MPI 4.0 会话模型中的故障感知

Roberto Rocco
米兰理工大学电子、信息与生物工程系 米兰，意大利
roberto.rocco@polimi.it

Gianluca Palermo 米兰理工大学电子、信息与生物工程系 米兰，意大利
gianluca.palermo@polimi.it

Daniele Gregori E4 Computer Engineering Spa 斯坎迪亚诺（RE），意大利
daniele.gregori@e4company.com

arXiv:2303.02956v1 [cs.DC] 6 Mar 2023

摘要

MPI 的最新版本引入了会话模型等新功能，但仍缺乏故障管理机制。以往的工作开发了用于管理故障的工具和 MPI 标准扩展，其中包括用户级故障缓解。这些措施能够有效应对故障，但无法全面支持标准中的新增功能。本文将用户级故障缓解的故障管理能力与标准 4.0 版引入的新会话模型功能相结合。我们重点关注通信器创建过程，指出其中的关键问题，并提出一种规避方法。实验评估表明，所提出的解决方案在更好地管理故障发生的同时，不会显著影响应用程序的执行时间和可扩展性。

CCS 概念

• **计算方法** → 并行计算方法； • **计算机系统组织** → **可靠性**。

关键词

高性能计算、MPI 会话、用户级故障缓解、容错

ACM 参考文献格式：

Roberto Rocco、Gianluca Palermo 和 DanieleGregori。2023 年。MPI 4.0 会话模型中的故障感知。载于 *ACM* 国际计算前沿会议论文集（*CF '23*）。ACM，美国纽约州纽约，5 页。https://doi.org/XXXXXXX.XXXXXXX

1 引言

计算能力的提升正在推动高性能计算（HPC）领域的创新。未来（以及当前）的百亿亿次级集群为应用开发人员、系统管理和框架设计人员带来了新的挑战，也促使高性能计算领域中一些已经成熟的工具不断发展。其中一种工具便是消息传递接口（MPI），即事实上的进程间通信标准。随着其最新版本（v4.0）[7],的发布，MPI 标准引入了会影响应用程序处理通信方式的功能。新的会话模型[4] 支持在多个彼此独立的实例中进行通信初始化从而实现不同应用模块之间的解耦。它还避免了在不需要时创建全局通信器，从而节省同步和初始化开销。

参与执行的节点数量越多，发生故障的可能性也越高。然而，消息传递接口标准并未规定故障发生后应用程序应如何处理。许多研究致力于提出工具，在消息传递接口应用程序中引入故障管理技术，其中用户级故障缓解（ULFM）扩展 [2] 目前最受关注。该扩展提出了一组额外功能，用于修复受故障影响的通信器。ULFM 扩展面向经典的初始化模型，但其方法也能使采用会话模型的应用程序受益。

在本研究中，我们进一步分析了采用会话模型的应用程序中 ULFM 的用法。我们展示了朴素方案的局限性和关键问题，并提出了 Horizonset 用法；该用法在实现故障支持的同时，保留了会话模型的优势。实验评估证明，我们的解决方案在正确管理故障的同时，不会损害应用程序的可扩展性和性能。

本文结构如下：第2 节介绍会话模型和 ULFM 扩展的基本概念，并回顾相关工作；第3 节探讨故障对两种模型初始化阶段的影响；第4 节讨论我们针对该问题提出的方法；第5节展示用于验证解决方案的实验评估。最后，第6 节总结全文。

2 背景

本节讨论影响本研究的主要概念和相关工作。我们将在2.1节介绍 MPI 会话模型的基础知识，在2.2节介绍用户级故障缓解的基本原则，并在 2.3节回顾以往的工作。

2.1 MPI 会话模型

引入会话模型的主要动因，是让不同应用模块之间的通信彼此隔离。经典 MPI 方法（World 模型）的问题在于，初始化调用只能唯一执行。如果许多应用组件、模块或线程都希望使用 MPI 功能，就必须相互协调，确保只有一个调用 MPI_Init 函数，这会损害各部分之间的隔离性。另一方面，会话模型允许每个组件、模块或线程独立于其他部分初始化和管理自己的通信。这一特性有助于软件实现模块化

