心 3 个圆形相交的区域,成本、效率和质量之间构成了一个"不可能三角",即不存在成本低、效率高和质量好的容错机制,更多时候需要考虑和利用分布式图计算作业的具体容错需求,在成本、效率和质量这三者之间进行权衡和取舍.

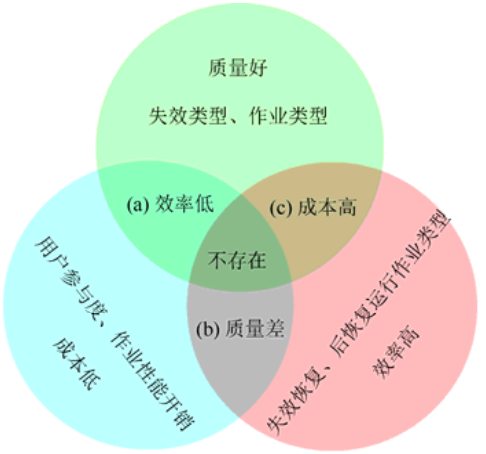

Fig.7 "Impossible Triangle" of cost, efficiency and quality of fault tolerance

图 7 成本、效率和质量的"不可能三角"

如图 7(a)中重叠区域所示,能够处理更复杂的失效类型(高质量)并消耗更少资源(低成本)的容错机制会导致恢复效率的降低.例如,采用长间隔的检查点容错机制会减少保存快照的数量,但会导致失效后需要更长的时间从最近快照恢复至失效前最新状态[41];如图 7(b)中重叠区域所示,能够冗余更少作业状态(低成本)并提供更高恢复效率(高效率)的容错机制会导致仅能支持相对较少的作业类型(低质量).例如,基于算法补偿的容错能够

实现零性能开销和快速恢复,但仅能支持有限的具有特殊性质的作业[36,37];如图 7(c)中重叠区域所示,能够处理更复杂的失效类型(高质量)并能提供更高恢复效率(高效率)的容错机制也会消耗更多的资源(高成本).例如,被动复制机制仅能处理崩溃失效,而采用 $3f+1$ 副本的主动复制机制则能一定程度地处理拜占庭失效[42].

以上性质在不同的容错工作中会反复体现,因此,本文在对一个面向分布式图计算作业的容错机制进行评估时,综合考量了成本、效率和质量这 3 个维度,同时,相应的有 6 个衡量指标:作业性能开销、用户透明度、失效恢复时间、后恢复运行时间、可支持的最难作业类型、可处理的最复杂失效类型.

3 发展脉络

分布式图计算的容错实现方式与其计算逻辑的实现方式息息相关,随着分布式计算框架的发展,分布式图计算的容错实现方式也随之发生变化,先后可以分为 3 个阶段,分别是面向专用计算作业的容错、面向通用大数据处理作业的容错和面向分布式图计算作业的容错,如图 8 所示.

Fig.8 Development of fault tolerance in distributed graph processing

图 8 分布式图计算的容错技术发展脉络

(1) 面向专用计算作业的容错.这一阶段最早可追溯到 20 世纪 80 年代的高性能计算领域,用户基于底层库编写专用程序来实现分布式图计算作业计算逻辑,用户须自己处理计算并行、图切分、节点通信等问题,同时计算资源和计算作业的分布式化对容错提出了新的需求,引起了一波面向专用计算作业的容错技术的研究热潮.在这一过程中,图计算作业作为高性能计算作业的子集,其容错也得到了一定的发展,其容错技术主要是基于硬件[15,16]、操作系统[17−20]或运行时库[21−24]的实现,以及少量基于用户应用程序[30,31]的实现.这些技术的局限性表现在:一方面,基于硬件、操作系统或运行时库的容错实现难以移植,并且粗粒度的应用状态冗余管理,无法满足分布式图计算作业的性能需求;另一方面,基于用户应用程序的容错实现使原本就复杂的专用程序变得更为复杂,用户在处理计算并行、图切分、节点通信等问题的同时还需要考虑容错,可能会引入更多的不确定性而导致故障的发生;

(2) 面向通用大数据处理作业的容错.这一阶段以 MapReduce[27]的提出为标志,通用大数据处理框架(即通用分布式数据流框架)开始蓬勃发展,用户基于通用大数据处理框架,如 Hadoop、Spark 等提供的数据流算子实现计算逻辑,同时,通用大数据处理框架在设计伊始便考虑到容错这一问题,其容错技术往往与框架的功能结构紧密耦合[27,28,43−45],如 Spark 的弹性分布式数据集(RDD)和血统(lineage),能够基于较细粒度的数据或者任务进行冗余状态管理,从而减少性能开销.图计算作业作为大数据处理作业中相当重要的一部分,其容错也受益于这样的设计,但同时也受限于框架通用性的考量,难以针对图计算作业任务粒度细、任务间的依赖关系复杂、容错需求多样等特点进一步优化;

(3) 面向分布式图计算作业的容错.随着对图数据和图计算的日益重视,这一阶段以第 1 个专用分布式图计算框架——Pregel[4]的提出为分界点,用户开始借助分布式图计算框架提供的编程抽象更方便地实现了分布式图计算作业的计算逻辑.相应地,其容错的数据粒度从 MapReduce 中的任务和 RDD 中的分区等进一步细化到了顶点状态和顶点间消息,并且,随着研究人员对分布式图计算框架、图计算作业鲁棒性和失效特征的深入认识和分析,容错在多个方向上进行了个性化的优化和发展.目前,面向分布式图计算作业容错的优势在于,其一方

面能够通过更细粒度的状态冗余管理实现更低的容错成本、更高的容错效率和更好的容错质量,另一方面能够通过结合不同作业的容错需求在容错的成本、效率和质量之间实现更灵活而准确的权衡和取舍.

因此,面向分布式图计算作业的容错技术是目前分布式图计算容错的主流技术.

4 面向分布式图计算作业的容错技术

如前面第 1.3 节所述,根据容错所采用的作业状态冗余方法,面向分布式图计算作业的容错可分为 4 类.(1) 基于检查点的容错;(2) 基于日志的容错;(3) 基于复制的容错;(4) 基于算法补偿的容错.大体来讲,基于检查点的容错应用得最为广泛,可以应用于基于同步和异步调度的图计算作业,并实现较高的容错质量;而基于日志和复制的容错仅能应用于基于同步调度机制的作业.其中,基于日志的容错和基于检查点的容错通常是正交的,前者通常可以作为后者的重要补充,从而提高整体的恢复效率,例如 Spark 中的容错就组合使用了基于检查点的容错机制和基于日志的容错机制——在 lineage 过长时通过检查点制作快照以缩短 lineage 的长度,减少重播,加速失效恢复效率.而基于复制的容错机制能够在容错效率方面实现更快的失效恢复,或者能够在容错质量方面处理更复杂的失效类型,如拜占庭失效等.在这种情境下,基于复制的容错机制是基于检查点容错的良好替代;此外,基于算法补偿的容错则依据补偿算法的不同可以应用在基于同步或异步调度的作业中,能够在容忍一定容错质量损失的情况下,实现容错质量方面的零性能开销和容错效率方面的快速恢复.该技术相比前 3 类容错技术有着极大的优势,但其适用范围相对较小,仅能用于某些特定类型的作业,并且要求用户对作业有深入的理解.本节将结合具体的工作介绍在实现具体的容错机制时如何充分利用分布式图计算框架的功能结构和图计算作业的运行特点,并通过 3 个维度、6 个指标的评价体系分析各个容错机制在成本、效率和质量方面的侧重和取舍.

4.1 基于检查点的容错

检查点是指在程序运行特定位置保存用于故障恢复的必要状态信息的过程[46].在面向分布式图计算作业容错机制中,根据保存快照一致性的不同,基于检查点的容错可以分为全局一致检查点容错和局部一致检查点容错.其中,前者应用得最为广泛,同时也成为后续大多数研究工作进行比较和优化的基准.在同步调度机制中,如图 9 所示,全局一致检查点容错的基本思路是在作业正常运行时,按照预先指定的超步间隔,保存系统全局一致快照到可靠存储中,包括顶点/边值、消息等;而在失效恢复时,重启或新启的节点加载失效节点的图拓扑结构和最新的快照数据,幸存节点也回滚并加载相应的全局一致快照,使得作业恢复至从最近保存的全局一致状态开始运行,以避免作业从头重新运行.

