·体系结构与软件技术· 文章编号:1000-3428(2021)04-0173-07 文献标志码:A 中图分类号:TP338.8

MapReduce框架下结合分布式编码计算的容错算法

张 基,谢在鹏,毛莺池,徐媛媛,朱晓瑞,李博文
(河海大学 计算机与信息学院,南京 211100)

摘 要: 随着分布式系统规模扩大及计算复杂度增加,分布式计算的平均故障修复时间和容错计算所产生的通信开销呈现日益上升趋势。结合分布式编码计算和副本冗余技术,提出一种新的容错算法。map节点应用分布式编码计算的思想,将数据冗余分配至多个计算节点创建编码中间结果,降低计算节点在shuffle阶段的数据传输量。reduce节点通过对接收到的编码中间结果进行解码,从而验证中间结果的正确性并得到最终计算结果。实验结果表明,在基于MapReduce的分布式计算框架下,与三模冗余和两阶段三模冗余容错算法相比,该算法在完成容错计算的同时能降低计算过程中的通信开销和平均故障修复时间,并提高分布式系统的可用性和可靠性。

关键词: 分布式系统;分布式计算;容错算法;分布式编码计算;三模冗余

开放科学(资源服务)标志码(OSID):

中文引用格式: 张基,谢在鹏,毛莺池,等.MapReduce框架下结合分布式编码计算的容错算法[J].计算机工程,2021,47(4):173-179.

英文引用格式: ZHANG Ji, XIE Zaipeng, MAO Yingchi, et al. Fault-tolerant algorithm combined with distributed coding computing in MapReduce framework[J].Computer Engineering,2021,47(4):173-179.

Fault-Tolerant Algorithm Combined with Distributed Coding Computing in MapReduce Framework

ZHANG Ji,XIE Zaipeng,MAO Yingchi,XU Yuanyuan,ZHU Xiaorui,LI Bowen
(School of Computer and Information,Hohai University,Nanjing 211100,China)

【Abstract】 The growing size and computational complexity of distributed systems lead to an increase in the Mean Time to Repair(MTTR) of distributed computing systems and the communication load caused by fault-tolerant computing.To solve the problems,this paper integrates distributed coding computing with replica redundancy to propose a novel fault-tolerant algorithm. The map node uses the idea of distributed coding computing to allocate data replica to multiple computing nodes to create intermediate coding results and reduce the amount of data transmitted by the computing nodes in the shuffle phase.The reduce node decodes the received intermediate coding result to verify its correctness and obtain the final computing result. Experimental results show that in the MapReduce framework, the proposed algorithm can reduce the communication overhead and MTTR compared with the Triple Modular Redundancy(TMR) and two-stage TMR fault-tolerant algorithms.It also improves the availability and reliability of distributed systems.

【Key words】 distributed system; distributed computing; fault-tolerant algorithm; distributed coding computing; Triple Modular Redundancy(TMR)

DOI: 10. 19678/j. issn. 1000-3428. 0057721

0 概述

容错技术是分布式系统的重要组成部分,确保了发生故障时的系统连续性和功能性[1]。近年来,随着分布式系统规模的不断扩大、分布式架构和计算复杂度的日益增加[2]以及廉价商业硬件的广泛使用,使得计算任务发生故障的概率持续增加,例如在Google生产集群中平均每天会有数十个节点发生故障[3]。瞬态故障尤其是软错误[4]会导致计算机系统的异常行为,这类故障会损坏数据的完整性,引起分布式节点的计算失效[4-5]。在一些大型分布式系统中平均每天会有1%~2%的节点发生失效[6],因此容

基金项目: 国家自然科学基金重点项目(61832005);国家重点研发计划(2016YFC0402710)。

作者简介: 张 基(1997—),男,硕士研究生,主研方向为分布式计算;谢在鹏,副教授、博士;毛莺池,教授、博士;徐媛媛、朱晓瑞,博士;李博文,硕士研究生。

收稿日期: 2020-03-13 **修回日期:** 2020-04-28 **E-mail:** jizhnag@hhu.edu.cn

错技术可在保证部分节点失效的情况下，使分布式系统仍能继续运行且得到正确结果[7-9]。在基于MapReduce[10]的分布式计算中，数据洗牌（shuffle）阶段较高的通信开销严重影响了分布式计算性能，例如在Facebook的Hadoop集群中，33%的任务执行总时间用于数据洗牌阶段[11]。针对基于MapReduce分布式计算框架的多副本容错算法通信开销较大的问题，本文提出一种结合分布式编码计算的容错算法CTMR，使基于MapReduce的分布式计算系统在发生瞬态故障的情况下仍能继续运行且得到正确结果，同时能有效降低容错计算过程中的通信开销。

