DafnyMPI：用于验证消息传递并发程序的 Dafny 库

ALEKSANDR FEDCHIN，塔夫茨大学，美国；中亚美国大学，吉尔吉斯斯坦ANTERO MEJR，塔夫茨大学，美国HARI SUNDAR，塔夫茨大学，美国JEFFREY S. FOSTER，塔夫茨大学，美国

arXiv:2512.18842v1 [cs.PL] 21 Dec 2025

消息传递接口（MPI）广泛用于并行高性能编程，但编写使用 MPI 的无错误软件仍然十分困难。我们提出了 DafnyMPI，这是一种新颖且可扩展的 MPI 软件形式化验证方法。DafnyMPI 支持证明无死锁性、终止性以及与更简单的顺序实现之间的函数等价性。与现有的专用框架不同，DafnyMPI 避免使用定制的并发逻辑，而是依赖 Dafny——一种用于顺序程序、面向验证的编程语言——并为其扩展并发推理能力。DafnyMPI 以库的形式实现，通过要求用户预先指定通信拓扑，并验证对通信原语（如 MPL_发送 和 MPL_等待）的调用满足其前置条件，从而实现安全的 MPI 编程。我们使用核心演算对 DafnyMPI 进行形式化，并证明这些前置条件足以保证无死锁性。我们通过对消息负载进行依赖—保证推理，并借助一个保证安全使用读写缓冲区的系统，证明函数等价性。我们使用标准的 Dafny 技术证明终止性和不存在运行时错误。为进一步展示 DafnyMPI 的适用性，我们验证了三个经典偏微分方程的数值解。我们相信，DafnyMPI 展示了如何使形式化验证适用于更广泛的程序类别，并为证明工程师提供了用于并行和并发系统软件验证的更多工具。

CCS 概念：• **软件及其工程** → **形式化软件验证**；**软件验证**；**消息传递**；• **计算理论** → 程序验证；• **计算方法** → 并行编程语言。

其他关键词和短语：消息传递、死锁、依赖-保证、Dafny、消息传递接口

ACM 引用格式：
Aleksandr Fedchin、Antero Mejr、Hari Sundar 和 Jeffrey S. Foster。2026。DafnyMPI：用于验证消息传递并发程序的 Dafny 库。*ACM*程序设计语言会议录 10，POPL，第63篇文章（2026年1月），27 页。
https://doi.org/10.1145/3776705

1 引言

计算科学高度依赖并行性来实现高性能，并支持大规模模拟、预测和决策。并行科学软件的正确性至关重要，但针对这类系统的实用形式化验证技术却寥寥无几。模型检查和动态验证方法 [24, 27, 31, 32, 34]可能受到状态爆炸问题的困扰，而且通常需要假定预先确定的输入和进程数。证明助手（例如采用并发分离逻辑的证明助手） [14, 17], 可以进行任意复杂的推理，

作者联系信息：Aleksandr Fedchin，aleksandr.fedchin@tufts.edu，塔夫茨大学，美国马萨诸塞州梅德福，以及中亚美国大学，吉尔吉斯斯坦比什凯克；AnteroMejr，antero.mejr@tufts.edu，塔夫茨大学，美国马萨诸塞州梅德福；Hari Sundar，hari.sundar@tufts.edu，塔夫茨大学，美国马萨诸塞州梅德福；Jeffrey S. Foster，jfoster@cs.tufts.edu，塔夫茨大学，美国马萨诸塞州梅德福。



本作品采用知识共享署名4.0国际许可协议授权。©2026 版权所有归所有者/作者所有。ACM 2475-1421/2026/1-ART63https://doi.org/10.1145/3776705

但即使是小型程序的验证，也需要专家团队付出大量努力。现有的求解器辅助语言方法 [13, 22] 通常通过支持带有动态线程创建的共享内存并发来强调通用性，但这种通用性可能使推理程序的功能行为变得困难。因此，先前的工作通常聚焦于相对简单的同步模式，例如假设所有接收操作都是阻塞的，而所有发送操作都不是 [22]。

为解决这些局限，我们提出了一种新的、可扩展的方法，用于验证使用消息传递接口（消息传递接口） [26] 的程序。我们的方法支持对阻塞、非阻塞和集合通信行为进行验证。这在消息传递接口环境中至关重要，因为多个消息可能相对于发起进程并发且异步地传输；当缓冲区不可用时，大消息还可能导致发送操作以非确定性方式阻塞。我们能够支持这些行为，但代价是要求用户预先指定通信拓扑，并且依赖于消息传递接口不允许动态创建进程且假设不存在共享内存这一事实。

我们将这一方法实现为一个名为 DafnyMPI 的库，供 Dafny 编程语言使用 [20, 21]。Dafny 通常用于验证顺序程序 [4, 5]，但它缺乏内置的并发支持。通过将 DafnyMPI 集成为一个库，我们展示了这一方法可以轻松集成到现有验证框架中。这也使我们能够保留 Dafny 已有的所有功能，让没有并行软件验证经验的软件工程师也能更容易地使用 DafnyMPI。具体而言，Dafny 已经能够确保终止性，并防止除以零等运行时错误。Dafny 会自动完成大部分证明工作，用户则通过编写前置条件、后置条件和不变式来引导验证。DafnyMPI 在此系统之上构建，提供一个程序可调用的 API，用于进程间通信。如果程序满足对 DafnyMPI 方法调用的前置条件，DafnyMPI 就能保证通信正确性（包括无死锁性）；后置条件则支持对返回值进行推理。（第2 节介绍 Dafny 和 DafnyMPI，并演示如何利用它们对线性对流偏微分方程的并行数值实现进行推理。）

为了证明 DafnyMPI 的规约能够保证无死锁性，我们引入了一种核心演算，该演算紧密模拟了关键 MPI 原语的行为：屏障以及非阻塞式发送和接收操作。消息标签上的排序性质确保，阻塞在具有最低标签的消息上的进程最终会被其通信伙伴发出的匹配命令解除阻塞。至关重要的是，我们的操作语义对 MPI 发送操作的非确定性行为进行了建模：此类操作可能会一直阻塞，直到接收进程在其一端发起通信；或者，如果 MPI 运行时有足够的内存可用，消息可能会被传输到内部缓冲区，从而允许发送方继续执行。（第 3 节正式定义了核心演算及其操作语义，并概述了无死锁性的证明。）

DafnyMPI 的用户可能不仅希望确保其程序正确，还希望对程序的输出进行推理。由于并发代码容易出错，证明并行版本产生的结果与更简单的顺序版本相同通常很有用。DafnyMPI 通过要求用户预先规约每个进程间通信操作的预期结果，并强制安全地使用读写缓冲区，使这一点成为可能。随后，可以通过传递性对两个手动编写的程序（一个并行程序和一个顺序程序）之间的函数等价性进行推理：用户编写一个函数规范，并证明两个实现都满足该规范。（第 4 节介绍了如何使用 DafnyMPI 证明函数等价性。）

为了将 DafnyMPI 应用于实际程序，我们使用标准的 Dafny 技术证明程序会终止，并解决各种工程问题，从实际的消息传递接口约束（例如标签溢出），到证明工程方面的挑战（例如脆弱性）。此外，我们还为若干

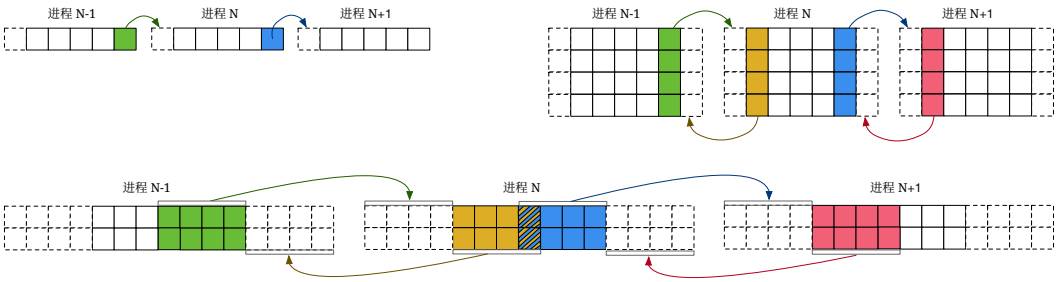

图 1. 线性对流（左上）、泊松方程（右上，转置）和热扩散（下方，转置）偏微分方程求解器实现中的 MPI 通信结构。

实用的高层集合通信操作指定了行为，而我们的核心演算将这些操作抽象为带注释的屏障。总的来说，解决上述所有问题后，DafnyMPI 包含约 2000 行 Dafny代码，其中不到 400 行构成了用于建模核心演算的信任基。（第 5 节讨论了 DafnyMPI 的结构及相关工程考量。）

为展示 DafnyMPI 的实用价值，我们使用它来验证三个偏微分方程（PDE）的数值解：线性对流、热扩散和泊松方程。我们讨论验证每个基准测试所需的证明工程工作量，并展示并行版本复杂度的增加与幽灵代码（不可执行代码）比例的提高相对应。最后，我们将经过验证的 Dafny代码编译为 Python，并展示生成的程序在并行执行下表现出预期的运行时行为——具体而言，进程数增加会使计算速度加快。（第 6 节介绍这些实验。）