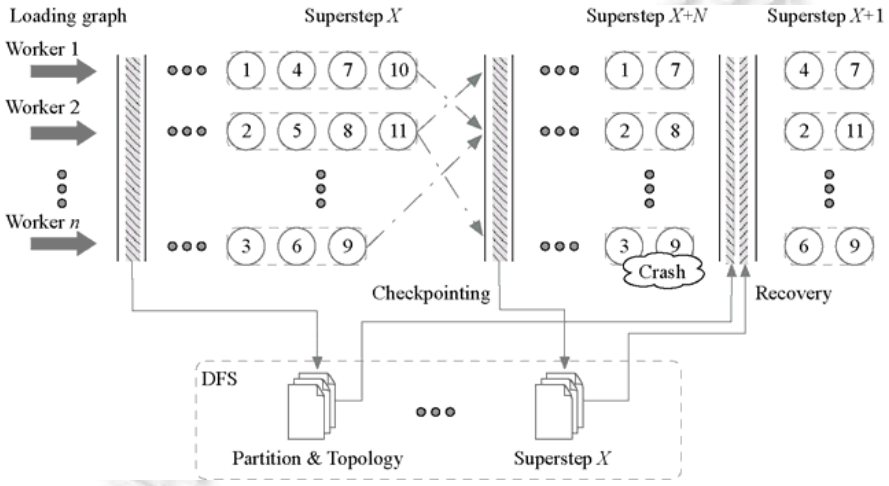

Fig.9 An example of fault tolerance based on global consistency checkpointing

图 9 全局一致检查点容错

Pregel[4]是采用该容错技术的典型代表,而异步调度机制由于没有全局同步栅帮助容错机制获取全局一致快照,GraphLab[6]和 PowerGraph[7]便以一定的时间间隔挂起所有计算节点同步保存全局一致快照,或是基于 Chandy-Lamport 异步保存的方法获得全局一致快照.相较全局一致检查点容错,局部一致检查点容错技术应用得较少,但后者也有其独特的优势.下文结合具体的研究工作对这两种检查点容错机制进行更为详细的分析和对比.

4.1.1 全局一致检查点容错

首先,图计算作业运行过程中的工作负载会动态变化,例如一个 PageRank 作业的大多数顶点在前 10 轮迭代过程中就基本收敛了,后续的迭代运算仅有少量顶点参加[41].而基于预先指定的固定间隔的检查点技术可能导致失效恢复的开销小于一次快照保存的开销,这对系统整体性能来说是不划算的.对这个问题有两种解决思路:一是随着计算负载的变化动态地调整检查点间隔,随着间隔的变大,失效恢复的开销也随之变大;二是调整检查点保存快照的大小,当计算负载变小时,降低检查点保存快照的开销.于是,针对第 1 种解决思路,文献[41]提出了基于作业运行时信息动态调整快照保存间隔的检查点容错机制.在采用同步调度机制的分布式图计算框架中,作业计算过程被全局同步栅分割成一个超步序列.在作业正常运行时,master 节点分别收集并记录每次超步的计算时长和每次检查点的快照写入保存时长,并在每次进入新的超步后,比较自最新快照至当前迭代的累积计算时长和历史快照的平均保存时长(用于估计当前快照的保存时长),当前者大于后者时,认为此后若是发生失效,则其失效恢复的开销将大于保存一次快照的开销,master 节点指示 worker 节点保存全局一致快照;否则,不保存.最终,基于动态间隔的检查点容错机制相比基于静态间隔的检查点容错机制,使得作业正常运行(failure-free)的性能最高提升了 8.5 倍,而付出的代价仅仅是失效恢复时间略微延长.针对第 2 种解决思路,文献[47]提出了一种增量检查点容错机制.当作业中参与运算的顶点规模小于一定阈值时,检查点操作时不再保存包含所有顶点状态的全量快照,而是保存仅发生改变的顶点状态的增量快照;在失效恢复阶段,加载最新的全量快照和之后的所有增量快照进行合并,并基于合并后的快照状态继续恢复作业状态并继续计算.最终,该增量检查点容错机制由于要对全量快照和增量快照进行合并,导致失效恢复效率降低,但增量快照减少了检查点的性能开销,作业的整体性能大幅提升.

其次,在进行全局一致检查点操作时,快照的大小也会影响作业正常执行的性能.在类 Pregel 分布式图计算系统中,每次检查点操作在当前迭代开始计算前保存:每个顶点 v 的状态 $\pi^{(i-1)}(v)$,包括顶点值 $a^{(i-1)}(v)$、顶点激活状态 $on^{(i-1)}(v)$、甚至是顶点邻接表$\Gamma^{(i-1)}(v)$(拓扑结构)和每个顶点 v 收到的消息 $M_{in}^{i}(v)$. 其中,消息的数量是与图中边的数量正相关的,并且,由于边的数量通常远远大于顶点数量,因此在检查点操作时保存消息会极大地占用磁盘 I/O 和网络通信资源,严重影响作业的正常执行性能.文献[48]提出一种轻量级的检查点容错机制 LWCP (lightweight checkpointing),其利用 MPI-3 标准中的弹性扩展 ULFM(user-level failure mitigation)在不改变 Vertex Centric 编程模型的情况下,将顶点更新和消息生成逻辑解耦.如图 10 所示.

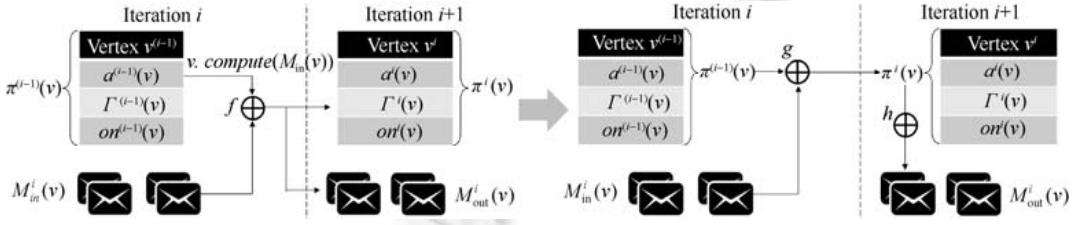

Fig.10 Decoupling of vertex update and message generation[48]

图 10 顶点更新和消息生成逻辑解耦[48]

顶点更新 $\pi^{i}(v)$和消息生成 $M_{out}^{i}(v)$ 的逻辑从单段式的 $\left(\pi^{i}(v),M_{out}^{i}(v)\right)=f\left(\pi^{(i-1)}(v),M_{in}^{i}(v)\right)$ 转换成了双段式的 $\pi^{i}(v)=g\left(\pi^{(i-1)}(v),M_{in}^{i}(v)\right);\pi^{i}(v)=g\left(\pi^{(i-1)}(v),M_{in}^{i}(v)\right);M_{out}^{i}(v)=h\left(\pi^{i}(v)\right)$, 即新的消息可以仅依赖更新后的顶点

值计算得出,而不依赖于顶点之前收到的消息.基于此,在保存快照时仅需要保存顶点值 $a^{(i-1)}(v)$和顶点激活状态 $on^{(i-1)}(v)$而无需保存消息 $M_{in}^{i}(v)$,缩小了快照体积.同时在失效恢复时,容错机制可以更快地加载所需快照,而所需的消息则可以根据加载的顶点值计算得出.最终,基于 LWCP 方法,PageRank 作业每次检查点保存快照所需时间在 WebUK 数据集上从 65.18s 减少到 2.14s,在 WebBase 数据集上,从 27.45s 减少到 2.16s,检查点操作性能提升 10 余倍,但在失效恢复过程中,由于加载最近的快照后需要额外生成消息,这个过程所需时间分别增加了 35s 和 15s 左右,考虑到作业运行过程中失效频率一般小于检查点操作频率,为了保障作业的整体运行性能,牺牲一部分的失效恢复效率是值得的.此外,LWCP 在使用时需要用户少量参与,一方面,用户需要在实现 compute()方法时将顶点更新逻辑和消息生成逻辑分开,另一方面,用户需编写用户自定义方法 LWCPable()对作业运行中少数不适宜采用 LWCP 的超步进行筛选.同样的轻量级检查点技术也应用在文献[49]中的 Seraph 框架,并且因为 Seraph 的 SG 编程模型本身就提供了 compute()和 generate()两个接口分别实现顶点更新逻辑和消息生成逻辑,因此在用 Seraph 框架实现的分布式图计算作业中,轻量级检查点容错机制对用户是透明的.

然而,上述全局一致检查点需要阻塞作业的正常运行,待全局一致快照保存结束后作业才能继续计算,这种阻塞式的检查点给分布式图计算作业的正常运行带来了较大的性能开销;而现有的非阻塞检查点仅适用于保存不变(immutable)数据,例如 Spark 中对 RDD 的非阻塞保存[28].于是,文献[50,51]通过对基于数据流机制通信的分布式图计算作业进行分析,发现在迭代执行 join-groupBy-aggregation 数据流模版的过程中,每轮迭代更新后的顶点值并不是立即就被消费使用了,于是提出了将检查点写快照的过程与超步结束时顶点更新生成的过程重叠(overlap)并行的尾部检查点(tail checkpointing),即一边更新顶点状态一边保存快照和与超步开始时顶点被消费使用的过程重叠并行的头部检查点(head checkpointing),即一边消费顶点状态一边保存快照.若检查点操作未能提前于顶点更新生成过程或顶点被消费过程结束之时完成,仍要进行阻塞以完成快照保存,但该方法仍然有效地减少了检查点操作带来的性能开销,尤其是头部检查点相较阻塞式检查点将性能开销从 33%降到了 13%.