1 相关工作

基于多副本冗余技术[12-15]的分布式容错算法于1962年由IBM提出，但在现代分布式系统中仍然被广泛采用[16-17]。这类算法的主要思想是：在含有n个节点的分布式系统中，如果每个节点要容忍f个故障，那么该节点可以使用（f+1）个独立的副本，显然存储和运行这些副本需要消耗大量的空间和其他资源。此外，由于多副本之间的一致性使得基于副本冗余技术的容错算法更易于设计，同时副本的维护和恢复成本较低。三模冗余（Triple Modular Redundancy，TMR）是软件和硬件系统中常用的基于副本的容错算法[18-19]，使用3个实现功能相同的模块同时进行操作，系统使用投票机制选出最终输出。由于3个模块相互独立，同时有2个模块出现相同错误的概率非常小，因此可大幅提高系统可靠性，掩蔽故障模块的错误。文献[20]提出一种优化的副本虚拟机放置算法，动态设置k个副本虚拟机并将其作为备份来提高云服务的可靠性。在基于MapReduce的分布式计算[10]中，每个任务被分配到多个节点以确保在某个节点出现故障的情况下系统仍能继续运行。文献[21]提出面向分布式流体系结构的多副本容错技术，该技术通过比较多个副本的数据进行检错，采取三取二的逻辑判断方式选择多数相同的结果，并对错误的数据进行纠错以防止错误进一步传播，但当数据量较大时通信开销会成为容错过程中的性能瓶颈。文献[18]提出一种两阶段三模冗余（two-stage TMR）容错算法，该算法将每个任务备份3份并分配给3个节点，先指定2个节点进行计算，当2个节点执行完毕后进行结果比较，如果比较结果不一致，则指定第3个节点进行计算，最终通过投票选出多数一致的结果。该算法在故障率较低的情况下可以有效节省系统资源，但当错误率较高时，重新执行第3份备份任务不仅增加了计算工作量并且大幅降低了系统实时性能。

通信开销是基于MapReduce的分布式计算中的主要性能瓶颈，这是因为在数据洗牌阶段交换大量中间结果。为解决该问题，文献[7]提出分布式编码计算方法，将map任务重复布置到多个不同的节点，通过增加计算冗余创建同时满足多个服务器数据需求的编码数据来降低通信开销。在一个分布式集群中，node_1包含数据集$\{v_1,v_2\}$，node_2包含数据集$\{v_2,v_3\}$，node_3包含数据集$\{v_1,v_3\}$。node_1将本地数据集编码结果c_1（$c_1=v_1\oplus v_2$，$\oplus$表示异或）运用广播发出。node_2接收到编码数据后利用本地数据v_2即可将收到的编码结果解码得到v_1，其中$v_1=v_2\oplus c_1$。同理，node_3通过解码即可得到v_2。在该过程中，通过将node_1与node_2和node_3冗余存储的数据$\{v_1,v_2\}$进行编码来创建同时满足node_2和node_3的编码数据，而无需将v_1、v_2分别发送至node_2和node_3，从而降低通信开销。然而，目前尚未发现能有效降低分布式容错计算中通信开销的相关讨论和研究。为降低现有分布式计算使用多副本容错过程中产生的通信开销，本文基于副本冗余和分布式编码计算技术，提出一种CTMR容错算法。

2 CTMR容错算法

2.1 CTMR模型

假设在分布式集群中有N个节点，标记为$\{\text{node}_1,\text{node}_2,\cdots,\text{node}_n\}$，$N$个节点协作完成一个计算任务。当前集群待处理的数据量为M，每份数据的冗余度为r，将该数据集分配到当前集群的N个map节点，每个节点的数据量为$M\cdot r/N$。map节点对数据进行计算，将产生的中间结果使用分布式编码计算得到的编码数据发送到Q个reduce节点，其中编码压缩比为c，即将c个中间结果通过编码得到一个编码中间结果。因此，每个reduce节点将收到其余（$N-Q$）个map节点的编码中间结果，如式（1）所示：

$$\frac{M\cdot r}{N}\cdot(r-1)\cdot\frac{1}{c}=N-Q \tag{1}$$

reduce节点对接收到的编码中间结果进行解码，通过校验识别发生故障的编码数据包，并利用冗余中间结果得到M份数据最终正确的计算结果，如式（2）所示：

$$Q\cdot\frac{M\cdot r}{N}=M \tag{2}$$

因为本文算法随机选取Q个reduce节点均能够完成所有中间结果的验证，所以节点应该包含编码数据所有可能的中间结果，如式（3）所示：

$$\begin{pmatrix}M/c\\ \dfrac{M\cdot r}{N}\cdot\dfrac{1}{c}\end{pmatrix}=N \tag{3}$$