总之，本文的主要贡献如下：

- 我们提出了一种验证并发消息传递接口程序，并证明其无死锁性、终止性以及与顺序实现函数等价性的技术。
- 我们将这项技术实现为 DafnyMPI，这是一个面向 Dafny 编程语言的库。
- 我们验证了三个消息传递接口程序，以评估该方法在实践中的优势。结果表明，这些并行实现的运行时性能确实优于相应的顺序实现。

DafnyMPI 及相关基准测试可在以下地址获取：https://doi.org/10.5281/zenodo.17102521总之，通过在一种专为顺序推理设计的语言中验证消息传递接口程序，我们旨在拓展可扩展验证的边界，并支持可靠软件的开发。

2 概述

消息传递接口最适用于通信拓扑明确、并行性旨在优化计算的应用场景。消息传递接口的一个关键用途是求解偏微分方程（PDE），其中每个进程负责计算解的一部分。图1 展示了我们用作基准测试的三个偏微分方程求解器的 MPI 通信结构。三者均通过迭代计算解，且每个进程都必须与邻居交换数据，如图中的箭头所示。本节重点介绍线性对流方程（图 1 左上角），这是三者中最简单的，因为每个进程只发送一个值，也只接收一个值。其余两个基准测试将在第 6节详细讨论。这里，我们先概述偏微分方程背后的数学原理，以说明并发的作用，然后给出相应代码并讨论我们的验证方法论。

2.1 示例：线性对流偏微分方程

线性对流方程描述以恒定速度移动的波。更正式地说，它描述一个随空间和时间变化的量 $u(x,\ t)$，并指出其随时间的变化（$\frac{\partial u}{\partial t}$）与斜率（$\frac{\partial u}{\partial x}$）成正比：

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{1}$$

要解这个方程，必须能够根据初始状态 u_0，求出给定时间步 t 和位置 x 处 u 的值。数值解必须通过时间（Δt）和空间（Δx）上的小离散步长来近似偏导数。一种称为迎风格式 的方法是使用前向差分近似时间导数，并使用后向差分近似空间导数：

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} + c\frac{u(x,t) - u(x-\Delta x,t)}{\Delta x} = 0 \tag{2}$$

求解方程 2 关于 $u(x,t+\Delta t)$，得到：

$$u(x,t+\Delta t) = u(x,t) - c\frac{\Delta t}{\Delta x}(u(x,t) - u(x-\Delta x,t)) \tag{3}$$

换句话说，数值解通过迭代计算空间域 u 中所有位置的 $u(x,t+\Delta t)$。请注意，该计算具有局部性——时间 $t+\Delta t$ 时，位置 x 的值仅取决于时间 t 时位置 x 和 $x-\Delta x$ 处的值。计算的局部性意味着它可以轻松地并行分配给多个进程，每个进程负责计算解的一部分。然而，各进程还必须在每个时间步交换信息，以计算边界值：如果位置 $x-\Delta x$ 不在其分配的空间域范围内，负责计算 $u(x,t+\Delta t)$ 的进程必须从邻居处接收值 $u(x-\Delta x,t)$。图1（左上）用进程之间的箭头展示了这种边界值交换。消息传递接口非常适合实现这种结构化通信。

2.2 使用消息传递接口求解偏微分方程

图 2 左侧的 Python 代码使用消息传递接口实现了方程 3。按照消息传递接口的标准做法，启动程序时会在多个 CPU 核心上同时运行该程序。每个进程首先获取自己的秩以及正在运行的进程总数，也称为大小（第1–3行）。接下来，每个进程计算自己负责的初始状态部分（第 5–8行）。初始条件可以各不相同；为简单起见，这里不进一步说明对应的函数`init`。秩相邻的进程负责空间域中相邻的部分。若进程 A 负责的空间域部分紧邻进程 B 左侧，我们就称进程 A 位于进程 B 的左侧，沿 x 轴方向。

接下来，每个进程根据 u 的方程模拟 nt 个时间步（第 12–32 行）。在每个时间步，进程根据方程 3 更新 u 中属于自己的部分；代码中的 upwind 函数实现了这一计算（第 9–10 行）。除了最左侧的边界值（代码中的 `u[0]`）外，每个进程都会在本地计算其负责的所有点（第 15–16 行）；该边界值取决于相邻进程提供的信息。最左侧的进程 0 只需将边界值发送给右侧的进程（第 18–19 行），而最右侧的进程 $N-1$ 只需接收来自左侧进程的边界值（第 21–22 行）。其他每个进程既要接收来自左侧进程的边界值（第 28–29 行），也要向右侧进程发送一个边界值（第 25–27 行）。请注意，为了提高效率，代码使用即时发送命令（Isend），使这两个操作能够并发执行——即时发送操作会在接收操作之前启动，但只有在第 31 行的等待调用之后才能保证完成。还要注意，每条消息都带有一个由以下内容推导出的唯一整数标签

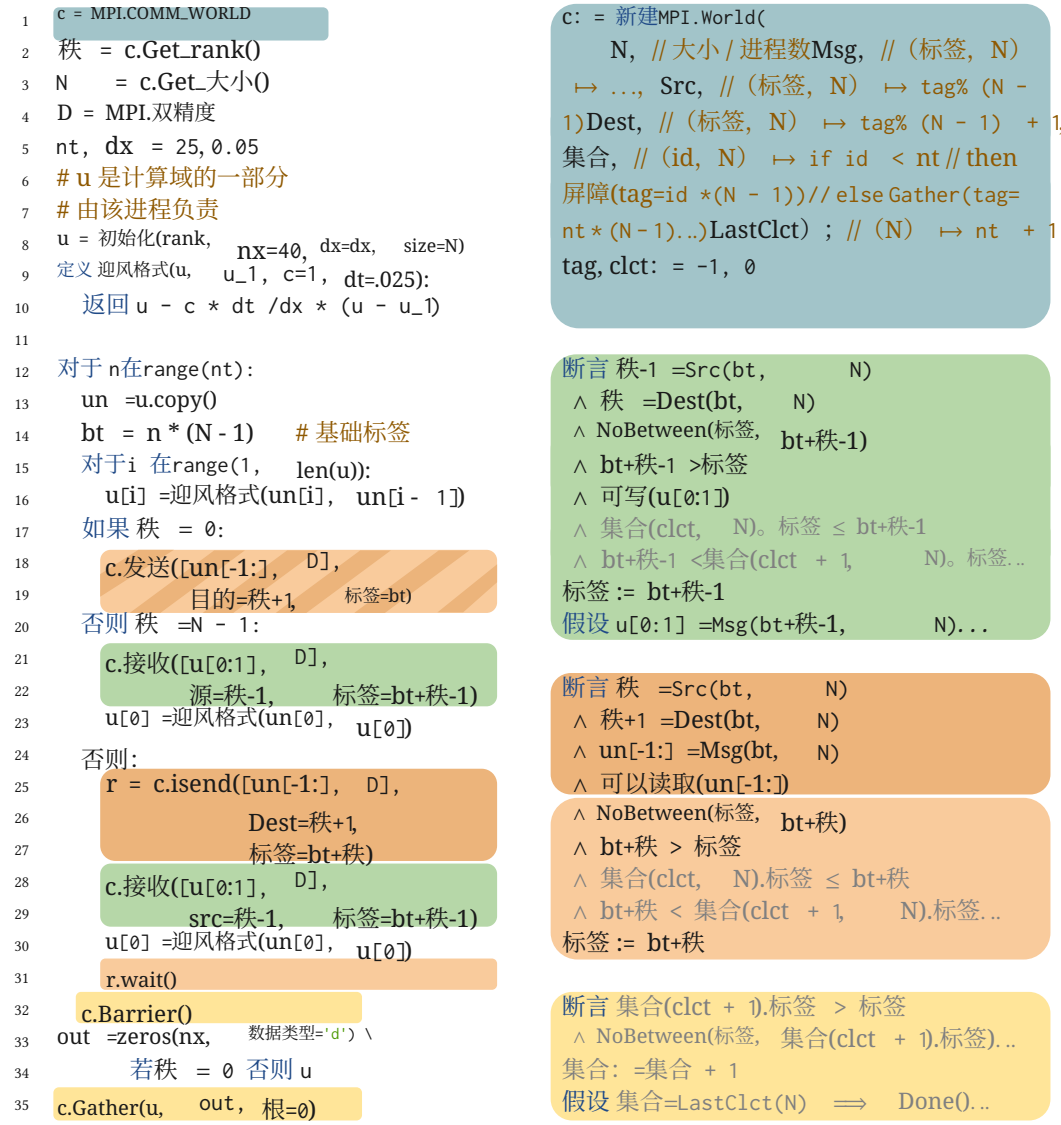

图 2. Python MPI 程序示例（左）及其对应的 Dafny 证明义务（右，简化版）。涉及集体操作的证明义务以灰色显示，以突出点对点通信。

时间步和秩。我们将这些标签用作验证的关键部分，详情见下文。所有发送和接收操作共有的其他参数分别是目标进程或源进程的秩，以及通过引用传递的缓冲区——发送时从中读取消息，接收时向其写入消息。