4.1.2 局部一致检查点容错

上述全局一致检查点容错与基于同步调度机制的图计算作业是契合的.而基于异步调度机制的图计算作业的计算过程本质上是非确定性的(non-deterministic)——即使是从同一个全局一致状态开始计算,结果通常也会存在些许差异,这降低了失效恢复中作业恢复至最近全局一致快照的必要性;而且全局一致检查点需要集群中各计算节点将快照在同一时间段内保存至可靠存储中,这增加了峰值带宽利用率,容易造成网络争用和性能下降.文献[52]提出一种基于局部一致快照进行异步保存和失效恢复的方法.首先,文献发现异步调度机制对读写依赖的放松导致分布式图计算作业无需保持全局一致性而只需保持 PR 一致性(progressive reads consistent)即可保证结果的正确.于是,在正常执行过程中,由各计算节点自身决定保存内部状态的时机,这减少了同时保存的快照数量,从而减少了网络争用.同时,由于局部一致快照节点内状态一致而节点间状态不一致的特点,master 节点需要依据快照间的依赖关系保存 PR 顺序,用于在多点失效情况下决定失效节点逐次恢复的顺序.基于此,如图 11 所示,在失效恢复阶段,所有失效节点需从 S^f 分别加载其最近的局部一致快照,然后按照 PR 顺序对失效节点逐点恢复:(1) 对第 1 个失效节点 a 用快照 s_a 初始化;(2) 用其他节点上的边界顶点值(包括剩余失效节点的快照 S^f-s_a 和所有幸存节点的状态 E^{cf})更新其子图状态(图中虚线);(3) 在该节点上进行局部计算至收敛,使更新后的边界值传播至该节点内的非边界顶点上,至此,该失效节点与其他节点保持了 PR 一致性;(4) 接着将节点 a 和在 PR 顺序中紧随其后的失效节点 b 看作一个整体,重复以上对失效节点 a 的恢复过程,直至图中所有节点均满足 PR 一致性,多点失效的恢复过程结束.最终,在实验中局部一致的异步检查点相较全局一致检查点的峰值带宽利用率降低了 22%~51%,减少了对作业正常运行造成的性能开销;并且在失效恢复阶段,该方法可以实现受限恢复,幸存节点不用回滚,失效恢复及后恢复运行阶段较基于全局一致检查点的全局回滚恢复方法加速了 1.5~3.2 倍;然而,该方法的容错质量较全局一致检查点容错方法有所降低,仅能支持自稳定作业、近似作业和局部可矫正作业,不能支持全局可矫正作业和全局一致作业.

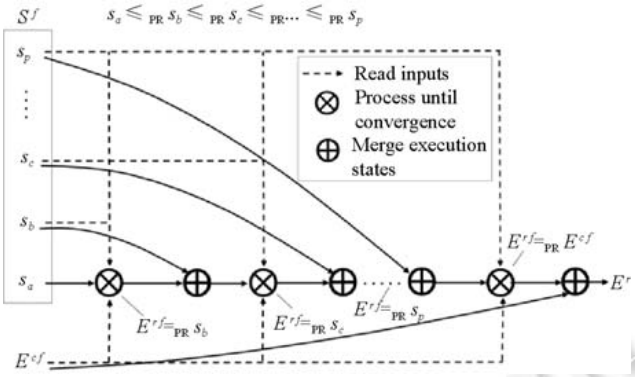

Fig.11 Recovery of multi-node failure[52]

图 11 多点失效的恢复过程[52]

4.1.3 基于检查点的容错对比分析

表 1 给出上述工作的优化对基于检查点的容错在成本、效率和质量这 3 个维度 6 个指标上的影响(以 Pregel、GraphLab 和 PowerGraph 中的全局一致检查点容错[4−6]为基准),并进行了对比.从整体上看,基于检查点的容错机制都只支持对任意数量的多点失效的恢复,而不支持对级联失效的逐步恢复(recovery progressively),当级联失效发生时已部分恢复(partially recovered)的计算会再次丢失,这一问题会在后文基于日志的容错技术中给出解决办法;然后发现相关工作中全局一致检查点容错的优化重点集中在其性能开销的降低;而局部一致检查点容错的优化重点则在其恢复效率的提高.具体而言,在全局一致检查点容错技术中,文献[41,47−49]分别利用了作业的运行时特性——负载动态变化和作业的编程模型——将顶点更新和消息生成解耦,实现了性能开销的大幅度降低,但其代价则是失效恢复时间的相应增加,并且在文献[48]中还额外增加了一定的人力成本;而文献[50,51]则通过将快照保存过程与数据流计算过程的重合,小幅度地降低了检查点的性能开销,并且未带来其他副作用.而在局部一致检查点中,文献[52]利用异步调度机制的特性,不用将失效状态恢复至全局一致状态,从而显著提高了失效恢复和后恢复运行的效率,并且,该方法的一个副产物是小幅降低了检查点的性能开销,然而其代价是容错质量的降低,无法对全局一致作业和全局可矫正作业提供容错支持.

Table 1 Comparison of fault tolerance based on checkpointing

表 1 基于检查点的容错机制对比

名称	文献	成本		效率		质量	
		性能开销	用户参与度	失效恢复时间	后恢复运行时间	可支持的作业类型(≤)	可处理的失效类型(≤)
全局一致检查点容错	[41,47,49]	显著降低	对用户透明	增加	不变	全局一致作业	任意数量的多点失效
	[48]	显著降低	需用户少量参与	增加	不变	全局一致作业	任意数量的多点失效
	[50,51]	降低	对用户透明	不变	不变	全局一致作业	任意数量的多点失效
局部一致检查点容错	[52]	降低	对用户透明	(合计)显著减少		局部可矫正作业	任意数量的多点失效

4.2 基于日志的容错

基于分段确定性假设(piecewise deterministic assumption),一个任务的执行可以被建模成一个状态间隔(state intervals)序列,每个间隔序列从一个非确定性事件开始.日志(logging)是保存非确定性事件决定性因素(the determinants of the nondeterministic event)的过程[53].通常,基于日志的容错与基于检查点的容错是正交的,前者是对后者的重要补充,前者避免了幸存节点的回滚(空间维度上),后者避免了作业从头重新运行(时间维度上),于是在很多面向分布式图计算作业的容错工作中,这两种机制搭配使用.其中,基于日志的容错的基本思路

是在作业正常运行期间,记录任务的非确定性事件,如在GraphX(Spark)中记录RDD分区间转换操作的Lineage;在失效恢复时,通过日志完成失效任务从最近快照状态至失效前最新状态的重播,而未失效任务则不用回滚重新计算,实现受限恢复.另外,在实际使用中,由于采用异步调度机制的分布式图计算作业计算过程的非确定性,日志记录是高消耗和不必要的,因此基于日志的容错技术多用于确定性更高的采用同步调度机制的分布式图计算作业.在文献[4]中,Pregel基于BSP同步调度机制对作业中的顶点任务进行调度运算.从顶点任务的角度来看,每个任务接收的消息都是非确定性事件,但从计算节点的角度来看,由于同一计算节点上顶点任务之间的消息在恢复期间会重新产生并且是确定性的,因此仅需在每个超步对来自其他节点的消息进行记录即可,然后在失效恢复时,幸存节点向重启或新启的节点重新发送保存的消息.此外,由于受限恢复的原因,基于日志的容错技术天然地支持对任意数量的级联失效的恢复,当新的级联失效发生时,正在恢复的节点M_i暂停恢复,并联合其他迭代进度大于M_i的节点对新的失效节点进行受限恢复,直至将其恢复至与M_i同样的迭代进度,然后将两者从逻辑上看作一个整体,继续进行恢复,从而实现逐步恢复.

在现有的研究工作中,基于日志的容错可以分为基于并行受限恢复方法的容错和基于轻量级日志保存方法的容错.下文结合具体的研究工作对这两种技术进行详细的分析和比较.