特此授予许可，可免费为个人或课堂使用而制作本作品的全部或部分数字或纸质副本，但前提是不得出于营利或商业利益制作或分发副本，且副本必须附有本声明及首页上的完整引文。对于本作品中由 ACM 以外的其他方拥有版权的组件，必须遵守其版权规定。在注明出处的情况下允许摘编。若要以其他方式复制、再版、发布到服务器或重新分发到列表，必须事先获得具体许可和/或支付费用。请通过 permissions@acm.org 联系我们申请许可。*CF '23*，*2023*年*5*月*9*日至*11*日，意大利博洛尼亚©2023 计算机协会。ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00https://doi.org/XXXXXXX.XXXXXXX

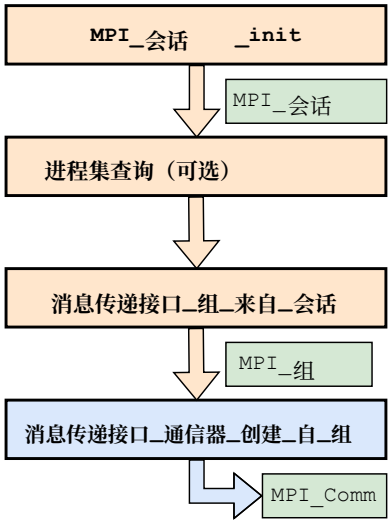


图 1：会话模型的执行流程，从会话创建到通信器生成。橙色框表示本地操作，绿色框表示生成的句柄，蓝色框表示需要同步的函数。

并减少协调需求，使会话方法更具可扩展性，也更适合百亿亿次计算场景。

World 模型的初始化过程完全通过 消息传递接口_Init 调用完成，而会话模型则需要更复杂的流程。图 1 概述了会话模型的创建流程。大多数函数都是本地操作，因此无需通信，只有 消息传递接口_通信器_create_from_组 调用例外，因为该调用需要未来通信器中的所有进程参与。通信器生成后，应用程序的运行方式与 World 模型类似。

2.2 用户级故障缓解扩展

MPI 标准未规定发生故障后的执行行为，实际上排除了任何故障管理技术。许多研究尝试提出解决方案来绕过这一限制，其中用户级故障缓解（ULFM） [2] 目前最受关注。它是 MPI 标准扩展，即使某个进程停止工作，也允许执行继续进行，并引入了用于传播故障、移除故障，以及在发生故障的情况下达成一致的调用（分别为 撤销、收缩 和 达成一致 函数）。然而，故障可能会影响执行的正确性，因此需要额外的恢复技术。

用户级故障缓解将故障视为计算的突然、非计划终止。其他故障类型，例如静默数据损坏和时序错误，通常需要采用特定措施，因此不在用户级故障缓解的分析范围内。此外，用户级故障缓解不提供任何恢复机制，而是由应用开发人员根据自身需求选择恢复机制。

用户级故障缓解的故障管理功能适用于从通信器创建到销毁的整个过程。鉴于通信器创建之后的行为与 World 模型中的行为相同，这一假设使用户级故障缓解方法能够适用于会话模型。然而，用户级故障缓解没有规定在 MPI_Init 调用之前及调用期间发生故障时应用程序的行为。这一限制对 World 模型影响不大，因为初始化之前无法调用任何 MPI 例程，所以应用程序会尽早执行初始化。另一方面，会话模型带来了更大的灵活性，因此我们可以避免调用 MPI_Init 函数，也可以多次调用会话初始化过程，甚至在执行过程中更晚的时候调用，从而延长可能出现未定义行为的时间段。在本研究中，我们尝试扩展用户级故障缓解所定义的行为，并分析会话模型的初始化阶段。