一轮通信在每个进程调用 `Barrier`（第32行）后完成，该调用会在下一轮通信开始前同步所有进程。经过 `nt` 次迭代后，`Gather`命令会将 `u` 的所有片段收集到由进程 0 管理的单个数组 `out` 中（第 33–35行）。随后，进程 0 可以根据需要处理输出，例如将其保存到磁盘或显示图表。

2.3 DafnyMPI

本文旨在建立一种方法，用于证明图 2 所示的 MPI 程序是正确的。这里的“正确”是指：程序必须成功终止，并且其结果要么与相应顺序实现的结果完全相同，要么与之等价，但允许存在由浮点运算的非结合性以及类似的机器级近似所导致的舍入差异。我们将顺序实现视为参考实现，因为它们通常更易于编写，因此包含错误的可能性更低，也更易于调试。因此，完整的程序正确性取决于以下性质：

- 无死锁性。任何等待发送（接收）的进程最终都会因匹配的接收（发送）而解除阻塞，任何在屏障处等待的进程最终都会通过该屏障。我们在第3节给出了无死锁性的形式化证明。

- 函数等价性。MPI 程序必须计算出与相应顺序实现相同的结果；该顺序实现由程序员手动编写，并作为参考实现。在我们的基准测试中，每个进程计算最终解的一部分，然后将所有这些部分合并，并由根进程（秩为零）返回。DafnyMPI 通过强制安全使用读写缓冲区，并确保所有消息都符合用户提供的规范，使得我们能够推理每个进程的输出。我们将在第 4节讨论这些属性以及证明函数等价性的方法。

- 终止性。每个进程最终都会终止。除我们单独列出的无死锁性之外，终止性还要求不存在无限循环、无限递归以及除以零等运行时错误。Dafny 默认强制执行所有这些属性，因此我们采用标准技术来证明它们。第 5 节讨论终止性以及相关的证明工程考量，例如证明脆弱性。

Dafny。 我们选择使用 Dafny 证明这些性质。Dafny 是一种具有验证能力的语言，能够保证程序终止且不会发生运行时错误，包括除以零、数组越界访问和空指针解引用。Dafny 通过应用最弱前置条件演算来生成验证条件，再由 SMT 求解器消除这些条件，从而自动完成大部分证明工作。Dafny 的用户可以编写前置条件、后置条件和不变式，以引导证明过程并保证其他性质。具体而言，只要一对方法的后置条件都满足相同的函数规范，就可以通过传递性证明它们函数等价。

因此，对于给定的 Python 程序，我们会手动将其转换为等效的 Dafny 程序。随后，我们使用 DafnyMPI 重写程序，使计算并行化，并证明所得版本与原程序函数等价。验证完成后，可以将这两个 Dafny 程序重新编译为 Python，以便执行。

这种方法的主要挑战在于，Dafny 是一种单线程语言，不具备内置的并行性或并发概念。因此，DafnyMPI 的设计具有这样一种元属性：只要其消息传递接口原语的证明义务得到满足，程序的并行执行就能保证满足上述性质。

用于点对点通信的 *DafnyMPI* 证明义务。图 2 右侧展示了我们示例中的一部分 DafnyMPI 证明义务，这些义务以 Dafny代码的形式书写，我们将在下文讨论。在此处以及第 3 节中，我们重点关注无死锁性，因为这是最复杂、最难证明的性质。在图中，源代码与证明义务之间的对应关系以颜色表示。在一种情况下，一条程序语句对应多个证明义务，这通过使用相应颜色的条纹表示。我们将涉及集体通信的证明义务灰显

，以便暂时聚焦于点对点通信；本节稍后将回到集体操作。

在程序开始处，即左侧第 1 行对应的位置，代码会创建一个新的 `MPI.World`实例，其中必须包含程序员提供的、描述程序整体通信模式的规约。具体来说，程序员需要提供大小 `N`（即进程总数；与此处一样，也可以不对其加以约束，以证明该证明对任意大于 1 的进程数均成立），以及一系列函数，用于描述程序对点对点消息和集体操作的使用情况。前三个函数`Msg`、`Src`和`Dest`描述点对点消息，分别将消息标签和大小映射到消息内容、消息源和消息目的地。在我们的示例中，为简洁起见，我们省略了消息内容；消息源和目的地则根据标签和大小计算得出。接下来的两个函数`Clct` 和`LastClct`描述程序的集体操作，包括屏障。`LastClct` 指定程序中集体操作的总数；一般情况下，该数量是进程数的函数。在我们的示例中，屏障始终恰好有`nt` 个——每次循环迭代一个（第12–32行）——此外还有第35行的一次`Gather` 调用，因此`LastClct` 是常数，设为`nt+1`。`Clct` 函数将每个集体操作 ID 映射到其描述信息，包括操作类型（例如`Barrier` 或`Gather`），以及紧随其后的任意点对点消息在所有进程中的最小标签。这使我们能够为集体操作排序，并将其与点对点消息关联起来；否则，集体操作无法分配标签。最后，变量`tag` 被初始化为 -1，用于跟踪最近完成的消息的标签；`clct` 则被初始化为 0，用于跟踪下一个即将执行的集体操作的 ID。

接下来，第`接收` 调用在第21–22 行和第28–29 行对应的代码依次断言：消息的源和目标与标签相符；没有消息被跳过（当且仅当不存在介于`a` 和`b` 之间、且当前进程是发送方或接收方的标签时，`NoBetween(a,b)` 谓词成立）；当前标签大于先前使用的标签；目标缓冲区（`u[0:1]`）可写；并且该标签位于前后两个集体操作（在本例中为屏障）之间。命令执行后，进程的`标签` 计数器会更新为该消息的标签。最后，这段代码假设（依赖–保证推理中的依赖部分）`recv` 会返回与规约中的标签相对应的消息`Msg(bt+rank-1,N)`。

第18–19 行`发送` 调用对应的代码也会先断言消息的源、目标和内容与标签相符（依赖–保证推理中的保证部分），并且源缓冲区可供读取。接下来，代码会强制执行与`接收` 调用相同的标签顺序属性。发送完成后，进程的`标签` 计数器会递增。对于非阻塞发送（例如第25–31行），这些条件会在程序执行过程中的不同时间点得到强制执行。标签的顺序属性会在相应的`等待` 调用（第31行）处强制执行，该调用负责完成发送；其余条件则会在初始化点对点通信时立即检查（第25–27行）。图中未显示的其他前置条件和后置条件确保进程不会发送重复消息，并且在调用`ISend`和`等待`之间不会写入发送缓冲区。

为了理解为什么上述条件可以避免死锁，假设出现了矛盾：死锁确实发生了，并且在所有阻塞进程中，进程 P 正阻塞在一个标签最低的 MPI 调用上。该消息的匹配进程一定尚未在其端启动与 P 的通信，否则 P 最终会解除阻塞。因此，匹配进程一定尚未执行到程序中启动与 P通信的位置。同时，这就产生了矛盾，因为匹配进程不可能阻塞在程序中的这一位置；否则，它就会是那个阻塞在

标签最低的消息上的进程。因此，匹配进程一定正在运行，所以不存在死锁。（第 3节对这一论证作了形式化说明。）

集体操作。 图 2 右侧的最后一段代码处理集体操作，对应于第 32 行的 `Barrier` 调用和第 35 行的 `Gather` 调用。代码断言没有跳过任何屏障或集体操作，并且只有在所有必须先于该屏障的消息都已处理后，才会调用屏障。相关代码片段中灰显的证明义务表明，每次调用 接收、发送 或 等待时，也会额外强制执行集体操作与点对点消息之间的顺序约束。

每次调用 `Barrier` 或 `Gather` 时，下一个集体操作的 ID 都会递增。当该 ID 达到 `LastClct(N)` 时，Dafny 可以假设 `Done()` 成立；这是一个特殊条件，表示程序已完成执行。DafnyMPI 库的使用者必须为主方法编写后置条件，以检查 `Done()` 是否为真。在我们的示例中，只有在第 35 行调用 `Gather` 后才满足这一条件。该调用还附带额外的前置条件和后置条件（图中未显示），以确保输入符合用户提供的规约，例如，`u` 包含与该进程的 秩相对应的部分输出。

这些条件共同保证了无死锁性，并使用户能够证明，对于零号进程，变量 `out` 将在经过`nt` 个时间步后包含偏微分方程的解。下一节将针对一种核心演算形式化这一方法，并证明这些条件是充分的。

3 形式化 DafnyMPI

本节将在一种核心演算上形式化 DafnyMPI，并证明：对于给定程序，如果满足 DafnyMPI 的要求，那么该程序并行运行时将无死锁。我们的核心演算在命令语言中加入了 MPI 1.0 标准 [26] 中的若干基本操作。据我们所知，后续 MPI 标准保留了这些基本操作的语义，因此我们的方法仍然适用。此外，根据我们的经验，大多数 MPI 程序的编写方式并不需要后续标准中的高级功能，例如 MPI 2.0 或 MPI 3.0 引入的单边通信和动态进程管理等额外功能。