4.2.1 基于并行受限恢复的容错

一方面,由于图计算作业中任务间相互依赖关系复杂,失效发生后,幸存节点无法继续计算,计算资源被闲置;另一方面,传统的重启或新启节点的恢复方法会导致失效恢复过程存在单点瓶颈,从而增加失效恢复的时间,进而造成了集群中计算资源更长时间的闲置.因此,快速的恢复方法是必要的.一种直觉的方法是利用闲置的计算资源进行并行受限恢复,文献[54,55]实现了基于失效图分区重调度的快速并行恢复方法.该方法基于 master 生成的重调度方案,将失效节点上的多个图分区重调度至被选中的幸存节点上,实现并行恢复,其中,重调度方案质量的好坏会直接影响恢复的效率.于是,为了生成一个高质量的重调度方案,该方法在作业正常运行期间,worker 节点会统计每个超步中各图分区的计算成本和图分区间的通信成本,并在全局同步栅期间,将相关统计数据汇总到master节点;然后,Lu和Shen等人[54,55]提出了一种综合考虑了计算和通信成本、可启发式地快速遍历生成综合成本“最小”的重调度方案的算法;在失效恢复期间,master基于收集到的数据和启发式算法快速生成重调度方案,并基于此,管理失效恢复过程.最终,采用快速并行受限恢复方法的日志容错机制结合检查点容错机制,相比单独的检查点容错技术在恢复速度上提升了12~30倍,但同时,日志的I/O也给系统正常运行带来了较大的性能开销;并行恢复方法相比重启或新启节点的恢复方法,在恢复效率上也有大幅提高,但在系统正常运行阶段,收集统计数据的行为也给作业性能带来一定的影响.

上述工作在基于Vertex-Centric编程模型和消息传递通信机制的分布式图计算作业中实现受限恢复是十分自然的,其工作重点也更多地集中在了并行恢复的“快速”上.然而,该方法并不适用于基于数据流通信机制的分布式图计算作业.在这种作业中,仅记录join操作产生的用于groupBy的消息是不能实现受限恢复的,aggregation操作产生的分区间宽依赖关系将引入 shuffle 操作,使得幸存节点上的分区同样参与运算,导致完全恢复(complete recovery).于是,文献[50,51]提出一种通过数据一致分区(co-partioned)和日志相结合的容错方法以实现快速的受限恢复.在正常执行期间,每个计算节点保存在 groupBy 操作中向其他节点发送的消息,记作 Log;失效恢复阶段,如图12所示,图中Vertex的第1列是顶点索引、第2列是顶点值,而Message和Log的第1列是目标顶点索引,第2列是传递的消息.该图展示了失效节点上的分区(包含顶点3、9、6)在幸存节点上进行groupBy操作时利用本地保存的消息进行并行受限恢复的过程.其中,丢失的Vertex和Edge可以分别通过最近快照和可靠存储中的原始图数据构建,因此,join操作不会导致完全恢复,而对aggregation操作来说,为了不引入宽依赖关系,需要将Vertex和Neighbor进行一致分区,从而将其转化为窄依赖关系.至此,才实现了基于数据流通信机制的分布式图计算作业的受限恢复.而该方法的“快速”主要体现在两个方面:一是受限恢复过程的并行化,二是将宽依赖转化为窄依赖后,节省了节点间数据交换的时间.最终,该方法相比完全恢复,节省了 48%~81%的失效恢复时间,显著提升了失效恢复效率.

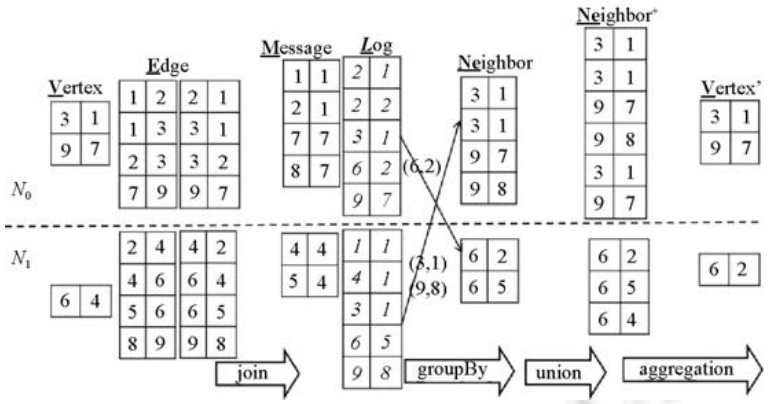

Fig.12 An example of parallel confined recovery[50,51]

图 12 并行受限恢复举例[50,51]

4.2.2 基于轻量级日志保存的容错

由于消息的数量和边的数量呈正相关,并且每轮迭代都要进行消息记录,这会占用大量的磁盘,并且对过期(最近检查点之前)的消息进行删除是耗时的.文献[48]在轻量级的检查点容错 LWCP 基础上提出了基于轻量级的日志保存 LWLog(LightWeight logging)的容错.基于图 10 中同样的顶点更新和消息生成逻辑解耦方法,在正常运行期间,LWLog 方法仅保存顶点值及标志该顶点值是否生成消息的标识位,将每个超步保存的数据量从 O(*E*)降到了 O(*V*),大大减少了磁盘占用.而在失效恢复期间,幸存节点需要在加载保存在本地的顶点值后,生成失效节点所需的消息并发送,但是由于幸存节点生成消息的过程与失效节点的重新计算过程重叠,恢复效率也因此未有明显下降.最终,因为磁盘占用的减少,清理过期消息的耗时也相应大幅度减少,从而显著减少了对系统正常运行带来的性能开销.另外,出于与 LWCP 方法一样的原因,LWLog 方法也需要用户少量参与.

4.2.3 基于日志的容错对比分析

表 2 列举了上述工作的优化对基于日志的容错在成本、效率和质量这 3 个维度 6 个指标上的影响(以 Pregel 中的日志容错机制[4]为基准),并进行了对比.

Table 2 Comparison of fault tolerance based on logging

表 2 基于日志的容错机制对比

名称	文献	成本		效率		质量	
		性能开销	用户参与度	失效恢复时间	后恢复运行时间	可支持的作业类型(≤)	可处理的失效类型(≤)
基于并行受限恢复的容错	[54,55]	略微增加	对用户透明	显著减少	不变	全局一致作业	任意数量的级联失效
	[50,51]	不变	对用户透明	减少	不变	全局一致作业	任意数量的级联失效
基于轻量级日志保存的容错	[48]	显著降低	需用户少量参与	未有明显增加	不变	全局一致作业	任意数量的级联失效

首先,基于日志的容错都保持了高质量的容错,都能够支持全局一致作业并处理任意数量的级联失效.而基于并行受限恢复的容错更侧重于对恢复效率的优化,文献[50,51,54,55]均通过利用幸存节点闲置的计算资源来加速失效恢复过程,并充分利用不同分布式图计算框架的特点设计合适的快速受限恢复方法,以至于仅增加了很小甚至没有额外增加性能开销.而基于轻量级日志保存的容错更侧重于降低性能开销,并且,文献[48]为了获得更好的通用性,基于单段的 Vertex-Centric 编程模型来进行顶点更新和消息生成的解耦,因此也带来了些许人力成本.

4.3 基于复制的容错

复制(replication)是指多个任务副本(task replicas)同时运行在不同的资源上以保证任务运行成功的方法[56].在追求支持处理更复杂的失效类型或更高效的失效恢复的情景下,基于复制的容错是基于检查点容错的良好替代,但其同样也存在着天然不足——副本数量限制了可以处理的失效同时发生的最大数量.基于复制的容错包括主动复制容错和被动复制容错.前者要求多个任务副本接收同样的输入,并分别独立运行输出,比较输出结果,并利用投票或者重运行等方式检测失效并容错,而后者则仅要求主任务接收输入进行计算和输出,副本任务通过与主任务之间保持状态同步来备份主任务,当主任务失效时,副本任务可以恢复或直接替换主任务,保证作业继续运行.下文结合具体的研究工作对两种复制容错技术进行详细的分析和对比.

4.3.1 被动复制容错

全局一致检查点技术对大量状态的写操作以及不均衡的全局同步栅,给系统正常运行带来大量开销,通常是一个迭代时长的 8~31 倍[57].而失效恢复时,全局一致检查点容错需要回滚幸存节点重新计算,以及加载全局一致快照时带来的大量 I/O 操作和单节点恢复的 I/O 瓶颈,导致失效恢复效率低.而在现有的分布式图计算框架中,由于图分区,尤其是基于 edge-cut(搭配 ghost,如图 3(c)所示)或基于 vertex-cut(如图 3(d)所示)的图切分方法,导致系统中存在大量天然的顶点副本.此外,文献[57]发现,有些图计算作业仅需要近似准确的结果,且图数据中度越大的顶点一般对图计算作业的结果贡献越大,而图分区对计算和通信负载均衡的内在要求导致度越大的顶点其副本越多,从而越不容易在失效后完全丢失状态,进而降低了对结果准确度的影响.上述因素为被动复制容错机制替代检查点容错机制提供了充足的动机和条件.于是,文献[57]提出了仅利用幸存节点上的副本恢复失效节点的方法.首先对失效节点上存在天然副本的边界顶点进行恢复,而对没有副本的内部节点则重新初始化,并且在 GAS 模型中,Scatter 阶段失效节点内更新的状态丢失了,需要执行局部的 Scatter 操作同步状态.最终,由于该方法未在作业正常运行期间做任何额外的容错准备,所以其正常运行期间带来的性能开销为 0;而在失效恢复上,由于仅利用幸存的状态直接重构失效节点的状态,并且局部的 Scatter 操作所需要的时间也远小于一个完整超步的时间,所以失效恢复所需的时间极短.但是,当同时失效节点数增多时,一方面,后恢复运行时间可能随之增长,另一方面,结果的准确度也可能随之降低,所以,该容错方法仅能对多点失效和级联失效提供有限的支持.