由于TMR的低复杂度及高可靠性，因此本文选取$r=3$、$c=2$，联立式(1)~式(3)解得$N=6$、$M=8$，即在含有6个节点的分布式集群中，将待处理的数据分成8个独立的子数据块，每个数据块冗余3份，每个节点包含4份数据，使用故障检测与恢复算法进行容错计算，同时降低计算过程中的通信开销。

在包含N个节点的分布式集群中，将每6个节点分为1个子集群，共有$\lfloor N/6 \rfloor$组子集群。各个子集群之间相互独立，而每个子集群内的节点在逻辑上是相邻的，在物理上可以是分散的。将计算任务分配到$\lfloor N/6 \rfloor$个子集群上，各个子集群并行计算。每个子集群的数据量为$B=M/\lfloor N/6 \rfloor$，同时将子集群的数据集划分为8个独立的子数据块，每个数据块复制3份。每个子集群使用如图1所示的CTMR模型进行表示，其中立方体的每个面表示1个节点，每个面的4个顶点代表当前节点包含的4个数据集。立方体中的每条棱代表与该棱邻接的两个面所表示的节点公共冗余数据集，而每个面所表示的节点数据集用B_i表示，例如node_1本地数据集为$B_1=\{b_1, b_2, b_3, b_4\}$。在模型中每个顶点与3个面邻接，即每份数据冗余3份并存放至3个节点上，例如b_3分别存放至node_1、node_3和node_5上。node_k在map阶段对本地数据集的每一个数据块b_i执行map函数$F(b_i)$得到相应的中间结果集$V_k=\{v_i^k, b_i \in B_k\}$。

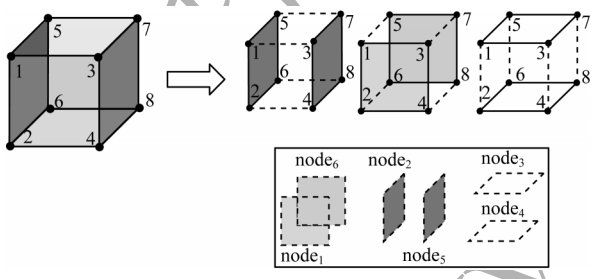

图1　CTMR模型

Fig.1　CTMR model

两个邻近节点node_k和node_m冗余存储的数据集用$R_{k,m}$表示，例如$R_{1,3}=\{b_1, b_3\}$。将node_k与node_m冗余存储的数据$R_{k,m}$所对应的中间结果使用分布式编码计算得到编码结果$u_{k,m}$，如式(4)所示：

$$u_{k,m}=\oplus b_i, b_i \in R_{k,m} \tag{4}$$

其中，$\oplus$表示异或操作。node_k在reduce阶段的函数为$H(v_1, v_2, \cdots, v_i)=r_k$。

2.2　CTMR故障检测与恢复

将一个包含N个节点的分布式集群划分为$\lfloor N/6 \rfloor$个子集群，各个子集群并行计算的同时进行检错和纠错。每个子集群中的6个节点分别表示为node_k、node_m、node_s、node_n、node_p、node_q，其中node_k和node_m、node_s和node_n、node_p和node_q分别为CTMR模型中的3组对面。可以看出，任意一组对面中的两个节点的本地数据集的交集为空，但并集是当前子集群数据集的全集。每个节点map阶段通过函数$F(b_i)$计算本地数据集得到相应的中间结果集，例如node_k通过计算本地数据集B_k得到相应的中间结果集$V_k=\{v_i^k, b_i \in B_k\}$。

随机选取一个节点node_k，将其在CTMR模型中的对面节点node_m作为校验节点。node_s、node_n、node_p、node_q为与node_k相邻的节点，将其与node_k冗余存储的数据集$R_{s,k}$、$R_{n,k}$、$R_{p,k}$、$R_{q,k}$所对应的中间结果进行编码，得到相应的编码结果$u_{s,k}$、$u_{n,k}$、$u_{p,k}$、$u_{q,k}$。将对应的编码结果发送给node_k，然后在node_m上执行相同过程。数据块分发与中间结果编码的伪代码如算法1所示。

算法1　数据分发与中间结果编码算法

输入　Dataset B

输出　各个节点的编码中间结果

```
1.for each node_i, i ≤ 6 do
2.distribute B_i to node_i//将数据分发给各个节点
3.V_i = F(B_i)//计算每个节点数据集的中间结果
4.end for
5.select two check nodes node_k, node_m//选取校验节点
6.for each node_i do
7.if i ≠ k && i ≠ m
8.u_{i,k} = ⊕R_{i,k}//对中间结果进行编码
9.u_{i,m} = ⊕R_{i,m}//对中间结果进行编码
10.send u_{i,k} to node_k, send u_{i,m} to node_m//将编码中间结果
//发送给校验节点
11.end for
```

node_k和node_m通过比较接收到的编码数据包与当前节点产生的中间结果来验证数据的正确性。若式(5)两个等式中的任意一个成立，则表明node_k本地数据集的运算结果正确，通过reduce函数可得到当前节点运算结果$r_k=H(v_1^k, v_2^k, \cdots, v_i^k)$。

$$\begin{cases}(u_{s,k}=u_{k,s})\wedge(u_{n,k}=u_{k,n})\\(u_{p,k}=u_{k,p})\wedge(u_{q,k}=u_{k,q})\end{cases} \tag{5}$$