DafnyMPI 直接编码了以下四个消息传递接口函数：

- MPI_IRECV（立即接收 或非阻塞接收）请求从给定来源接收带有指定标签的消息。该操作会立即返回，之后消息可能在后台接收。该操作返回一个 MPI_REQUEST 对象，可通过等待该对象来确保消息已接收。
- MPI_ISEND（立即发送 或非阻塞发送）请求将带有指定标签的消息发送到给定目标。该操作会立即返回一个 MPI_REQUEST 对象，可通过等待该对象来确保发送已完成。请注意，此处的完成是局部的：它并不保证消息已在目标处接收，只表示发送方可以重用与该操作关联的缓冲区。
- MPI_WAIT（等待）在 MPI_REQUEST 对象上调用时，会阻塞进程，直到相应的通信完成。
- MPI_BARRIER（屏障）会阻塞，直到所有进程也到达屏障。

在第5节中，我们讨论如何扩展核心演算，以支持DafnyMPI中可用的MPI函数，这些函数可以通过语法糖实现，或作为最初四个函数的简单变体实现。

- MPI_接收（阻塞接收）可以编码为一个MPI_非阻塞接收，随后立即执行相关的MPI_等待。
- MPI_发送（阻塞发送）可以用 MPI_ISEND 和 MPI_WAIT 类似地编码。

- MPI_GATHER 必须由所有进程调用，之后，与每个进程关联的数据会传输到指定的根进程，并存储在一个数组中。我们将此操作编码为修饰屏障调用，要求所有进程在以这种方式交换信息之前完成同步。
- MPI_ALLREDUCE 必须由所有进程调用，随后使用求和或逻辑与等二元运算对每个进程提交的数据进行归约。然后将结果传回所有进程。此操作被编码为一种装饰屏障，与 MPI_GATHER 类似。

DafnyMPI 目前不支持 MPI 通信器，通信器将进程组织在一起，例如，只有某个进程组需要通过屏障；也不支持通配符，通配符允许一个进程与同一通信器中的任意进程交换消息。我们将这些功能留待未来实现。相反，DafnyMPI 程序在所有操作中都使用单一的默认 MPI_COMM_WORLD 通信器。

3.1 核心演算

图 3 展示了核心演算的语法，该演算分为表达式和命令。由于我们的证明侧重于进程间通信，表达式 e 仅限于整数 n、变量 x、比较运算 $e = e$和基本算术运算 $e + e$、$e * e$、e **div** e、$-e$。由于不存在共享内存，每个进程都有自己的变量环境，因此一个进程无法影响另一个进程对表达式的求值。不过，表达式不能读取正由后台接收调用写入的变量（见第3.3节）。

命令 c 包括与上述消息传递接口函数相对应的四种通信原语：**irecv** e x，接收带有标签 e 的消息的非阻塞接收，其中消息写入 x；**isend** e x，发送带有标签 e 的消息的非阻塞发送，其中消息从 x 读取；**wait** e，等待带有标签 e 的发送或接收完成；以及**barrier**，在所有进程到达屏障之前阻塞。在所有情况下，标签均为整数。其余命令——**skip**、**if**、**while**、**set**以及顺序执行——具有标准语义，其中零被视为假，非零整数被视为真。

我们的语义将单个进程的状态建模为元组 c，σ，t，b，其中 c是待执行的命令，σ是将变量映射到整数的变量环境，t是此前在 **等待** 调用中使用过的最大标签，而 b 是该进程迄今已通过的屏障数。随后，包含所有进程的全局程序状态 S是一个元组 (P，B_r，B_s，B_m)，其中 P 是进程状态的集合，而 B_r、B_s和 B_m 分别是全局接收缓冲区、发送缓冲区和消息缓冲区。在本节中，我们还将使用 $N = |\mathrm{P}|$表示进程数。

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e = e \mid e + e \mid -e \\
 & \mid & e * e \mid e \textbf{ 除以 } e \\
c & ::= & \textbf{非阻塞接收 } e\ x \mid \textbf{非阻塞发送 } e\ x \\
 & \mid & \textbf{等待 } e \mid \textbf{屏障} \\
 & \mid & \textbf{跳过} \mid \textbf{如果 } e\ c\ c \\
 & \mid & \textbf{当 } e\ c \mid \textbf{设置 } x\ e \mid c\textbf{；}\ c
\end{array}
$$

图 3. 核心演算语法。

接收缓冲区和发送缓冲区是从消息标签到变量名的映射，分别表示进程环境中消息的目标（对于接收）或来源（对于发送）。消息缓冲区是从消息标签到消息负载的映射；消息负载是最终会从发送方环境中的源变量复制到接收方环境中的目标变量的值。这个中间载荷缓冲区是必要的，因为消息传递接口允许发送操作在相应的接收调用之前完成；在这种情况下，消息内容必须在内部缓冲，而发送方可以重复使用源变量。只需对缓冲区进行全局建模，因为程序员必须

提供能够唯一标识每个消息标签对应的发送方和接收方的函数，并将其作为程序规约的一部分。第3.2 节将通过一个具体示例说明如何使用这些缓冲区。

我们的操作语义是非确定性的，可以在任何未阻塞的进程中执行一步。因此，我们将正在运行的进程视为一个集合，并为方便起见采用集合表示法。具体来说，我们写作 $c,\ \sigma,\ t,\ b \cup \mathrm{P}$（省略单元素集合外的大括号），以选取一个进程 $c,\ \sigma,\ t,\ b$，其余进程记为 P；我们写作 $\mathrm{P}_1 \cup \mathrm{P}_2 \cup \ldots$，以非确定性地将进程集合拆分为若干子集。我们通过将一个整数绑定到 σ 中的 rank 来存储进程秩。作为简写，我们在进程状态上使用下标表示秩，即 $c,\ \sigma,\ t,\ b_i$表示 $\sigma(\mathrm{rank}) = i$ 的状态。秩无关紧要时，我们省略下标。

状态归约是由一系列状态构成的序列 $S_0,\ S_1,\ \ldots\ldots,\ S_k$其中每个前序状态都会根据操作语义（第3.2节）转换为下一个状态。作为另一种简写，我们有时会通过下标 $i,\ k$来指代第 k次归约步骤中第 i 个进程的状态分量，例如 $\sigma_{2,\ 0}$ 表示初始状态 S_0中秩为 2 的进程的变量环境。

DafnyMPI 程序采用单指令多数据 （SIMD）模式，所有节点都会执行相同的命令 c。在 N 个节点上执行命令 c 时，其初始状态定义如下：

$$S_0 = (\bigcup_{0 \le i < N} \langle c;\ \textbf{barrier}, \{\mathtt{rank} \mapsto i, \mathtt{size} \mapsto N\}, -1, 0\rangle, \emptyset, \emptyset, \emptyset)$$

请注意，进程初始化时会设置变量 rank 和 size，以便进程了解自身的秩以及正在运行的进程总数 N。每个进程还会初始化为执行完全相同的命令序列，其中最后一条命令必须是一个**屏障**。

我们根据用户提供的规约验证给定程序。与第 2节一样，用户必须预先描述程序的整体通信模式。具体而言，用户需提供以下五个函数；本节中我们将一直使用这些函数，它们共同描述所有 MPI 操作。请注意，这五个函数仅以与进程无关的量为参数，因此任意两个进程计算出的值始终相同：

- Sender 和 Receiver 是全函数，它们将消息标签和进程数映射到发送或接收该标签消息的进程的秩。在图 2 所示的贯穿示例中，它们分别对应函数 `Src` 和 `Dest`。
- 消息(`Msg` 见图 2) 是一个全函数，将消息标签和进程数映射到消息负载；在我们的核心演算语义中，消息负载只能是整数。
- BarrierCount (`LastClct` 见图2) 将进程数映射为进程终止前必须经过的屏障总数。
- Barrier 是一个全函数，将屏障的秩和进程数映射到消息标签，表示进程到达屏障之前，必须发送并接收所有标签小于该标签的消息。这个函数大致对应图 Clct 22中的函数，尽管 DafnyMPI 支持核心演算未明确建模的集体操作，正如我们在第5节所述。

3.2 操作语义

本节介绍我们核心演算的小步操作语义。我们的目标是尽可能贴近 MPI 标准，模拟消息传递程序的行为，从而使后续关于无死锁性的证明适用于所有可能的情况。与许多以往工作不同，我们的语义设计允许发送操作以非确定性的方式阻塞。在以往工作中（例如 [22]），发送操作通常被假定为非阻塞，这虽然简化了推理，却无法涵盖这样一种情形：

当消息负载较大时，MPI 实现很容易暂时耗尽内部缓冲区空间。为模拟这一行为，我们的语义允许消息通过以下两种方式之一从发送方传输到共享消息缓冲区 B_m： (i) 发送方执行 **等待** 调用之前，提前传输（表示正常的立即发送）；或 (ii) 延迟传输，直到接收方执行自己的 **等待**（表示会阻塞的立即发送）。本节余下部分将介绍这些规则。

图 4 给出了小步操作语义，分为两组：（a）作用于单个进程的局部规则，以及（b）还会同时操作共享缓冲区或多个进程的全局规则。我们的语义还包括大步表达式求值，$\langle e, \sigma\rangle \Downarrow v$，表示在变量环境 e 中对表达式求值 σ 可得值 v。由于表达式求值完全遵循标准做法，我们省略这些规则。