而文献[58,59]则通过对多个图数据集和图算法进行分析后发现,图顶点中没有天然副本的仅占一小部分,并且还有一部分是“自私(selfish vertex)”顶点,这些“自私”顶点在图拓扑结构上体现为没有出边,在计算过程中体现为不参与其他任何顶点的计算更新.因此,仅需要为很小一部分没有天然副本的非“自私”顶点创建额外的容错副本即可.图 13 展示了在不同的图数据集中没有天然副本的顶点所占比例和额外创建的容错副本所占比例.于是,Wang 和 Chen 等人[58,59]扩展消息传递机制,在正常运行的每个超步对顶点和其副本(包括天然副本和容错副本)间状态进行同步;并且在每次全局同步栅前后进行快速失效检测,若失效,则通过分布在多个幸存节点上的副本并行地恢复丢失的节点状态.最终,该方法相对全局一致检查点方法在正常运行期间带来的性能开销显著减少,仅消耗了少量内存和通信,在实验中,该方法的性能开销仅为 0.6%~3.7%;在失效恢复阶段,由于失效节点上的顶点副本分布在不同的幸存节点上,所以通过多个幸存节点并行地重构失效节点的状态也极为快速,在实验中,该方法在 3.4s 内将超过 100 万个顶点的作业从失效中恢复,并且相比文献[57],该方法重构的失效状态与失效前状态一样,所以也不会增加作业的后恢复运行时间.但是,该方法仅能支持指定数量的多点失效,并且随着指定数量的增加,内存和通信开销也会随之增加.

4.3.2 主动复制容错

在生产环境中,除了故障停止失效外,分布式图计算作业还面临着大量的拜占庭失效,这类失效很难通过简单的心跳机制或者全局同步栅检测到.为了检测并处理一定程度的拜占庭失效,文献[60]提出了一种基于f+1副本的重试容错方法.首先,在加载图分区时,为每个分区创建f+1 个副本(容忍每次超步计算出现f个故障),并将分区副本放在不同的节点上运行;接着,每个超步完成计算后,各个 worker 节点将每个分区副本的状态映射成哈希码,并将其发送给 master 节点;最后由 master 节点比较分区副本间的哈希码是否相同,若相同,则进入下一个超

步,否则,回滚重新计算,直到副本哈希码相同或判定作业运行失败.最终,该方法为拜占庭失效提供了一定程度的容错能力,但其代价是严重的性能开销和低效率的失效恢复.

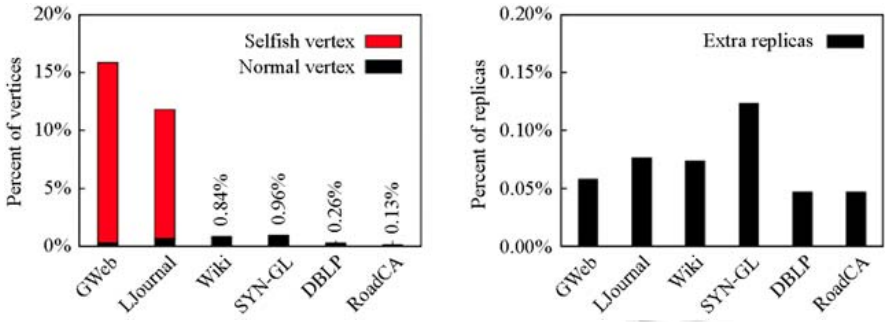

Fig.13 The percent of vertices without replicas, including normal and selfish vertex (left) and the percent of extra replicas for fault tolerance (right)[59,59]

图 13 无天然副本的顶点所占比例和额外创建的容错副本所占比例[58,59]

4.3.3 基于复制的容错对比分析

表 3 列举了上述工作的优化对基于复制的容错在成本、效率和质量这 3 个维度 6 个指标上的影响(以 Pregel 中的全局一致检查点容错[4]为基准),并进行了对比.整体来看,基于复制的容错均只用于采用同步调度机制的分布式图计算作业,并且在容错能力上均受到了副本数量的限制,所以均只支持处理有限数量的失效,而且随着副本数量的增加,付出的相应代价也随之增加——在文献[57]中是后恢复时间的增长和结果准确度的降低,在文献[58,59]中是内存和通信开销的成倍增长,在文献[60]中则是性能开销和失效恢复时间的急剧增长.而分开来看,则是被动复制容错技术和主动复制容错技术所要应用的场景不同,因此导致了两者在成本、效率和质量三者权衡上迥然不同的选择,前者对作业的整体性能极为看重,为此可以牺牲一定的容错质量,而后者对作业的结果准确度有极高的要求,为此可以付出极高的资源成本并接受极低的容错效率.

Table 3 Comparison of fault tolerance based on replication

表 3 基于复制的容错机制对比

名称	文献	成本		效率		质量	
		性能开销	用户参与度	失效恢复时间	后恢复运行时间	可支持的作业类型(≤)	可处理的失效类型(≤)
被动复制容错	[57]	零性能开销	对用户透明	显著减少	不确定	近似作业	有限数量的级联失效
	[58,59]	少量内存和通信开销	对用户透明	显著减少	不变	全局一致作业	有限数量的多点失效
主动复制容错	[60]	显著增加	对用户透明	显著增加	不变	全局一致作业	有限数量的拜占庭失效

4.4 基于算法补偿的容错

算法补偿(algorithmic compensation)是针对特定类型的图计算作业,在故障发生后,使用预定义的补偿算法直接生成丢失的状态的过程.基于算法补偿的容错均建立在对图计算作业特性的深入分析和理解的基础上.在正常运行阶段,不对任何状态和值做冗余备份,从而实现对作业正常运行的零性能开销;当失效发生后,按照预定义的补偿算法,直接生成丢失顶点的状态(往往不会是全局一致状态),幸存节点不回滚,作业继续运行.按照预定义的补偿算法的存在形式可将基于算法补偿的容错分为内置补偿算法容错和自定义补偿算法容错.前者将补偿算法嵌在分布式图计算框架之内,用以支持一类图计算作业,当用户实现这类作业时,补偿算法对其是透明的;而后者则是对为用户提供了编程范式,类似于一种异常处理框架,其内部具体的处理逻辑需要用户自己根据图计算作业的需要加以实现.下文结合具体的研究工作对这两种基于算法补偿的容错技术进行详细的分析和对比.

4.4.1 内置补偿算法容错

与前文的局部一致检查点容错机制所面临的情况一样,全局一致检查点阻塞进程保存快照,性能开销大,与分布式图计算框架为追求性能而采用的异步调度机制的初衷相悖;而且现有的日志容错技术,无论是 GraphX 基于 RDD 的 Lineage,还是 Pregel 的基于消息的日志容错技术,均不适合细粒度的异步计算过程,无法实现高效的受限恢复.文献[36,37]观察到,在 Maiter 框架上可实现一类能够改写成基于增量(delta-based)累加计算过程的分布式图计算作业[49],在这类作业中,图中每个顶点的状态被初始化为两部分:顶点值和增量值,顶点值在迭代的过程中根据入边传来的邻居顶点的增量值进行更新,而自身顶点值则沿着出边向外传播.这一过程具体为:顶点沿出边向邻居顶点发送增量值后将自身增量值重置为初始顶点值,之后接收入边方向传来的增量值并累加更新自身的增量值,接着用更新后的增量值和顶点当前值对顶点值进行更新,最后顶点再将增量值沿出边发送出去,异步地迭代以上过程,直至顶点值收敛至一个固定点,作业达到终止条件.文献[36,37]通过分析这类图计算作业,发现并证明:在迭代过程中,失效顶点的增量值已经沿着边传播在了整张图上,幸存的顶点上包含部分失效顶点的信息,所以,当失效顶点的状态丢失后,可用幸存顶点来恢复.具体是通过重新初始化失效顶点值、利用入边方向邻居顶点的值来计算得到失效顶点的增量值,最终自动地构建了一个失效顶点的特殊状态(弱有效状态),基于该状态继续计算,最终可以得到几乎完全准确的作业结果.最终,该方法在成本方面实现了零性能开销和零人力成本,在恢复效率方面,幸存节点不回滚不等待,恢复效率极为可观.