2.3 相关工作

除用户级故障缓解扩展外，消息传递接口容错领域的另一个主要方向是 Reinit 提案 [3, 6]。其主要特点是可以多次调用消息传递接口_Init 函数。发生故障时，无需修复进程，因为重新初始化通信器即可。Reinit 方案与会话的多重性类似，但仅关注 World 模型；与用户级故障缓解相比，它在会话模型中不会带来额外收益。

在所有采用用户级故障缓解扩展的研究中， [9] 分析了消息传递接口_通信器_Create_From_Group 函数，并提出了一种解决方案，使该调用即使在涉及的进程发生故障时也能完成。他们引入了一种活性发现算法，用于查询执行情况并主动发现失效。该方案前景良好，因为它处理了会话初始化过程中的一个关键函数，但其适用范围仅限于 World 模型。

一种截然不同的方法来自会话工作组（负责在消息传递接口标准内开发会话模型）。他们目前正在将会话模型背后的理念扩展为隔离的通信气泡 [5]。这项工作应能在消息传递接口中引入形式化的可塑性，使执行能够动态重配置。此外，这些优势也有助于容错，因为它们允许将故障作为资源变化来处理。尽管这是一个很有前景的方向，但容错分析仍需额外投入：气泡理念可以表示故障带来的后果，但仍需要有效的检测和传播机制。这项工作与气泡提案正交，因为它作用于不同层面，同时两种方法彼此兼容。

3 初始化流程中的失效

本节分析故障对两种消息传递接口执行模型的初始化流程所产生的影响。作为参考，我们采用启用了用户级故障缓解功能的消息传递接口实现。否则，无论我们采取何种措施，一旦发生故障，执行都会停止。

在 World 模型中，初始化过程仅由 MPI_Init 函数组成。它涉及参与执行的所有进程，并生成初始通信器。如果某个进程未参与调用，其他进程将一直等待，直到它加入。如果该进程无法加入，执行就会进入死锁状态；如果某个进程在函数调用前发生故障，也会出现同样的情况。从执行角度来看，无法判断该函数最终会终止，还是会陷入死锁

状态：这一判断类似于停机问题 [10]。不过，这个易受死锁影响的函数是每个 MPI 应用程序的首次调用，因此将易受影响的时间窗口降至最低。总体而言，死锁风险是可接受的，因此用户级故障缓解可以专注于处理应用程序其余部分中的故障。

在会话模型中，情况有所不同：初始化流程中的几乎所有函数都是本地操作，因此无需通信，即使发生故障也能给出结果。唯一的非本地操作是 MPI_Comm_create_from_group 调用，它需要所有参与创建通信器的进程共同参与。与 MPI_Init 函数类似，创建通信器的调用无法区分故障和延迟；只要有进程未参与，就会导致死锁。 然而，与 MPI_Init 函数不同，初始化并非唯一的，可以在应用程序执行过程中的任意时刻进行，而且每个会话中可以进行多次。这些因素意味着死锁风险影响很大，发生概率中等，因此这个问题值得关注。

4 Horizon 集合

这项工作 [9] 从故障管理的角度分析了 MPI_Comm_create_from_group 函数，消除了其行为中发生死锁的可能性。作者借助活性发现算法（LDA）来探索组并移除故障进程。他们的方法要求预先创建一个通信器，其中包含所有打算加入新通信器的进程，因此他们选择了 MPI_COMM_WORLD。由于 MPI_COMM_WORLD 只能在 World 模型中使用，因此只有在存在另一个通信器时，该方法才适用。然而，要创建这样的通信器，我们只能使用 MPI_Comm_create_from_group 函数，而这仍可能导致死锁。这种情况就像先有鸡还是先有蛋的问题。尽管如此，如果我们提供一个正确的通信器，就可以使用 LDA 算法，消除发生死锁的可能性。本分析旨在创建这样一个通信器，从而降低死锁漏洞，同时保持应用程序的性能。