图 4a所示的局部规则定义了进程状态上的单步关系 $\rightsquigarrow c,\ \sigma,\ t,\ b\rangle$，其含义是：对左侧状态所对应进程执行一步，便得到右侧状态。IfTrue 和IfFalse 涵盖了条件语句的两种情况，而While将循环展开一步。最后，Set 会相应更新变量环境，而SeqSkip 会移除序列开头的**跳过**。命令仅为**跳过**的进程被视为已终止，因此没有规则可以进一步对其求值。

图 4b所示的全局规则定义了程序状态上的单步关系 $\longrightarrow (\mathcal{P},\ \mathcal{B}_r,\ \mathcal{B}_s,\ \mathcal{B}_m)$。重要的是，这些规则省略了一些通常会在运行时检查的前提，但这些前提由 Dafny 验证器保证成立。例如，发送 规则不会验证发送方的秩是否与标签指定的发送方匹配，因为使用 DafnyMPI 时会检查这一点。我们在下文第 *3.3* 节将这些前置条件形式化为公理。

前两条归约规则是管理性规则：Proc 将单个进程的归约提升为整个程序状态的归约，而 SeqStep 则对序列中的第一个命令执行一步。请注意，这两条规则都使用集合表示法来非确定性地选取一个进程。

其余大多数规则描述发送操作，而主要挑战在于其非确定性行为。根据实现和消息大小，MPI 运行时可能会将有效载荷复制到内部缓冲区，从而使发送方后续的**等待**立即完成。或者，进程可能会阻塞，直到调用匹配的接收操作。我们的操作语义旨在模拟这两种行为。

我们使用图 5 中的示例来说明所述的通信规则。图中展示了一个简单程序的两种可能的部分归约序列。在初始状态（图中的状态 0）下，秩 0 的进程即将发起一条带有标签 t的消息的非阻塞发送，而秩 1 的进程即将发起对同一消息的非阻塞接收。这两条命令中的任意一条都可以先执行，分别进入状态 1 或状态 6。

Send 规则会添加一个发送 x 作为消息的请求，并使用相应的标签 t'。该请求通过将 x 放入发送缓冲区 B_s 中进行登记。同样，接收会添加一个带标签 t' 接收 x 的请求。如上所述，这些操作都是非阻塞的。在我们的示例中，我们用 S（发送）和 R（接收）标记状态转换，并分别看到 B_s 和 B_r 的相应更新。

沿图的上方继续看，从状态 1 到状态 2 的转换展示了 Trans- ferNoWait：给定一个未处于等待状态或屏障处的发送进程（回想一下，c_i 表示第 i 个进程的命令），该规则会将待发送消息的内容从 x 复制到 B_m，并附上相应的标签 t'。在此示例中，消息负载为 5，因此 B_m 会更新为将 t 映射到 5。这种行为描述了 MPI 实现将消息负载复制到缓冲区的情形，使发送进程能够在匹配的接收甚至尚未发起之前解除阻塞。

消息复制到消息负载缓冲区后，秩 0 的进程即可完成发送，这由通过 WaitSend 从状态 2 转换到状态 3 表示。该规则会检查带有标签 t' 的消息是否已放入 B_m，然后从中移除该标签

$$\frac{\langle e, \sigma\rangle \Downarrow v \quad v \neq 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_1, \sigma, t, b\rangle}\ \textsc{IfTrue} \qquad \frac{\langle e, \sigma\rangle \Downarrow 0}{\langle \textbf{if } e\ c_1\ c_2, \sigma, t, b\rangle \rightsquigarrow \langle c_2, \sigma, t, b\rangle}\ \textsc{IfFalse}$$

$$\frac{}{\langle \textbf{while } e\ c, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{if } e\ (c;\ \textbf{while } e\ c)\ \textbf{skip}, \sigma, t, b\rangle}\ \textsc{While}$$

$$\frac{\langle e, \sigma\rangle \Downarrow v}{\langle \textbf{set } x\ e, \sigma, t, b\rangle \rightsquigarrow \langle \textbf{skip}, \sigma[x \mapsto v], t, b\rangle}\ \textsc{Set} \qquad \frac{}{\langle \textbf{skip};\ c, \sigma, t, b\rangle \rightsquigarrow \langle c, \sigma, t, b\rangle}\ \textsc{SeqSkip}$$

(a) Single-Process Operational Semantics Rules.

$$\frac{\langle c, \sigma, t, b\rangle \rightsquigarrow \langle c', \sigma', t', b'\rangle}{(\langle c, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Proc}$$

$$\frac{(\langle c_1, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c', \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}{(\langle c_1; c_2, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle c'; c_2, \sigma', t', b'\rangle \cup \mathcal{P}, \mathcal{B}'_r, \mathcal{B}'_s, \mathcal{B}'_m)}\ \textsc{SeqStep}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{isend } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s[t' \mapsto x], \mathcal{B}_m)}\ \textsc{Send}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t'}{(\langle \textbf{irecv } e\ x, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t, b\rangle \cup \mathcal{P}, \mathcal{B}_r[t' \mapsto x], \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Recv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_r \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma[\mathcal{B}_r(t') \mapsto \mathcal{B}_m(t')], t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r \setminus \{t'\}, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{WaitRecv}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad t' \in \mathcal{B}_m}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{skip}, \sigma, t', b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s \setminus \{t'\}, \mathcal{B}_m)}\ \textsc{WaitSend}$$

$$\frac{\langle e, \sigma\rangle \Downarrow t' \quad \textsc{Receiver}(t', N) = i \quad t' \in \mathcal{B}_s \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_{\textsc{Sender}(t', N)}\rangle \Downarrow v}{(\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\langle \textbf{wait } e, \sigma, t, b\rangle_i \cup \mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferOnWait}$$

$$\frac{t' \in \mathcal{B}_s \quad i = \textsc{Sender}(t', N) \quad c_i \neq \textbf{wait } e; c' \quad c_i \neq \textbf{barrier}; c' \quad \mathcal{B}_s(t') = x \quad \langle x, \sigma_i\rangle \Downarrow v}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m[t' \mapsto v])}\ \textsc{TransferNoWait}$$

$$\frac{i = \textsc{Sender}(t', N) \quad j = \textsc{Receiver}(t', N) \quad t_i \geq t' \quad t_j \geq t' \quad t' \in \mathcal{B}_m}{(\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\mathcal{P}, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m \setminus \{t'\})}\ \textsc{FreeBuffer}$$

$$\frac{}{(\bigcup_{0 \leq i < N} \langle \textbf{barrier}; c, \sigma, t, b\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m) \longrightarrow (\bigcup_{0 \leq i < N} \langle \textbf{skip}; c, \sigma, t, b+1\rangle_i, \mathcal{B}_r, \mathcal{B}_s, \mathcal{B}_m)}\ \textsc{Barrier}$$

(b) Multi-Process Operational Semantics Rules.

Fig. 4. Core calculus reduction rules

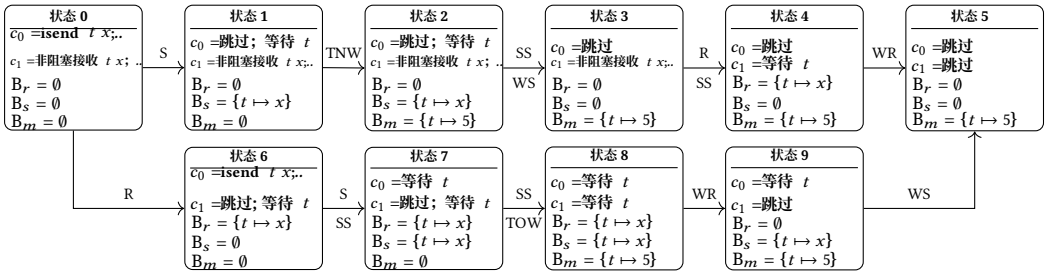

图5。程序 **set** x 5;**if** 秩 = 0 **{ isend** t x **} { 非阻塞接收** t x **}; 等待** t 的两种部分归约。该程序由两个进程执行；边上标注的是归约规则的缩写（SS 代表 SeqSkip，等等）。先应用横线上方的归约规则。

待处理发送的集合。在我们的示例中，可以看到，由于之前执行了 TransferNoWait，该规则可以触发，因此标签 t 会从 B_s 中移除。

从状态 3 到状态 4 的转换使用 Recv 添加一个请求，以接收带有标签 t的消息；随后，WaitRecv 可以触发，使状态从状态 4 转换到状态 5。与 WaitSend 类似，当消息缓冲区中存在带有给定标签的消息时，WaitRecv 可以执行，并将要写入的变量更新为消息负载，同时从 B_r 中移除该标签。虽然图中未显示，x 已在 c_1 中更新为 5。

相比之下，底部一行展示了 MPI 发送会一直阻塞，直到执行匹配的接收操作的情形。具体来说，应用发送规则从状态 6 转换到状态 7 后，秩 0 的进程无法继续执行。由于消息负载缓冲区 B_m 为空，WaitSend 规则无法触发；而由于该进程已因等待而阻塞，TransferNoWait 规则也无法触发。因此，唯一能够解除秩 0 进程阻塞的规则是 TransferOnWait；当秩 1 的接收进程到达对应的等待调用时，该规则即可触发。图中通过从状态 7 到状态 8 的转换展示了这一过程：秩 1 的进程先通过顺序步进规则到达**等待**调用，然后通过 TransferOnWait 规则启动传输。随后，WaitRecv 将系统从状态 8 转换到状态 9，而 WaitSend 则通过推进到状态 5 完成通信。