4.4.2 自定义补偿算法容错

虽然上述内置补偿算法的容错无需用户参与,但其所支持的作业范围相当狭窄,且其最终只能得到近似准确的结果,于是在文献[32]中,Phoenix 按照失效后图计算作业对恢复状态的需求将图计算作业分为 4 类:自稳定作业、局部可矫正作业、全局可矫正作业和全局一致作业.Dathathri 等人[32]对这 4 类作业的特点进行了详细的分析和归纳,并举例说明了前 3 类作业的容错思路,并为这 3 类作业分别提供了相应的失效恢复 API,以便于用户自己实现补偿算法进行容错.对于自稳定作业,API 以初始化函数作为输入并更新顶点状态;对于局部可矫正作业,API 以初始化函数作为输入,同时更新顶点状态和作业中的任务工作列表;对于全局校正算法,API 通过将初始化、重新初始化和重新计算的函数作为输入,并同时更新顶点状态和工作列表.虽然失效恢复过程是基于 BSP 同步调度机制的,其至少需要经历一轮计算和通信从而完成失效状态的修复,然而,作业本身采用同步或异步调度机制均可.最终,基于 Phoenix API 的用户应用程序级的容错机制在作业正常运行时无性能开销;而在后恢复运行期间,当失效发生的迭代越早、失效节点越少时,容错机制对作业运行有加速效果;反之,有减速效果.此外,用户在自定义实现补偿算法时,作业的类别也会影响用户需要付出的人力成本.当实现诸如 SSSP 之类的局部可矫正作业的容错逻辑时,仅需 30 行代码,而实现诸如 *k*-core 和 PageRank 等全局可矫正作业的容错逻辑时,需要 150~300 行代码,需要花费 1 人日左右的编程时间.此外,文献[61]也采用了类似的思路,其通过扩展数据流编程模型,增添了用户自定义补偿算法的方法接口,该方法只在失效发生时执行.然而,该方法仅支持自稳定作业和局部可矫正作业.

4.4.3 基于算法补偿的容错对比分析

表 4 列举了上述工作的优化对基于算法补偿的容错在成本、效率和质量这 3 个维度 6 个指标上的影响,并进行了对比.从表中可以看到,在这 6 个指标上,内置补偿算法容错和自定义补偿算法容错有 4 个都是一样的,这在一定程度上体现了基于算法补偿的容错的一些共性.首先关于零性能开销和快速恢复的特性就不再赘述了,第 3 个共性是后恢复运行时间的不确定,因为这段时间受失效发生的时机、失效节点的数量等因素共同决定,一般来说,当失效发生的时机越晚、失效节点的数量越多时,则失效丢失的状态越重要、越多,从而导致作业需要更多轮次的迭代来重新计算这些丢失的状态,反之亦然.第 4 个共性是都仅能处理多点失效,而不能处理级联失效,因为这些基于算法补偿的容错技术对失效状态的“修复”机制决定了当其遭遇级联失效时,不得不重启“修复”机制以保证“修复”的正确性,但是,由于其快速恢复的特性,“修复”机制通过重启将级联失效当作多点失效处理也不会耗费太多时间,于是,在实际使用中,基于算法补偿的容错能不能处理级联失效变得无关紧要.至此,内置补偿算法容错和自定义补偿算法容错仅有两个指标是不同的,而在这两者上的权衡和取舍,产生了前后这两

种不同的基于算法补偿的容错机制,前者更追求对用户透明,减少容错需要耗费的人力成本,而后者则更追求对难度更高的作业类型的容错支持.同时,后者出现的背景是近年来分布式图计算框架飞速发展,编程模型日益便利,用户实现分布式图计算作业愈发简单,在此基础上,用户应用程序级的容错机制已经相当简化了,但文献[32]显示,对于难度较高的作业类型,仍需要用户对图计算作业有深入的理解,需要花费时间编写冗长的容错代码.

Table 4 Comparison of fault tolerance based on algorithmic compensations

表 4 基于算法补偿的容错机制对比

名称	文献	成本		效率		质量	
		性能开销	用户参与度	失效恢复时间	后恢复运行时间	可支持的作业类型(≤)	可处理的失效类型(≤)
内置补偿算法容错	[36,37]	零性能开销	对用户透明	显著减少	不确定	近似作业	任意数量的多点失效
自定义补偿算法容错	[32]	零性能开销	需要用户深度参与	显著减少	不确定	全局可矫正作业	任意数量的多点失效
	[61]	零性能开销	需要用户深度参与	显著减少	不确定	局部可矫正作业	任意数量的多点失效

5 未来研究方向展望

作为分布式图计算研究领域中极其重要的一个子领域,面向分布式图计算作业的容错技术近年来得到了快速的发展,也取得了一定的成果和进展,然而该领域依然有许多亟待解决的问题.本节总结了 3 个未来值得研究的方向,分别是:分布式图计算作业中故障和失效特征分析、主被动容错协同的作业质量保障机制以及面向复杂图计算作业的容错技术.

5.1 分布式图计算作业中故障和失效特征分析

目前在面向分布式图计算作业的容错技术的研究领域内,绝大多数工作将分布式图计算作业面临的失效简单地假设为故障停止失效,并基于该假设做了相当多的容错机制优化工作.但是,随着图数据规模的日益庞大、图计算作业的运行环境日益复杂,分布式图计算作业中的故障和失效特征必然是极为复杂的,将分布式图计算作业面临的失效不加分辨地当作是故障停止失效进行处理必然是不合适的,如图 14 所示.

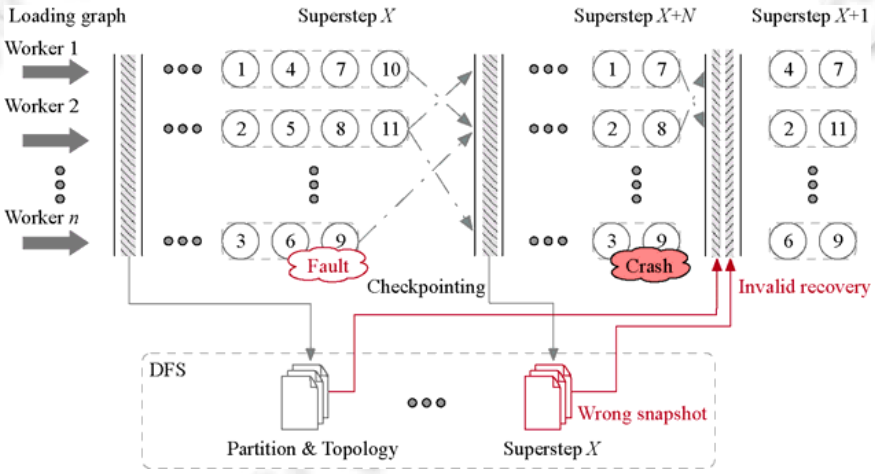

Fig.14 An example of non-fail-stop failure threat in global consistency checkpointing fault tolerance

图 14 非故障停止失效对全局一致检查点容错的威胁示例

在非故障停止失效(non-fail-stop failure)中,基于故障停止失效假设的全局一致检查点容错技术只有当计算节点在第 $X+N$ 超步崩溃时才会检测到失效,并开始加载最近的来自第 X 超步结束时保存的快照进行失效恢

复,然而故障是在第 X 超步运算过程中发生的,例如内存指针错误,这可能会导致容错机制在将作业恢复至第 $X+N$ 超步时节点再次崩溃或者使得作业运行得到错误的结果.然而仅有文献[60]一个工作涉及到了处理拜占庭失效,并且该工作并没有面向分布式图计算作业作进一步优化.此外,还有分布式图计算作业可能面临的性能失效、遗漏失效和崩溃失效则完全没有工作给予讨论,相关研究一片空白.之所以是这样的现状,归结原因在于缺乏对分布式图计算作业中故障和失效特征的深入认识.

研究分析分布式图计算作业中的故障和失效特征,不但能为相关研究提供目标指引,还能进一步打开面向分布式图计算作业的容错技术的优化空间,使其在充分耦合分布式图计算框架、分布式图计算作业的特点之后,也与故障和失效特征作进一步耦合,实现容错质量更好、恢复效率更高且容错成本更低的容错机制.

5.2 主被动容错协同的作业质量保障机制

现有的面向分布式图计算作业的容错技术全部采用被动容错机制,即是一类针对系统中已经发生的故障或失效进行处理、减少故障或失效带来的负面影响的机制.但被动容错的弊端在于失效已经发生,再优秀的被动容错技术也难以熨平失效给作业性能带来的颠簸,而颠簸会导致作业运行环境的不稳定,会给作业运行带来更多的不确定性,作业发生失效的概率亦随之增加,使得作业质量进入了一个负反馈循环中.而主动容错作为一类试图减少系统中故障发生概率,从而避免和阻止失效发生的机制,为减少和避免作业性能颠簸提供了可能,例如基于失效预测的预迁移技术[62,63],其中失效预测是该技术的核心,典型的失效预测方法通常需要人为地划分时间段作为预测窗口,然后基于日志(log)、监控(monitor)或追踪(track)数据进行预测[40].而在分布式图计算领域中,一方面,由于作业迭代计算的特性,天然地存在预测窗口(超步),为失效预测提供了便利;但另一方面,由于分布式图计算对性能的追求,可能缺乏充足的日志、监控和追踪数据,给准确地预测失效带来了挑战.