若式(5)中两个等式均不成立，而式(6)两个等式中的任意一个成立，则假设$u_{s,k}$验证成功，即$u_{s,k}=u_{k,s}$，$u_{k,s}=v_i^k \oplus v_j^k$，$R_{s,k}=\{b_i, b_j\}$。

$$\begin{cases}(u_{s,k}=u_{k,s})\vee(u_{n,k}=u_{k,n})\\(u_{p,k}=u_{k,p})\vee(u_{q,k}=u_{k,q})\end{cases} \tag{6}$$

若式(7)成立,则表明$node_k$本地有数据计算错误,但是收到的其他节点的编码数据包正确。此时,通过reduce函数得到当前节点数据集的正确结果为$r_k=H(v_i^k,u_{q,k}\oplus v_i^k,v_j^k,u_{p,k}\oplus v_j^k)$。

$$u_{q,k}\oplus v_i^k\oplus u_{p,k}\oplus v_j^k=u_{k,s} \tag{7}$$

若$u_{s,k}$、$u_{n,k}$、$u_{p,k}$、$u_{q,k}$全部验证失败,则重新选取两个校验节点进行上述操作。当$node_k$与$node_m$均能通过验证得到正确运算结果时,利用reduce函数便可得到CTMR模型的最终结果$r=H(r_k,r_m)$。故障检测与恢复的伪代码如算法2所示。

算法2 故障检测与恢复算法

输入 各个节点的编码中间结果U,$U=\{u_{s,k},u_{n,k},u_{p,k},u_{q,k},u_{s,m},u_{n,m},u_{p,m},u_{q,m}\}$

输出 子集群的reduce结果

```
1.for node_i in { node_k, node_m }//利用校验节点对收到的编
//码中间结果做校验
2.if i = k do
3.if one of the formula (5) is true do//如果校验节点所在
//面有一组对边校验成功
4.r_k = H( v_1^k, v_2^k, ⋯, v_i^k )//当前节点的reduce结果
5.else if one of the formula (6) is true do//如果校验节点
//有一条边验证成功
6.assume u_{s,k} = u_{k,s}
7.if formula (7) is true do//如果通过当前校验成功的边
//验证收到的其他编码中间结果正确,则证明当前节点有中间
//结果计算错误
8.r_k = H( v_i^k, u_{q,k}⊕v_i^k, v_j^k, u_{p,k}⊕v_j^k )//通过收到的编码中
//间结果得到当前节点的reduce结果
9.else goto step 6//重新选取校验节点进行故障检测与
//恢复
10.if node_m compute the result r_m
11.r = H( r_k, r_m )//当前子集群的reduce结果
12.return r
```

在随机选取子集群中的两个reduce节点后,每次TMR算法验证都需发送16份中间结果给reduce节点,假设每个中间结果大小为τ,那么每个CTMR模型在验证过程中需要发送16τ的数据。在含有N个节点的分布式集群中,共需发送$L_{\mathrm{TMR}}=16\tau\times\lfloor N/6\rfloor$的数据量。two-stage TMR算法在最优情况下只需发送8τ的数据量,即最初选择的两个副本对应的中间结果一致,在最坏情况下所有数据最初选择的两个副本的对应中间结果均不一致,因此需要第3个副本进行多数一致表决,共需发送16τ的数据量。在CTMR算法中,每个节点使用分布式预编码本地计算的中间结果减少通信开销。在最优情况下,每个模型只需8τ的通信量,即模型最初选择两个校验点即可得到正确结果。在最坏情况下,每个模型需要16τ的通信量,即在最初选择两个校验节点后不能得到正确结果,需更换节点重新做校验。因此,在包含N个节点的分布式集群中,共需发送$L_{\mathrm{CTMR}}\left(8\tau\times\lfloor N/6\rfloor\leqslant L_{\mathrm{CTMR}}\leqslant 16\tau\times\lfloor N/6\rfloor\right)$的数据量。CTMR算法与TMR算法的通信开销之比如式(8)所示,即CTMR算法总能在小于等于TMR算法通信量的情况下得到正确结果。

$$\frac{1}{2}\leqslant\frac{L_{\mathrm{CTMR}}}{L_{\mathrm{TMR}}}\leqslant 1 \tag{8}$$