请注意，在图5中的最终状态 5 下，消息缓冲区$\mathcal{B}_m$ 仍包含带有标签 t的消息。与该标签关联的通信完成后，根据FreeBuffer 规则，可以随时将其从FreeBuffer m 中移除。该规则模拟了 MPI 运行时的预期行为，即 MPI 内部使用的内存最终应当释放。具体时机取决于实现，与我们的证明无关。

我们操作语义中的最后一条规则「屏障」规定，所有进程必须同时执行屏障。请注意，我们稍微宽松地使用了符号：允许屏障规则应用于最终屏障，尽管该规则在形式上预设后续还存在一条命令。

3.3 公理

我们的核心演算表示 DafnyMPI 用户可以编写的程序。要让 Dafny 验证这些程序，用户必须证明若干性质。这些性质包括 Dafny 默认强制检查的性质，例如循环终止、不存在除以零的情况，以及不引用未声明变量。此外，还有一些与消息传递接口相关的性质，它们以相关 DafnyMPI 方法的前置条件形式表达。我们将这些性质称为公理，因为在构建

表 1。 核心演算的公理（除 AtStart 外，均隐式以程序状态作为参数）。

公理	前提	结论
AtStart		$S_0 = (\bigcup_{0 \le i < N} c;\ \textbf{屏障},\ \{\text{秩} \mapsto i,\ \text{大小} \mapsto N\} - 1,\ 0\ \emptyset,\ \emptyset,\ \emptyset)$
AtSend	$c_{i,k} = \textbf{isend}\ e\ x;\ c$ $e,\ \sigma_{i\ k} \Downarrow v$,	$i = \text{发送方}(v, N) \quad x, \sigma_{i,k} \Downarrow \text{消息}(v) \quad v \notin \text{Bs}_{,k}$
接收点	$c_{i,k} = \textbf{非阻塞接收}\ e\ x;\ c$ $e, \sigma_{i,}{}^{k} \Downarrow v$	$i = \text{接收方}(v, N) \quad v \notin \text{Br}_{,k}$
AtWait	$c_{i,k} = \textbf{等待}\ e;\ c$ $e,\ \sigma_{i\ k} \Downarrow v$,	$v > t_{i,k}$ $\forall v'.\ t_{i,k} < v' < v \Rightarrow (\text{发送方}(v',\ N) \neq i \wedge \text{接收方}(v',\ N) \neq i)$ $\text{屏障}(b_{i,\ k},\ N) \le v < \text{屏障}(b_{i,\ k} + 1,\ N)$ $(i = \text{发送方}(v,\ N) \wedge v \in \text{Bs}_{,\ k}) \vee (i = \text{接收方}(v,\ N) \wedge v \in \text{Br}_{,\ k})$
AtBarrier	$c_{i,k} = \textbf{屏障};\ c$ $v = \text{屏障}(b_{i,\ k} + 1,\ N)$	$v > t_{i,k}$ $\forall v'.\ t_{i,\ k} < v' < v \Rightarrow (\text{发送方}(v',\ N)\ \text{接收方}(v',\ N) \neq i)$ $\forall v.\ \text{发送方}(v, N) = i \Rightarrow v \notin \text{Bs}_{,k}$ $\forall v.\ \text{接收方}(v, N) = i \Rightarrow v \notin \text{Br}_{,k}$
结束	$c_{i,k} = \textbf{跳过}$	$b_{i,\ k} = \text{BarrierCount}(N)$
AtSet	$c_{i,}{}^{k} = \textbf{集合}\ x\ e;\ c$	$\forall v.\ \text{发送方}(v,\ N) = i \Rightarrow (v \notin \text{Bs}_{,\ k} \vee x \neq \text{Bs}_{,\ k}(v))$ $\forall v.\ \text{接收方}(v,\ N) = i \Rightarrow (v \notin \text{Br}_{,\ k} \vee x \neq \text{Br}_{,\ k}(v))$ $x \neq \text{秩}$
AtRead	$\text{接收方}(v,\ N) = i$ $\text{Br}_{,\ k}(v) = x$	$x \notin$ 直接子表达式（属于）$c_{i,\ k}$

我们的证明时，可以假设它们对任何执行都成立，因为这些性质由 Dafny 验证器完成验证。除 AtStart 以外，每条公理都隐式地以某个特定程序状态为参数：对于从初始状态可达的任意状态，如果公理的前提成立，那么其结论也成立。

需要注意的是，Dafny 只对单个（任意）进程的行为建模，并且只能检查该进程有权访问的程序状态部分。因此，公理中都不会提及消息缓冲区 $\mathcal{B}_m$，因为其内容属于 MPI 运行时内部状态。同样，只有当公理所适用的进程是预期的发送方或接收方时，公理才会检查给定标签是否存在于发送或接收缓冲区中。此外，由于 Dafny 验证器假定所有方法调用都会终止，因此它本身无法考虑发生死锁的可能性。DafnyMPI 在此基础上明确地证明无死锁性。同时，Dafny 的非阻塞假设保证公理在该假设下可达的任何状态中都成立。这使我们能够通过考察必然会到达的未来状态中必须成立的条件，来推理程序状态。具体而言，由于 Dafny 证明了任意进程的最终屏障计数等于 BarrierCount(N)，我们可以证明屏障计数永远不会超过该值；这是第 3.5 节引入的引理之一。

还应结合上文给出的归约规则来理解这些公理。例如，我们说表1 中的 AtWait 公理保证标签单调递增，但只有将该公理与图4中的 WaitRecv 和 WaitSend 归约规则结合起来考虑时，这一说法才成立。具体而言，这两条归约规则总是将新的标签$t_{i,\ k+1}$ 设为与**等待**调用的参数相同；而 AtWait 公理保证该参数大于先前的标签$t_{i,\ k}$。

表1 列出了我们语言的公理。 AtStart 公理描述了程序的初始状态，如上文第 3.1 节所述。AtSend 和 AtRead 规定，带有给定标签的消息只能由指定的 发送方 和 接收方发送或接收。这两条公理还要求相应的缓冲区为空，以禁止在调用等待之前重复发送或接收同一

表 2。死锁及相关术语的定义。

Term	条件
OnSend((**等待** e; c', σ, t, $b_i \cup \mathrm{P}$, B_r, B_s, B_m))	$e, \sigma \Downarrow t'$ $i =$ 发送方(t', N) $t' \notin \mathrm{B}_m$
接收操作((**等待** e; c', σ, t, $b_i \cup \mathrm{P}$, B_r, B_s, B_m))	$e, \sigma \Downarrow t'$ $i =$ 接收方(t', N) $t' \notin \mathrm{B}_m$ $t' \notin \mathrm{B}_s$
障碍调用(**屏障**; c', σ, t, b)	
已终止(**跳过** σ, t, b)	
死锁(($\mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3 \cup \mathrm{P}_4$, $\mathcal{B}_r$, $\mathcal{B}_s$, $\mathcal{B}_m$))	$\lvert\mathrm{P}_3\rvert \neq N$ $\lvert\mathrm{P}_4\rvert \neq N$ 接收$(p, \mathrm{B}_r, \mathrm{B}_s, \mathrm{B}_m)$ $\forall p \in \mathrm{P}_1$ 发送$(p, \mathrm{B}_r, \mathrm{B}_s, \mathrm{B}_m)$ $\forall p \in \mathrm{P}_2$ 障碍调用(p) $\forall p \in \mathrm{P}_3$ 已终止(p) $\forall p \in \mathrm{P}_4$

消息。调用 **等待**。此外，AtSend 还保证消息负载符合 Message函数的规定。

AtWait 公理强制实施标签顺序性质。首先，它保证每个进程使用 **等待** 时，消息标签单调递增。其次，它确保不会跳过任何**等待** 调用——如果某进程是某条消息的指定接收方或发送方，则在转而处理标签更高的消息之前，必须先接收或发送该消息。第三，该公理确保连续的**等待** 调用之间不会跳过任何屏障。最后，它确保只有在相应的 **isend** 或**非阻塞接收**之后才调用**等待** 。

AtBarrier 要求各进程按顺序调用屏障，并等待所有应在屏障之前完成的发送和接收操作。与上述屏障规则类似，AtBarrier 公理也应适用于最终屏障。该公理还保证在屏障处发送缓冲区和接收缓冲区均为空，即此时不能有正在进行的发送或接收调用。由于每个进程执行的最后一条命令都是屏障，这项要求保证每个**非阻塞接收** 和**isend** 最终都会有相匹配的**等待**调用。在每个进程执行结束时，结束公理指出，该进程恰好通过了 BarrierCount(N) 个屏障。

最后，AtSet 和AtRead通过禁止读写用于进程间通信的变量，防止非确定性行为。我们将在第 4 节讨论如何安全使用读写缓冲区，并说明 Dafny 在实际中如何强制执行这两个公理。AtSet 还会阻止对变量 秩进行更改，这意味着我们可以确信它始终等于操作语义中该进程的实际秩。