此外,被动容错仅能处理由于系统瞬时故障导致的失效,而主动容错则有机会对程序 bug 导致的永久故障进行处理,例如自修复(self-healing)和软件重生(rejuvenation)技术等[62],前者能够使系统自动感知自身是否处于错误状态,并在无人工干预的情况下进行必要的调整以恢复正常,避免失效.从原理上看,这种技术更适合具有较强鲁棒性的分布式图计算作业;而后者则是定期对软件进行重启,从而清理掉内部的状态,进而避免错误在系统内部的累积.从原理上来看,这种技术与图计算作业的迭代计算特性相得益彰.

然而,目前尚无研究将主动容错尝试应用于面向分布式图计算作业的容错机制之中.随着分布式图计算在各行各业的广泛应用,保障图计算作业质量的需求也会水涨船高,研究主被动容错协同的作业质量保证机制能够充分利用主动容错机制和被动容错机制的优势,在保障作业正常执行的情况下,尽量提高作业运行质量.

5.3 面向复杂图计算作业的容错技术

目前,面向分布式图计算作业的容错技术相关研究主要集中在对基于静态图的分析作业的容错.基于静态图的分析作业具有计算会遍历访问整张图和多轮迭代计算直至收敛的特点,典型的有 PageRank、图着色(graph coloring)、图模式挖掘(graph pattern mining)和 K-core 等作业.然而,图计算作业正在变得愈发复杂,原因主要来自于两方面:(1) 动态图、流式图变得愈发普遍,图数据变得更为复杂,导致作业的计算模式发生改变[64−68].其中,动态图相比于静态图而言,图中的属性值或拓扑结构会随着时间发生变化,而流式图则在图上进一步引入了流的特性,一是流式图是无界的,作业无法访问到整个图.另一方面,流式图的传输速率可能会非常高,导致图更新的速度会非常快.目前,基于动态图和流失图的作业计算模式主要有基于快照的批量计算(batch computation of each snapshot)和增量计算(incremental computation),前者对不同时刻的图进行快照,然后在每个快照上进行分析;而后者则是将图的变动建模进计算过程中,需要对计算模型进行特殊设计;(2) 算法自身变得更为复杂,例如顶点/图分类、链接预测等作业.这些与深度学习相结合的图计算作业——图神经网络[69−71],主要包括图卷积网络、图循环神经网络和图注意力网络,其基本的计算模式是通过图结构进行信息传播,利用神经网络来进行顶点更新,并且在模型训练阶段,需要批量执行和利用反向传播更新神经网络参数.相比传统的基于静态图的分析作业,图神经网络无论是在编程实现还是在任务调度方面均有很大的不同.近年来,各大开源图计算项目也在陆

续支持这些复杂图计算作业的实现,工业界和学术界还出现了一些新型的分布式图计算系统——图神经网络系统,如微软的 Neugraph[72]和阿里巴巴的 AliGraph[73],专门用于大规模图神经网络的训练和推理,这进一步验证了图计算作业愈发复杂化的发展趋势,也为作业容错提出了新的需求,面向复杂图计算作业的容错技术相关研究正日益迫切和重要.

References:

[1] Sahu S, Mhedhbi A, Salihoglu S, Lin J, Özsu MT. The ubiquity of large graphs and surprising challenges of graph processing. Proc. of the VLDB Endowment, 2017,11(4):420–431. [doi: 10.1145/3164135.3164139]

[2] Sahu S, Mhedhbi A, Salihoglu S, Lin J, Özsu MT. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. The VLDB Journal, 2019,29(2-3):595–618. [doi: 10.1007/s00778-019-00548-x]

[3] Salihoglu S, Ozsu MT. Response to “scale up or scale out for graph processing”. IEEE Internet Computing, 2018,22(5):18–24.

[4] Grzegorz MG, Austern MH, Bik AJC, Dehnert JC, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In: Proc. of the 2010 ACM SIGMOD Int’l Conf. on Management of Data. New York: Association for Computing Machinery, 2010. 135–146. [doi: 10.1145/1807167.1807184]

[5] Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM. GraphLab: A new framework for parallel machine learning. In: Proc. of the 26th Conf. on Uncertainty in Artificial Intelligence. Catalina Island, 2010. 340–349.

[6] Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, 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] Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In: Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2012. 17–30.

[8] Giraph. http://giraph.apache.org

[9] Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: Graph processing in a distributed dataflow framework. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2014. 599–613.

[10] Gelly. http://flink.iteblog.com/dev/libs/gelly

[11] Egwutuoha IP, Levy D, Selic B, Chen S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. The Journal of Supercomputing, 2013,65(3):1302–1326. [doi: 10.1007/s11227-013-0884-0]

[12] Novaković D, Vasić N, Novaković S, Kostić D, Bianchini R. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In: Proc. of the 2013 USENIX Conf. on Annual Technical Conf. USENIX Association, 2013. 219–230.

[13] Wang KJ, Jia T, Li Y. State-of-the-art survey of scheduling and resource management technology for colocation jobs. Ruan Jian Xue Bao/Journal of Software, 2020,31(10):3100–3119 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6066.htm [doi: 10.13328/j.cnki.jos.006066]

[14] Fried J, Ruan Z, Ousterhout A, Belay A. Caladan: Mitigating interference at microsecond timescale. In: Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation. USENIX Association, 2020. 281–297.

[15] Bartlett J, Gray J, Horst B. Fault tolerance in Tandem computer systems. In: The Evolution of Fault-tolerant Computing. 1987, 55–76.

[16] Bartlett W, Spainhower L. Commercial fault tolerance: A tale of two systems. IEEE Trans. on Dependable and Secure Computing, 2004,1(1):87–96. [doi: 10.1109/TDSC.2004.4]

[17] Borg A, Blau W, Graetsch W, Herrmann F, Oberle W. Fault tolerance under UNIX. ACM Trans. on Computer Systems, 1989, 7(1):1–24. [doi: 10.1145/58564.58565]

[18] Zhong H, Nieh J. CRAK: Linux checkpoint/restart as a kernel module. 2001. http://systems.cs.columbia.edu/files/wpid-cucs-014-01.pdf

[19] Agarwal S, Garg R, Gupta MS, Moreira JE. Adaptive incremental checkpointing for massively parallel systems. In: Proc. of the 18th Annual Int’l Conf. on Supercomputing. New York: Association for Computing Machinery, 2004. 277–286. [doi: 10.1145/1006209.1006248]

[20] Hargrove PH, Duell JC. Berkeley laboratory checkpoint/restart (BLCR) for Linux clusters. Journal of Physics (Conf. Series), 2006, 46(1):494–9. [doi: 10.1088/1742-6596/46/1/067]

[21] Plank JS, Kai L. ICKP: A consistent checkpointer for multicomputers. IEEE Parallel & Distributed Technology: Systems & Applications, 1994,2(2):62–67. [doi: 10.1109/88.311574]

[22] Plank JS, Kai L, Puening MA. Diskless checkpointing. IEEE Trans. on Parallel and Distributed Systems, 1998,9(10):972–986. [doi: 10.1109/71.730527]

[23] Sankaran S, Squyres JM, Barrett B, Sahay V, Lumsdaine A, Duell J, Hargrove P, Roman E. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. The Int'l Journal of High Performance Computing Applications, 2005,19(4):479–493. [doi: 10.1177/1094342005056139]

[24] Zheng G, Ni X, Kalé LV. A scalable double in-memory checkpoint and restart scheme towards exascale. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks Workshops (DSN 2012). Boston, 2012. 1–6. [doi: 10.1109/DSNW.2012.6264677]

[25] Heidari S, Simmhan Y, Calheiros RN, Buyya R. Scalable graph processing frameworks: A taxonomy and open challenges. ACM Computing Surveys (CSUR), 2018,51(3):1–53.

[26] Mccune RR, Weninger T, Madey G. Thinking like a Vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Computing Surveys (CSUR), 2015,48(2):1–39.

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

[28] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the Presented as Part of the 9th USENIX Symp. on Networked Systems Design and Implementation (NSDI 12). 2012. 15–28.

[29] Stutz P, Bernstein A, Cohen W. Signal/collect: Graph algorithms for the (semantic) Web. In: Patel-Schneider PF, *et al*. eds. Proc. of the Int'l Semantic Web Conf. (ISWC). Berlin: Springer-Verlag, 2010. 764–780.

[30] Bronevetsky G, Marques D, Pingali K, Stodghill P. Automated application-level checkpointing of MPI programs. In: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: Association for Computing Machinery, 2003. 84–94. [doi :10.1145/781498.781513]

[31] Beguelin A, Seligman E, Stephan P. Application level fault tolerance in heterogeneous networks of workstations. Journal of Parallel and Distributed Computing, 1997,43(2):147–155.