2.3 算法实例

如图2所示,在含有$N=6$个节点的分布式集群中,输入系统数据量为$M=8$,将该数据集划分为8个数据块$\{b_1,b_2,\cdots,b_8\}$。每个节点在map阶段计算本地数据集产生相应的中间结果集,例如$node_2$在map阶段针对本地数据集$B_2=\{b_1,b_2,b_5,b_6\}$计算得到相应的中间结果集$V_2=\{v_1^2,v_2^2,v_5^2,v_6^2\}$。考虑$node_1$在map阶段由于瞬态故障导致$v_1^1$和$v_2^1$计算错误情况下的容错过程,首先选取$node_1$和$node_6$作为校验节点,子集群中其余4个节点为$node_2$、$node_3$、$node_4$、$node_5$,将4个节点与$node_1$冗余存储的数据集$R_{2,1}$、$R_{3,1}$、$R_{4,1}$、$R_{5,1}$所对应的中间结果分别使用分布式编码计算得到相应的编码结果$u_{2,1}$、$u_{3,1}$、$u_{4,1}$、$u_{5,1}$发送给$node_1$,将4个节点与$node_6$冗余存储的数据集$R_{2,6}$、$R_{3,6}$、$R_{4,6}$、$R_{5,6}$所对应的中间结果分别使用分布式编码计算得到相应的编码结果$u_{2,6}$、$u_{3,6}$、$u_{4,6}$、$u_{5,6}$发送给$node_6$,其中,$u_{2,1}=v_1^2\oplus v_2^2$,$u_{3,1}=v_1^3\oplus v_3^3$,$u_{4,1}=v_2^4\oplus v_4^4$,$u_{5,1}=v_3^5\oplus v_4^5$。

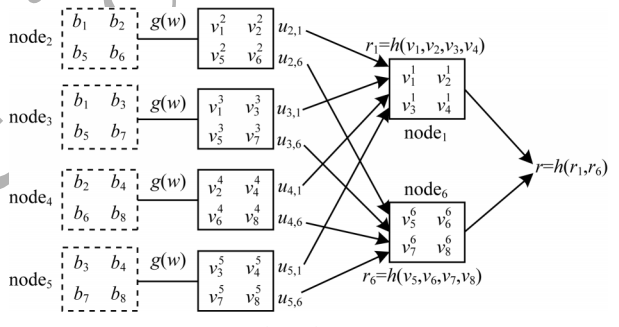

图2 $node_1$中v_1^1和v_2^1错误时的容错过程

Fig.2 Fault-tolerant process of v_1^1 and v_2^1 incorrect in the $node_1$

$node_1$接收到编码数据$u_{2,1}$、$u_{3,1}$、$u_{4,1}$、$u_{5,1}$。由于上文假设$node_1$中v_1^1和v_2^1为错误,因此$u_{2,1}\oplus v_1^1\neq v_2^1$,其原因为编码数据包$u_{2,1}$发生破坏或者$v_1^1$和$v_2^1$中的某一项计算错误,同时由于$u_{3,1}\oplus v_1^1\neq v_3^1$、$u_{4,1}\oplus v_2^1\neq v_4^1$、$u_{5,1}\oplus v_3^1=v_4^1$,因此$v_3^1$和$v_4^1$正确。然后验证$u_{3,1}\oplus v_3^1\oplus u_4^1\oplus v_4^1=v_1^2\oplus v_2^2$是否成立,若成立则表明$node_1$本地数据块计算错误,即$v_1^1$或$v_2^1$计算错误,但收到的编码数据包全部正确。此时,$node_1$通过$u_{3,1}\oplus v_3^1$得到$v_1^3$,同时利用$u_{4,1}\oplus v_4^1$得到$v_2^4$。当$node_1$获得所有正确的中间结果$\{v_1^3,v_2^4,v_3^1,v_4^4\}$时,通过reduce函数即可得到当前节点

的输出结果 $r_1 = H(v_1^3, v_2^4, v_3^1, v_4^1)$。$node_6$ 经过执行上述步骤得到正确结果 r_6，因此当前集群最终的输出结果为 $r = H(r_1, r_6)$。

3　实验结果与分析

3.1　实验方案

本文分布式计算的测试程序为 Terasort[22]，CTMR 算法的评价指标为任务执行总时间、map 和 shuffle 阶段执行时间以及平均故障修复时间（Mean Time to Repair，MTTR），对比算法为 TMR 和 two-stage TMR 算法。