为了更直观地说明这个问题，我们引入一组概念。对于给定的进程组 G，我们将生成一个通信器 C_G（包含 G 中所有进程）的调用定义为 $Create(C_G)$。此外，我们将 超集 $S(G)$ 定义为包含至少所有 G 中进程的通信器集合。根据这些定义，可以推导出以下简单假设：

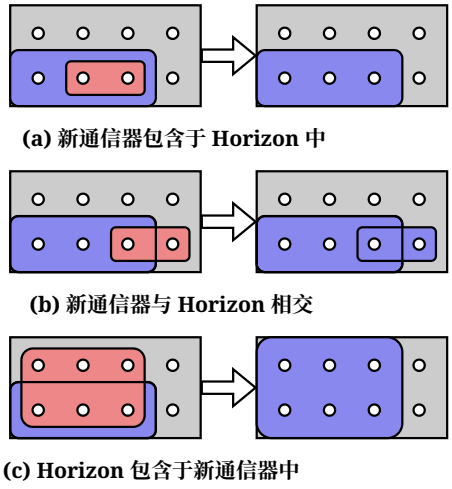
A.1 $\forall G : C_G \in S(G) \wedge \texttt{MPI_COMM_WORLD} \in S(G)$

A.2 $\forall G1, G2 : C_{G2} \in S(G1) \Rightarrow G2 \supseteq G1$

第一个假设保证每个组的超集都不为空，而另一个假设则定义了不同超集之间的关系。活性发现算法（LDA）和死锁消除的要求是：存在一个 Create 通信器的调用，该通信器属于作为参数传入的组的超集。形式化地说：

$$LDA\ applicable\ for\ Create(C_G) \Leftrightarrow \exists K \in S(G) \mid Create(K)\ prior\ to\ Create(C_G) \quad (1)$$

根据上述分析，如果我们想尽可能降低发生死锁的可能性，就必须定义一个通信器集合 $W \mid K \in S\ G$ W $\forall G\ \exists K \in$，使得 ()。我们应在首次调用通信器创建调用之前初始化 W 内的所有通信器，这样，对于每个



(a) 新通信器包含于 Horizon 中

(b) 新通信器与 Horizon 相交

(c) Horizon 包含于新通信器中

图 2：引入新的通信器（红色）时 Horizon 集合（蓝色）的演变。灰色矩形表示全局通信器，其中的每个白色圆圈表示一个进程。其行为取决于是否存在包含关系（情况2a和2c）或不存在包含关系（情况2b）。

连续的 $Create(C_G)$，公式 1 成立。此外，根据假设 **A.1**，我们可以推断，最小W集合仅包含一个与 消息传递接口_通信器_全局等价的通信器。由此可得，针对死锁漏洞问题的朴素方案是在首次调用 消息传递接口_会话_初始化时生成一个全局通信器，并且该调用会先于任何其他操作发生。然而，这种方案存在许多问题：

- MPI_Session_init 函数是本地函数，因此无需进程间通信。通信器创建需要协调，这改变了该函数的特性。
- MPI_Session_init 调用必须尽早进行，以尽量缩短漏洞窗口，但这会损害会话模型的灵活性。
- 会话模型的优势之一是具有可扩展性，因为它不要求应用程序初始化整个全局通信器，而只需初始化所需部分。该方案失去了这一优势。

因此，这种朴素方案并不充分，但它是一个很好的起点。它的主要弱点在于，无论应用程序的特性如何，都需要创建一个全局通信器。此外，采用该方案会损失会话模型的一些优势，因此我们选择了另一种方法。

虽然使用全局通信器是最安全的选择，但并非唯一选择。为了进一步分析，我们提出使用*Horizon* 集合s $H(t)$，其定义使得以下公式成立：

$$\forall G: \ Create(C_G)\ prior\ to\ t \Rightarrow \exists C \in H(t) \mid C \in S(G) \wedge Create(C)\ prior\ to\ t \quad (2)$$