[32] Dathathri R, Gill G, Hoang L, Pingali K. Phoenix: A substrate for resilient distributed graph analytics. In: Proc. of the 24th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: Association for Computing Machinery, 2019. 615–630. [doi: 10.1145/3297858.3304056]

[33] Hoang L, Pontecorvi M, Dathathri R, Gill G, You B, Pingali K, Ramachandran V. A round-efficient distributed betweenness centrality algorithm. In: Proc. of the 24th Symp. on Principles and Practice of Parallel Programming (PPoPP 2019). New York: Association for Computing Machinery, 2019. 272–286.

[34] Iyer AP, Liu Z, Jin X, Venkataraman S, Braverman V, Stoica I. ASAP: Fast, approximate graph pattern mining at scale. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 745–761.

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

[36] Wang Z, Gao L, Gu Y, Bao Y, Yu G. A fault-tolerant framework for asynchronous iterative computations in cloud environments. In: Proc. of the 7th ACM Symp. on Cloud Computing. New York: Association for Computing Machinery, 2016. 71–83.

[37] Wang Z, Gao L, Gu Y, Bao Y, Yu G. A fault-tolerant framework for asynchronous iterative computations in cloud environments. IEEE Trans. on Parallel and Distributed Systems, 2018,29(8):1678–1692.

[38] Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. on Dependable and Secure Computing, 2004,1(1):11–33.

[39] Poola D, Salehi MA, Ramamohanarao K, Buyya R. Chapter 15—A Taxonomy and Survey of Fault-tolerant Workflow Management Systems in Cloud and Distributed Computing Environments. Elsevier Inc., 2017. 285–320.

[40] Salfner F, Lenk M, Malek M. A survey of online failure prediction methods. ACM Computing Surveys, 2010,42(3):1–42.

[41] Wang Z, Gu Y, Bao Y, Yu G, Gao L. An I/O-efficient and adaptive fault-tolerant framework for distributed graph computations. Distributed and Parallel Databases, 2017,35(2):177–196.

[42] Jhawar R, Piuri V, Santambrogio M. A comprehensive conceptual system-level approach to fault tolerance in cloud computing. In: Proc. of the 2012 IEEE Int'l Systems Conf. (SysCon 2012). Vancouver, 2012. 1–5.

[43] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems. New York: Association for Computing Machinery, 2007. 59–72.

[44] Power R, Li J. Piccolo: Building fast, distributed programs with partitioned tables. In: Proc. of the 9th USENIX Conf. on Operating Systems Design and Implementation. 2010. 293–306.

[45] Carbone P, Fóra G, Ewen S, Haridi S, Tzoumas K. Lightweight asynchronous snapshots for distributed dataflows. arXiv Preprint arXiv: 1506.08603, 2015.

[46] Garg R, Kumar P. A review of checkpointing fault tolerance techniques in distributed mobile systems. Int'l Journal on Computer Science and Engineering, 2010,2(4):1052–1063.

[47] Bi YH, Jiang SY, Wang ZG, Leng FL, Bao YB, Yu G, Qian L. A multi-level fault tolerance mechanism for disk-resident Pregel-like systems. Journal of Computer Research and Development, 2016,53(11):2530–2541

[48] Yan D, Cheng J, Chen H, Long C, Bangalore P. Lightweight fault tolerance in Pregel-like systems. In: Proc. of the 48th Int'l Conf. on Parallel Processing. New York: Association for Computing Machinery, 2019. 1–10

[49] Xue J, Yang Z, Qu Z, Hou S, Dai Y. Seraph: An efficient, low-cost system for concurrent graph processing. In: Proc. of the 23rd Int'l Symp. on High-performance Parallel and Distributed Computing. 2014. 227–238.

[50] Xu C, Holzemer M, Kaul M, Soto J, Markl V. On fault tolerance for distributed iterative dataflow processing. IEEE Trans. on Knowledge and Data Engineering, 2017,29(8):1709–1722.

[51] Xu C, Holzemer M, Kaul M, Markl V. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In: Proc. of the 32nd IEEE Int'l Conf. on Data Engineering (ICDE). 2016. 613–624.

[52] Vora K, Tian C, Gupta R, Hu Z. CoRAL: Confined recovery in distributed asynchronous graph processing. In: Proc. of the 32nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. 2017. 223–236.

[53] Elnozahy EN, Alvisi L, Wang YM, Johnson D. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys, 2002,34(3):375–408.

[54] Lu W, Shen Y, Wang T, Zhang M, Jagadish HV, Du X. Fast failure recovery in vertex-centric distributed graph processing systems. IEEE Trans. on Knowledge and Data Engineering, 2019,31(4):733–746.

[55] Shen Y, Chen G, Jagadish HV, Lu W, Ooi BC, Tudor BM. Fast failure recovery in distributed graph processing systems. Proc. of the VLDB Endowment, 2014,8(4):437–448.

[56] Kaur J, Kinger S. Analysis of different techniques used for fault tolerance. Int'l Journal of Computer Science and Information Technologies, 2014,5(3):4086–4090.

[57] Pundir M, Leslie LM, Gupta I, Campbell RH. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In: Proc. of the 6th ACM Symp. on Cloud Computing. New York: Association for Computing Machinery, 2015. 195–208.

[58] Wang P, Zhang K, Chen R, Chen H, Guan H. Replication-based fault-tolerance for large-scale graph processing. In: Proc. of the 44th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. 2014. 562–573. [doi: 10.1109/DSN.2014.58]

[59] Chen R, Yao Y, Wang P, Zhang K, Guan H, Zang B, Chen H. Replication-based fault-tolerance for large-scale graph processing. IEEE Trans. on Parallel and Distributed Systems, 2018,29(7):1621–1635.

[60] Presser D, Lung LC, Correia M. Greft: Arbitrary fault-tolerant distributed graph processing. In: Proc. of the 2015 IEEE Int'l Congress on Big Data. New York, 2015. 452–459.

[61] Schelter S, Ewen S, Tzoumas K, Markl V. "All roads lead to Rome": Optimistic recovery for distributed iterative data processing. In: Proc. of the 22nd ACM Int'l Conf. on Information & Knowledge Management. 2013. 1919–1928.

[62] Marcotte P, Gregoire F, Petrillo F. Multiple fault-tolerance mechanisms in cloud systems: A systematic review. In: Proc. of the 2019 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). Berlin, 2019. 414–421.

[63] Gan Y, Zhang Y, Hu K, Cheng D, He Y, Pancholi M, Delimitrou C. SEER: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In: Proc. of the 24th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. 2019. 19–33.

[64] Mariappan M, Vora K. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In: Proc. of the 14th EuroSys Conf. 2019. 1–16.

[65] Sheng F, Cao Q, Cai H, Yao J, Xie C. GraPU: Accelerate streaming graph analysis through preprocessing buffered updates. In: Proc. of the ACM Symp. on Cloud Computing. 2018. 301–312.

[66] Feng G, Ma Z, Li D, Zhu X, Cai Y, Han W, Chen W. RisGraph: A real-time streaming system for evolving graphs. arXiv Preprint arXiv: 2004.00803, 2020.

[67] Mcsherry F, Murray DG, Isaacs R, Isard M. Differential dataflow. In: Proc. of the 6th Biennial Conf. on Innovative Data Systems Research (CIDR 2013). 2013.

[68] Ammar K, McSherry F, Salihoglu S, Joglekar M. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. arXiv Preprint arXiv: 1802.03760, 2018.

[69] Peng N, Poon H, Quirk C, Toutanova K, Yih W. Cross-sentence *n*-ary relation extraction with graph LSTMS. Trans. of the Association for Computational Linguistics, 2017,5:101–115.

[70] Veličković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. arXiv Preprint arXiv: 1710.10903, 2017.

[71] Bresson X, Laurent T. Residual gated graph ConvNets. arXiv Preprint arXiv: 1711.07553, 2017.

[72] Ma L, Yang Z, Miao Y, Xue J, Wu M, Zhou L, Dai Y. Neugraph: Parallel deep neural network computation on large graphs. In: Proc. of the 2019 USENIX Annual Technical Conf. (USENIX ATC 19). 2019. 443–458.

[73] Zhu R, Zhao K, Yang H, Lin W, Zhou C, Ai B, Li Y, Zhou J. AliGraph: A comprehensive graph neural network platform. Proc. of the VLDB Endowment, 2019,12(12):2094–2105.

附中文参考文献:

[13] 王康瑾,贾统,李影.在离线混部作业调度与资源管理技术研究综述.软件学报,2020,31(10):3100–3119. http://www.jos.org.cn/1000-9825/6066.htm [doi: 10.13328/j.cnki.jos.006066]

[47] 毕亚辉,姜苏洋,王志刚,冷芳玲,鲍玉斌,于戈,钱岭.面向磁盘驻留的类 Pregel 系统的多级容错处理机制.计算机研究与发展,2016,53(11):2530–2541.

张程博(1994－),男,博士生,主要研究领域为分布式图计算系统,容错.

贾统(1993－),男,博士后,主要研究领域为分布式系统,智能运维.

李影(1975－),女,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式计算,可信计算.