实验使用多台虚拟机搭建的分布式集群，包括 1 个管理节点和 6 个工作节点，节点间的带宽为 100 Mb/s。实验中动态选择发生故障的节点个数，随机选取节点并修改其对应数据块数值实现故障注入。假设系统在单位时间内的故障发生概率服从泊松分布 $p(x=k)=\frac{\lambda^k}{k!}\cdot e^{-\lambda}$，即在单位时间内出现 k 个故障的概率为 $p(k)$，本文中 λ 的取值为 2。因此，在满足泊松分布的条件下，假设该分布式系统随机产生 k 个故障。如果各个故障之间相互独立，那么在连续运行的分布式系统中，当产生 k 个故障时的无故障运行时间为 t_k，则系统平均无故障运行时间如式（9）所示：

$$T = \sum t_k \cdot p(k) \tag{9}$$

故障修复时间为检测到故障直至故障修复的时间，假设有 k 个故障时的故障修复时间为 θ_k，则平均故障修复时间如式（10）所示：

$$T_{\mathrm{MTTR}} = \sum \theta_k \cdot p(k) \tag{10}$$

实验中每个 MapReduce 任务可以分为 map、shuffle、check 和 reduce 这 4 个阶段。在 map 阶段，管理节点按照 CTMR 算法要求将用户输入数据分发给 6 个工作节点，同时指定 2 个校验节点。每个工作节点对本地数据集进行排序，得到相应的中间结果集。在 shuffle 阶段，每个节点将其中间结果编码，发送给之前指定的校验节点。在 check 阶段，校验节点将收到的数据包进行故障检测和恢复，校验成功后得到相应结果。在 reduce 阶段，管理节点收到校验节点发送来的部分 reduce 计算结果后执行 reduce 函数得到最终输出结果。任务执行总时间 T_{total} 如式（11）所示：

$$T_{\mathrm{total}} = T_{\mathrm{map}} + T_{\mathrm{shuffle}} + T_{\mathrm{check}} + T_{\mathrm{reduce}} \tag{11}$$

3.2　实验结果

图 3 给出了 CTMR、two-stage TMR 以及 TMR 算法的任务执行总时间对比结果。可以看出，CTMR 算法能有效降低分布式计算的任务执行总时间。当故障个数较少时，CTMR 算法的执行效率远高于另外两种算法。随着故障个数的不断增加，two-stage TMR 算法由于需要重新执行第 3 个副本做验证，而 CTMR 算法则需要更换节点做验证，在该过程中需要重新发送编码数据包，因此这两种算法的任务执行总时间也会随之增加。

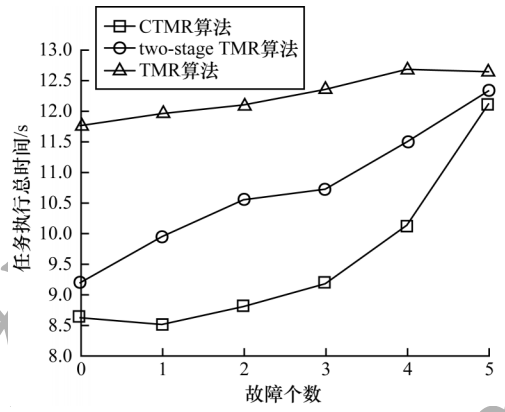

图 3　任务执行总时间对比

Fig.3　Comparison of total task execution time

图 4 给出了 CTMR、two-stage TMR 以及 TMR 算法在 map 阶段的执行时间对比结果。可以看出，随着故障个数的增加，two-stage TMR 算法由于第 1 次投票选择的 2 个副本对应的中间结果不同，因此需要进行第 2 次投票。这时会选择第 3 个副本并执行 map 任务，因此 map 阶段所需时间随故障个数的增加不断增加。TMR 和 CTMR 算法由于最初都要对 3 个副本执行 map 任务，因此 map 阶段的执行时间基本保持不变。

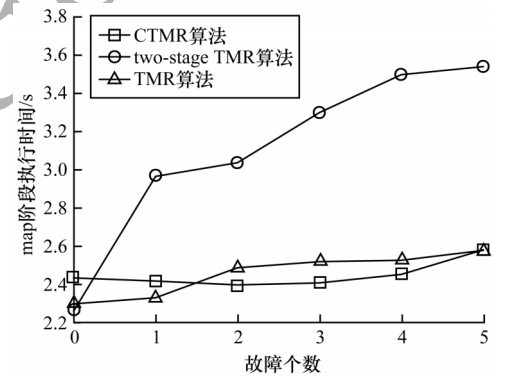

图 4　map 阶段执行时间对比

Fig.4　Comparison of the execution time in the map phase

图 5 给出了 CTMR、two-stage TMR 以及 TMR 算法在 shuffle 阶段的执行时间对比结果。可以看出，CTMR 算法在 shuffle 阶段所需时间明显低于 TMR 算法，并且相比 two-stage TMR 算法有一定程度的减少。本文将 shuffle 阶段的执行时间作为通信开销的衡量指标，当系统在单位时间内的故障发生概率服从泊松分布 $p(x=k)=\frac{\lambda^k}{k!}\cdot e^{-\lambda}$ 时，可以计算得出发生

k个故障的概率为$p(k)$,本文中λ的取值为2。根据式(9)可计算出CTMR、two-stage TMR以及TMR算法在shuffle阶段的执行时间分别为1.90 s、2.21 s和3.22 s。因此,CTMR算法在shuffle阶段的执行时间相比TMR算法降低了41.0%,相比two-stage TMR算法降低了14.0%。