3.4 死锁

表2介绍了用于形式化定义和推理死锁的术语。我们首先列出判定进程处于阻塞状态的条件：

- 如果一个进程即将执行对其发送的消息的**等待**，但该消息尚未放入载荷缓冲区，则该进程在发送操作上处于阻塞状态。发送方将一直处于阻塞状态，直到接收方执行相应的**等待**。（表中的 OnSend 谓词描述了这一条件。）
- 如果一个进程即将执行接收其消息的**等待**命令，但消息负载缓冲区和发送缓冲区中都没有相关标签，则该进程在接收操作上处于阻塞状态。这意味着发送方尚未启动通信。（表中的 OnRecv 谓词定义了这一条件。）
- 如果某个进程即将执行表中 OnBarrier 谓词所定义的 **障碍**命令，则它可能会被阻塞在屏障处。只要其他进程尚未全部到达屏障，该进程就会一直处于阻塞状态。

有了这些定义后，我们现在可以形式化地定义死锁。死锁是指除所有进程均已终止的状态之外，任何不适用归约规则的状态。请注意，我们将 FreeBuffer 规则排除在这一条件之外，因为任何仅适用 FreeBuffer 的状态最终都会导致死锁。可以看出，死锁的这一定义等价于每个进程都处于阻塞或已终止状态，但所有进程均已终止的情况除外。表 2 中 Deadlock 的定义对此进行了描述。处于死锁状态的所有进程必须属于以下四组之一：等待接收或发送操作完成时被阻塞的进程（表中的 P_1 和 P_2）、被阻塞在屏障处的进程（P_3），以及已经终止的进程（P_4）。此外，不可能所有进程都属于 P_4（因为这样程序就会已经终止），也不可能所有进程都属于 P_3（因为这样进程就会通过屏障）。请注意，OnRecv 和 OnSend 谓词明确排除了可以应用 WaitSend、WaitRecv、TransferOnWait 或 TransferNoWait 的状态。死锁形式化定义后，我们现在继续证明无死锁性。

3.5 无死锁性证明

本节概述了证明核心演算不会发生死锁的过程。证明首先选出等待最小标签消息的进程，并说明其通信伙伴不会被阻塞，因为消息标签的排序只允许进程等待更小的标签。主要定理依赖若干辅助引理，为简洁起见，我们在此仅列出而不予证明。这些引理和定理均假设第3.3 节所述公理成立，其中包括对初始程序状态的描述。完整证明见补充材料。

引理 1 (NoTwoTagsTheSame)。 如果两个进程在死锁状态下即将执行**等待** 命令，则相应的消息标签必须不同。

引理 2 (NonEmptyBuffer)。 如果某个进程负责发送或接收标签为 $v \in \mathbb{N}$的消息，并且该进程已使用的最大标签值最终达到 v，那么在执行过程中的某个时刻，消息缓冲区必定包含带有标签 v 的有效载荷。

引理 3 (BufferStaysNonEmpty)。 如果进程是载荷缓冲区中某条消息的预期接收方或发送方，但该消息的标签高于该进程迄今为止接收或发送的任何消息的标签，那么该消息必须留在缓冲区中。

引理 4 (NumberOfBarriers)。 如果进程的命令序列末尾有一个屏障，那么其内部屏障计数器严格小于 *BarrierCount(N)*。

定理 1 (无死锁性)。 不存在从初始状态 S_0 到死锁状态的归约。

证明。反设存在归约 $S_0 \longrightarrow \cdots \longrightarrow S_k$，且 Deadlock($S_k$)。根据死锁的定义，必然存在某个 $\mathrm{P}_1, \mathrm{P}_2, \mathrm{P}_3, \mathrm{P}_4, \mathrm{B}s$、$\mathrm{B}r$ 和 $\mathrm{B}m$，使得 $S_k = (\mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3 \cup \mathrm{P}_4, \mathrm{B}s, \mathrm{B}r, \mathrm{B}m)$，并且 $\mathrm{P}_1, \mathrm{P}_2, \mathrm{P}_3, \mathrm{P}_4$ 具有表 2 中描述的性质。分两种情况：$\mathrm{P}_1 \cup \mathrm{P}_2 = \emptyset$ 或 $\mathrm{P}_1 \cup \mathrm{P}_2 \neq \emptyset$。

- **情况 1**。假设 $\mathrm{P}_1 \cup \mathrm{P}_2 = \emptyset$。这意味着所有仍在运行的进程都在等待某个屏障。根据死锁的定义，$|\mathrm{P}_3| \neq N$。换句话说，至少有一个进程已经终止。设该进程为 p_m 秩为 m。由于该进程已经终止，AtEnd 公理适用，这意味着 $b_{m,\ k} = \mathrm{BarrierCount}(N)$。注意，所有进程必须始终共享相同的 b，因为所有进程同时通过屏障，并且根据 AtStart 公理，所有进程均以 $b = 0$ 开始。然而，我们知道至少有一个进程（称其为 $p_i \in \mathrm{P}_3$）尚未终止。根据

NumberOfBarriers引理，我们知道 $b_{i,\ k} < \text{BarrierCount}(N)$。然而，我们也知道 $b_{i,\ k} = b_{m,\ k} = \text{BarrierCount}(N)$。因此，我们得出了矛盾。所以，$\mathrm{P}_1 \cup \mathrm{P}_2 \neq \emptyset$。

- **情况 2**。假设 $\mathrm{P}_1 \cup \mathrm{P}_2 \neq \emptyset$。这意味着至少有一个进程被阻塞在某个**等待**调用上，而该进程是该调用的接收方或发送方。令 m 表示正在等待标签最小的消息的进程的秩，即 $c_{m,\ k} =$ **等待** e_m**;** c' 且 e_m，$\sigma_{m,\ k} \Downarrow v_m$，其中 v_m 小于 $\mathrm{P}_1 \cup \mathrm{P}_2$ 中任何其他进程对应的值。根据 NoTwoTagsTheSame 引理，任意两个进程都不能等待同一个标签，因此这样的最小值一定存在。现在我们假设 $p_m \in \mathrm{P}_1$（当 $p_m \in \mathrm{P}_2$ 时，推理过程对称）。由于 $p_m \in \mathrm{P}_1$，我们知道 $m = \text{Receiver}(v_m,\ N)$。

  现在考虑秩为 $i = \text{Sender}(v_m,\ N)$ 的进程。接下来我们将证明 $t_{i,\ k} \geq v_m$。我们知道 $p_{i,\ k} \in \mathrm{P}_1 \cup \mathrm{P}_2 \cup \mathrm{P}_3 \cup \mathrm{P}_4$。因此有以下三种子情况：

  – 子情形 1：$p_i \in \mathrm{P}_1 \cup \mathrm{P}_2$。那么 $c_{i,\ k} =$ **等待** e_i**;** c' 并且 e_i，$\sigma_{i,\ k} \Downarrow v_i$。根据构造，我们知道 $v_i > v_m$。根据 AtWait 公理，我们还知道 $v_i > t_{i,\ k}$。根据同一公理，必然有 $t_{i,\ k} \geq v_m$，因为在 v_i 和 $t_{i,\ k}$ 之间不能有任何标签，使得 i 是发送方。

  – 子情形 2：$p_i \in \mathrm{P}_3$。将 AtBarrier 公理应用于状态为 k 的进程 i，可得 $\text{Barrier}(b_{i,k} + 1, N) > t_{i,k}$。将 AtWait 公理应用于状态为 k 的进程 m，可得 $v_m < \text{Barrier}(b_{m,k} + 1, N)$。如上所述，在任何给定状态下，所有进程的 b 值都相同。因此，我们有 $b_{i,k} = b_{m,k}$ 和 $v_m < \text{Barrier}(b_{i,k} + 1, N)$。将 AtBarrier 公理应用于状态为 k 的进程 i，则在 $t_{i,k}$ 和 $\text{Barrier}(b_{i,k} + 1, N)$ 之间不能有任何标签，使得 i 是发送方，因此，$t_{i,k} \geq v_m$。

  – 子情况 3：$p_i \in \mathrm{P}_4$，这意味着 p_i 已终止。注意，根据 AtStart 公理，p_m 的命令序列末尾有一条**屏障**命令。由于 p_m 尚未通过此屏障，因此 NumberOfBarriers 引理适用，并且 $b_{m,\ k} < \text{BarrierCount}(N)$。与此同时，AtEnd 公理适用于秩为 i 的进程，因此 $b_{i,\ k} = \text{BarrierCount}(N)$。然而，这会导致矛盾，因为如上所述，在任何给定状态下，所有进程的 b 值都相同，所以 $b_{i,\ k} = b_{m,\ k}$。因此，子情况 3 不可能成立。

  我们现在已经证明 $t_{i,\ k} \geq v_m$。另请注意，根据 AtWait 公理，$v_m > t_{m,\ k}$。由于 $t_{i,\ k} \geq v_m > t_{m,\ k}$，我们可以对进程 i 使用 NonEmptyBuffer 引理，并对进程 m 使用 BufferStaysNonEmpty 引理。由此可得 $v_m \in B_{m,\ k}$。然而，这是一个矛盾，因为死锁谓词要求 $v_m \notin B_{m,\ k}$。因此，$\mathrm{P}_1 \cup \mathrm{P}_2 = \emptyset$。