换言之，在每个时刻 t，一个 Horizon 集合至少包含此前某次通信器创建调用中作为参数传入的所有组的超集中的一个元素。此外，结合公式2 与公式1，我们推导出 LDA 为

如果 Horizon 集合中存在作为参数传入的组的超集中的一个元素，则 LDA 适用。形式化地说：

$$\forall Create(C_G) : Create(C_G)\ happens\ at\ t+1$$
$$LDA\ applicable\ for\ Create(C_G) \Leftrightarrow$$
$$\exists K \in H(t) \mid K \in S(G) \quad (3)$$

方程 3 将 Horizon 集合与我们的问题联系起来，因此我们可以通过分析前者来解决后者。

Horizon 集合会随时间演化，随着新通信器的加入而增大。所有在时间 t 之前创建的通信器构成一个 Horizon 集合，我们可以利用它来判断 LDA 算法是否适用。然而，该 Horizon 集合可能不是基数最低的集合。我们将最小 Horizon 集合定义为 $H^*(t)$。如果希望降低通信器管理所带来的开销，就必须关注最小 Horizon 集合。

我们可以通过分析 $H^*(t)$ 在通信器创建的所有可能拓扑情况下的演化来实现这一点，如图 2所示。一般而言，如果最小 Horizon 集合中已经存在包含某个通信器的另一个通信器，则不应将该通信器加入最小 Horizon 集合（图 2a）。另一方面，新插入的通信器会从 Horizon 集合中移除其所包含的所有通信器（图 2c）。基于这些概念，我们可以跟踪 $H^*(t)$ 并利用它消除某些通信器创建调用发生死锁的可能性。

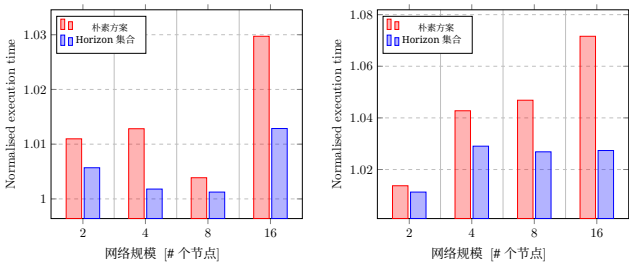
在应用程序代码中，对 $Create(C_G)$ 的调用表示调用函数 MPI_Comm_create_from_group 或 MPI_Init（后者会创建 MPI_COMM_WORLD）。通过跟踪这些函数，我们可以正确更新最小 Horizon 集合，并用它来限制会话模型初始化流程中的死锁漏洞。此外，我们提供了一个新的专用集体函数 MPIX_Horizon_from_group，它会将作为参数传入的组的通信器纳入 Horizon 集合。虽然其行为与通信器创建函数类似，但它允许用户指定应用程序的通信意图。

5 实验结果

我们将所提出的 Horizon 集合管理功能集成到了 Legiolibrary [8, 9] 中，因为它已经实现了 LDA 算法，并且工作在相同的抽象层级上。所有新增功能都利用 PMPI 层，避免直接修改应用程序代码。

实验评估对所提出的解决方案进行了评估真实场景中的开销和可扩展性。具体而言，我们将该解决方案集成到 NAS 基准测试集中的易并行（ EP ）和数据流量（ DT ）基准测试中 [1]，并在 IT4Innovations Karolina 集群上运行这些测试。该集群配备节点，每个节点搭载 2 颗 AMD Zen 2 EPYCTM 7H12 处理器，主频为 2.6 GHz，内存为 256 GB，每个节点运行 128 个消息传递接口进程。我们使用集成了用户级故障缓解（ULFM）（v5.0.0）并实现消息传递接口标准 4.0 的最新版本 OpenMPI。我们修改了这两个基准测试，使其采用会话模型。两个应用程序均使用“C”规模的工作负载，这是它们可执行的最大规模。