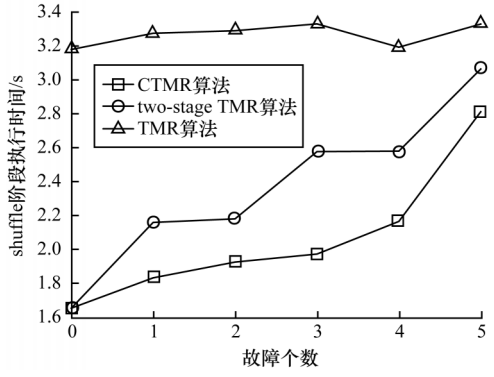

图5 shuffle阶段执行时间对比

Fig.5 Comparison of the execution time in the shuffle phase

图6给出了CTMR、two-stage TMR以及TMR算法在check阶段的执行时间对比结果。可以看出,CTMR算法在一定的故障个数范围内,故障修复效率明显优于TMR与two-stage TMR算法。随着故障个数的不断增加,TMR和two-stage TMR算法均需要对所有副本进行第3次投票,而CTMR算法也需要更换节点进行校验,因此3种算法的故障恢复时间不断增加并最终趋于一致。

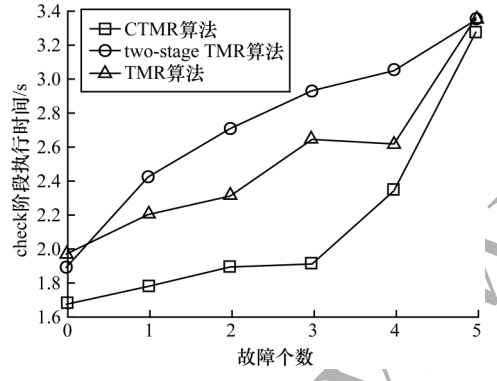

图6 check阶段执行时间对比

Fig.6 Comparison of the execution time in the check phase

当系统在单位时间内发生故障的概率服从泊松分布时,根据式(9)分别计算出CTMR、two-stage TMR以及TMR算法在tatal阶段的任务执行总时间以及map、shuffle和check阶段的任务执行时间,结果如图7所示。可以看出,CTMR算法的任务执行总时间相比TMR算法降低了25.8%,相比two-stage TMR算法降低了13.2%。根据式(10),CTMR算法的平均故障修复时间相比TMR算法降低了18.3%,相比two-stage TMR算法降低了26.2%。

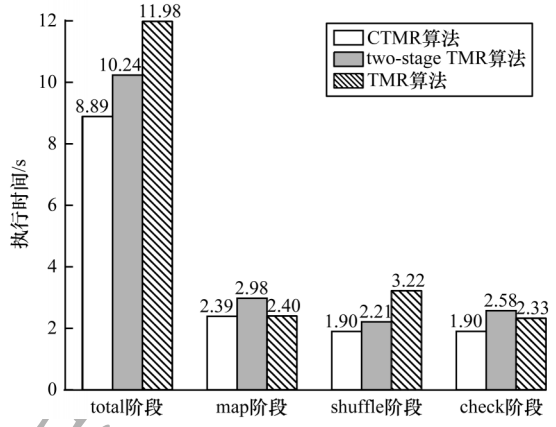

图7 故障发生概率服从泊松分布时3种算法的执行时间对比

Fig.7 Comparison of the execution time of the three algorithms when the probability of failure obeys the Poisson distribution

4 结束语

为降低MapReduce分布式计算中容错算法的通信开销,本文结合副本冗余技术和分布式编码计算技术,提出一种新的容错算法。实验结果表明,CTMR算法在完成容错计算的同时,相比TMR和two-stage TMR容错算法,平均降低了41.0%和14.0%的shuffle阶段的通信开销以及18.3%和26.2%的平均故障修复时间,并且提高了分布式系统的可用性和可靠性。但由于本文中的副本数量固定为3具有一定的局限性,因此下一步将根据分布式系统的故障发生概率,通过动态调整副本数量以增强容错算法的灵活性。

参考文献

[1] SARI A, AKKAYA M. Fault tolerance mechanisms in distributed systems[J]. International Journal of Communications, Network and System Sciences, 2015, 8(12): 471-482.

[2] MARIANI L, PEZZE M, RIGANELLI O. Predicting failures in multi-tier distributed systems [EB/OL]. [2020-02-15]. https://arxiv.org/abs/1911.09561.

[3] ITANI M, SHARAFEDDINE S, ELKABANI I. Dynamic single node failure recovery in distributed storage systems[J]. Computer Networks, 2017, 113: 84-93.

[4] GUO Baolong, WANG Jian, YAN Yunyi, et al. Optimal design of DSP protection based on multi-target PSO algorithm[J]. Computer Engineering, 2018, 44(4): 74-80. (in Chinese)
郭宝龙,王健,闫允一,等. 基于多目标PSO算法的DSP防护优化设计[J]. 计算机工程,2018,44(4):74-80.

[5] LEI Changjian, LIN Yaping, LI Jinguo, et al. Research on Byzantine fault tolerance under volunteer cloud environment[J]. Computer Engineering, 2016, 42(5): 1-7. (in Chinese)
雷长剑,林亚平,李晋国,等. 志愿云环境下的拜占庭容错研究[J]. 计算机工程,2016,42(5):1-7.

[6] BERROCAL E, BAUTISTA-GOMEZ L, DI S, et al. Toward general software level silent data corruption detection for parallel applications[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(12): 3642-3655.

[7] LI S Z, MADDAH-ALI M A, QIAN Y, et al. A fundamental tradeoff between computation and communication in distributed computing[J]. IEEE Transactions on Information Theory, 2018, 64(1): 109-128.

[8] REISIZADEH A, PRAKASH S, PEDARSANI R, et al. Coded computation over heterogeneous clusters [J]. IEEE Transactions on Information Theory, 2019, 65(7): 4227-4242.

[9] KONSTANTINIDIS K, RAMAMOORTHY A. Leveraging coding techniques for speeding up distributed computing[C]// Proceedings of 2018 IEEE Global Communications Conference. Washington D. C., USA: IEEE Press, 2018: 1-6.

[10] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[11] LI S Z, QIAN Y, MADDAH-ALI M A, et al. Coded distributed computing: fundamental limits and practical challenges [C]//Proceedings of the 50th Asilomar Conference on Signals, Systems and Computers. Washington D. C., USA: IEEE Press, 2016: 509-513.

[12] D'ANGELO G, FERRETTI S, MARZOLLA M. Fault tolerant adaptive parallel and distributed simulation through functional replication[J]. Simulation Modelling Practice and Theory, 2019, 93: 192-207.

[13] LEDMI A, BENDJENNA H, HEMAM S M. Fault tolerance in distributed systems: a survey[C]//Proceedings of the 3rd International Conference on Pattern Analysis and Intelligent Systems. Washington D. C., USA: IEEE Press, 2018: 1-5.

[14] LIAO Weicheng, WU Janjan. Replica-aware job scheduling in distributed systems[C]//Proceedings of Advances in Grid and Pervasive Computing. Berlin, Germany: Springer, 2010: 290-299.

[15] BARKAHOUM K, HAMOUDI K. A fault-tolerant scheduling algorithm based on check pointing and redundancy for distributed real-time systems[J]. International Journal of Distributed Systems and Technologies, 2019, 10: 58-75.

[16] LYONS R E, VANDERKULK W. The use of triple-modular redundancy to improve computer reliability[J]. IBM Journal of Research and Development, 1962, 6(2): 200-209.

[17] FU M, HAN S J, LEE P P C, et al. A simulation analysis of redundancy and reliability in primary storage deduplication [J]. IEEE Transactions on Computers, 2018, 67(9): 1259-1272.

[18] SALEHI M, KHAVARI TAVANA M, REHMAN S, et al. Energy-efficient permanent fault tolerance in hard real-time systems [J]. IEEE Transactions on Computers, 2019, 68(10): 1539-1545.

[19] XU Wenfang, LIU Hongwei, SHU Yanjun, et al. Management board for triple module redundant fault-tolerance system[J]. Journal of Tsinghua University(Science and Technology), 2011, 51(S1): 1434-1439. (in Chinese)
徐文芳,刘宏伟,舒燕君,等. 三模冗余容错系统管理板[J]. 清华大学学报(自然科学版),2011,51(S1):1434-1439.

[20] ZHOU Ao, WANG Shangguang, CHENG Bo, et al. Cloud service reliability enhancement via virtual machine placement optimization[J]. IEEE Transactions on Services Computing, 2017, 10(6): 902-913.

[21] LI Xin, LIN Yufei, GUO Xiaowei. A triple modular eager redundancy fault-tolerant technique for distributed stream architecture[J]. Computer Engineering and Science, 2015, 37(12): 2233-2241. (in Chinese)
李鑫,林宇斐,郭晓威. 面向分布式流体系结构的多副本积极容错技术[J]. 计算机工程与科学,2015,37(12):2233-2241.

[22] O'MALLEY O. TeraByte sort on Apache Hadoop[EB/OL]. [2020-02-15]. http://sortbenchmark. org/YahooHadoop. pdf.

编辑　陆燕菲