我们使用不同的网络规模进行了测试，以评估解决方案对可扩展性的影响。我们分别在采用朴素方案、支持 Horizon 集合以及不提供故障管理支持的情况下运行这两个应用程序。对于每种配置，我们测量了总完成十次执行所需的时间。图 3 展示了基准测试的执行时间，并以不提供故障管理支持时的数值进行归一化。



(a) DT 基准测试 **(b) EP 基准测试**

图 3：两个基准测试在不同网络规模下的归一化执行时间。

两个基准测试的执行结果各有不同，这是由它们自身的特点所致。具体而言，我们使用 EP 基准测试（图 3b）来考察解决方案的可扩展性，而 DT 则侧重于整体执行影响（图 3a）。从图中可以看出，朴素方案的开销始终高于所提出的 Horizon 集解决方案。此外，朴素方案的开销会随着网络规模增大而增加，如图 3b所示。另一方面，Horizon 集解决方案不会显著影响应用程序的可扩展性，因为它无需像朴素方案那样初始化全局通信器。

除了以性能为导向的实验外，我们还进行了功能测试，以证明我们方法的弹性。测试表明，采用我们解决方案的应用程序具有更强的抗死锁能力，即使执行过程中出现故障，也能完成运行。

6 结论

在本研究中，我们分析了使用会话模型的消息传递接口应用程序中的故障影响，并对其进行了限制。我们阐述了引入 Horizon 集合的必要性，并分析了其行为。我们展示了所提出的 Horizon 集合管理功能的实现，并在真实场景中使用基准测试对其进行了测试。结果表明，Horizon 集合方案不会损害执行的性能和可扩展性，能够降低故障导致的死锁漏洞，且表现优于朴素方案。

该领域的未来工作包括与 MPI Bubbles 提案进行对接，并使用新的基准测试开展更多评估，以进一步检验会话模型的能力。

参考文献

[1] David Bailey、Tim Harris 等。1995。*NAS 并行基准测试 2.0*。技术报告。技术报告 NAS-95-020，NASA 艾姆斯研究中心。[2] Wesley Bland、Aurelien Bouteiller 等。2013。失效后恢复 MPI 通信能力：设计与原理。《国际高性能计算应用期刊》27, 3（2013），244–254。[3] Sourav Chakraborty、Ignacio Laguna 等。2020。EReinit：面向批量同步 MPI 应用程序的可扩展、高效容错。《并发与计算：实践与经验》32, 3（2020），e4863。[4] Daniel Holmes、Kathryn Mohror 等。2016。MPI 会话：利用运行时基础设施提升百亿亿次级应用程序的可扩展性。载于第 *23* 届欧洲 *MPI* 用户组会议论文集。121–129。[5] Dominik Huber、Maximilian Streubel 等。2022。借助 MPI 会话和 PMIx 实现动态资源管理。载于第 29 届欧洲

MPI 用户组会议。57–67。[6] Ignacio Laguna、David F Richards 等。2014。评估面向 MPI 应用程序的用户级容错。载于第 *21* 届欧洲 *MPI* 用户组会议论文集。57–62。[7] 消息传递接口论坛。2021。*MPI*：消息传递接口标准 *4.0* 版。https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[8] Roberto Rocco、Davide Gadioli 等。2021 年。Legio：面向易于并行化的消息传递接口应用程序的故障恢复能力。《超级计算杂志》（2021），1–21。[9] Roberto Rocco 和 Gianluca Palermo。2022 年。消息传递接口中的故障感知非集体通信创建与修复。https://doi.org/10.48550/ARXIV. 2209.01849[10] 艾伦·麦席森·图灵等。1936 年。论可计算数及其在判定问题中的应用。数学杂志58，345-363（1936），5。