我们现在已经证明，这两种情况都会导致矛盾。因此，最初的假设必定是错误的，死锁不可达。□

上述证明将 MPI 标准所描述的行为（形式化为核心演算的操作语义）与我们在 Dafny 中编写的规约（形式化为核心演算的公理）联系起来，从而证明这些规约能够防止使用 DafnyMPI 的程序发生死锁。要证明程序完全正确性，唯一还需要进行的进程间推理涉及消息的内容，我们将在下文讨论。

4 证明函数等价性

我们的方法保证使用 DafnyMPI 编写的程序永远不会发生死锁，但其本身并不能防范领域特定错误。与此同时，软件工程师在编写并发代码时犯下此类错误并不罕见。因此，为了进一步确保 DafnyMPI 程序的正确性，我们提出了一种方法，用于建立使用 DafnyMPI 编写的并行程序与使用纯

Dafny 编写的顺序程序之间的功能等价性。我们采用该方法证明第 6 节所述全部三个基准测试的功能正确性。具体而言，我们证明零号进程——按照惯例，它负责在最后汇总所有进程的结果——产生的结果与顺序版本相同。该方法包括以下关键步骤：

- 首先，程序员使用 Dafny 编写所需并行化计算的顺序版本。顺序代码通常更容易理解，从而降低此阶段出现领域特定错误的可能性。
- 接下来，程序员为程序编写幽灵函数规范，并证明对于所有输入，顺序版本返回的结果都与函数规范相同。函数规范通常更难理解，因为所有循环都必须替换为递归。这个证明相对直接，因为这正是 Dafny 的常见用途（例如，参见 [5]）。我们将在第5节讨论一些证明工程方面的考量，尤其是如何应对证明脆弱性。
- 接下来，程序员确定并行版本的拓扑结构，并编写相应的通信幽灵规约，如图2中的第一段 Dafny代码所示。此规约可以调用先前编写的函数规范，以描述所有消息的载荷。
- 最后，程序员编写程序的并行版本，同时确保：(i) 零号进程返回的结果与函数规范相同（因此根据传递性，它与顺序实现等价）；(ii) 程序遵循拓扑结构规约（由 DafnyMPI 强制执行）。这通常是整个过程中最困难的部分。在我们的证明中，我们采用自顶向下的方法，先勾勒主程序循环，并假设底层方法的行为符合预期，然后再证明这些底层方法的正确性。

为了推理 MPI 程序中某个进程的返回值，并由此确立函数等价性，我们必须推理各个 MPI 操作返回的值。DafnyMPI 通过确立两项性质来支持此类推理。首先，DafnyMPI 使用依赖—保证推理，证明任意进程接收的消息都与用户指定的完全一致。其次，DafnyMPI 强制要求安全使用缓冲区，以防止两个接收操作同时写入同一缓冲区。综合起来，这些性质保证在相应读操作终止时，写缓冲区的内容恰好与用户指定的一致。下面我们将更详细地讨论这两项性质。

依赖-保证推理。 要让 Dafny 验证有关每个进程计算所得值的性质，它必须知道这些进程接收的消息内容。显然，消息内容始终会严格遵循 Message 函数（在第 3.1 节中介绍）的规定。具体而言，AtSend 公理保证消息来源变量符合 Message 函数的规定，而 AtSet 和 AtRead 公理则确保在通信完成之前，该进程或其他消息传递接口操作不会写入此变量。由于 Message 函数是消息标签和进程数的纯函数，任意两个进程对具有相同标签的消息内容总会达成一致。更正式地说，这就是依赖-保证推理的一个例子。Dafny 可以依赖这样的保证：某个进程接收到的每条消息都符合用户提供的规约，因为它保证了所有曾发送的消息也都符合该规约。在 Dafny代码中，这是通过每个 **等待** 调用的后置条件来体现的，该后置条件描述了接收到的消息。

安全使用读写缓冲区。我们的核心演算通过 AtSet 公理阻止写入用于 MPI 通信的变量，并通过 AtRead 公理阻止使用 MPI 操作

可能正在写入的变量。这些限制是必要的，因为在存在非阻塞操作时，进程可能与其发起的 MPI 操作并发执行。要在 Dafny 中落实这些保证，我们还必须考虑别名，以及真实的 MPI 实现通过缓冲区（即按引用共享的数据数组）进行通信这一事实。实际上，在相应操作完成之前，MPI 库可能随时读取指定缓冲区区域的内容，或向其中写入。发送命令可以使用重叠缓冲区，而且通常确实如此，但每个接收命令都必须独占相关缓冲区区域。此外，进程不能写入当前正被任何 MPI 操作使用的缓冲区，也不能读取正被 MPI 调用写入的缓冲区。

在 Dafny 中，可以强制正确使用缓冲区，因为 Dafny 抽象掉了直接内存访问，这意味着访问缓冲区的唯一方式是调用相关对象的实例方法。为支持缓冲区，我们引入了两个自定义类，分别用于表示浮点数一维数组和二维数组。这两个类实现了一个共享接口（在 Dafny 中也称为 特质 ），用于对内存连续性以及作为幽灵状态一部分的逐元素读写锁进行建模。对于每个数组元素，对象的幽灵状态会跟踪该元素当前是否正被接收操作写入，或正被（可能多个）发送操作读取。写锁充当标志：当数组的一部分被锁定以供写入时，在锁释放之前，该区域既不能读取也不能写入。相反，读锁充当计数器：多个 MPI 操作可以并发读取相同元素，但只要仍有至少一个操作在读取这些元素，就不允许写入。每个发送或接收命令都以用作缓冲区的数组引用，以及消息的起始索引和大小作为输入。相应 MPI 调用的前置条件确保指定的数组元素可供读取或写入。如果 MPI 调用是非阻塞的，后置条件便会将数组中的相关元素标记为锁定状态，直到操作完成。对于二维数组，我们对 C 风格连续存储进行建模（正如 5 中所讨论的，这由 Python 的 Numpy 库在运行时保证），这意味着一行或多行在内存中是连续的，但列并非如此，除非数组只有一列。

对消息负载的依赖—保证推理，以及用于控制缓冲区访问的读写锁机制，共同保证接收到的消息内容与用户规约完全一致，从而能够证明函数等价性。

5 实现

DafnyMPI 由两部分组成。第一部分是对消息传递接口原语的规约，它们是 MPI.World 类的实例方法。这些规约连同少量辅助引理和函数，共包含 379 行 Dafny代码，构成该库的信任基。此文件不会直接编译为 Python；相反，每个已规约的消息传递接口原语都由一个 Python 包装函数实现（总计约 60 行代码），这些函数与该库一并提供。

DafnyMPI 的第二部分定义了两个类，用于操作实数的一维和二维数组，并且能够跟踪哪些元素当前正被消息传递接口使用（见上文）。为最大限度地确保可移植性，所有数组操作（如滚动操作、切片、标量和逐元素乘法与加法等）均在 Dafny 中实现，且仅依赖对构造函数以及三个基本 `Get`、`Set` 和 `Size` 方法的调用。库的这一部分共计 1772 行 Dafny代码，除了上述构造函数和三个基本方法外，均可直接编译为 Python（或 Dafny 支持的任何其他目标语言）。这几个方法均使用 Numpy 库以 Python 实现，相关代码与 DafnyMPI 库一并提供。Numpy 的默认数组构造函数还可确保数组元素在内存中采用 C 风格连续存储，这是我们的 Dafny代码在构造数组时所假定、并在后续操作中始终保持的属性。实际上，为优化性能，可以在运行时将所有数组操作委托给 Numpy，但我们选择尽量缩小信任基。

接下来，我们将讨论一些相关的实现挑战。

处理标签溢出。 第 3.5 节中无死锁性的证明假设 MPI 标签没有上限。实际上，不同的 MPI 实现对标签大小设定了不同的限制，例如，Intel MPI库使用 20 位表示标签 [35]。MPI 标准本身唯一的要求是，标签的上界不得小于 32767 [26]。

为确保标签始终有效，DafnyMPI 增加了一项证明义务，要求我们在第 屏障 3.1 节引入的函数 每次增加不超过 32767。这确保任意两个屏障之间使用的 标签最多为 32767 个。因此，即使发生标签溢出，屏障之间的标签仍保持唯一，使进程能够确定性地识别消息。

支持阻塞式发送和接收。我们的核心演算仅对非阻塞式发送和接收操作建模。如前所述，我们可以轻松地将阻塞式变体编码为紧接着调用 **isend** 或**非阻塞接收** 后立即调用**等待**。DafnyMPI 库在验证时通过这种方式表达阻塞式发送和接收，而在运行时使用相应的 MPI_SEND 和 MPI_RECV 命令。支持阻塞式发送和接收可以简化代码：当非阻塞通信没有额外优势时，无需管理 MPI_REQUEST 对象。

支持集体操作。消息传递接口标准描述了多种一对多、多对一和多对多的集体操作，例如 MPI_SCATTER、MPI_GATHER 和 MPI_ALLREDUCE。核心演算不包含这些功能，但 DafnyMPI 通过将后两种操作视为带注释的屏障来支持它们。具体而言，DafnyMPI 对集体操作施加了限制，要求只有在后台没有待处理的发送或接收操作运行时才能使用这些操作。运行时，DafnyMPI 会在任何此类集体操作之前显式设置屏障。因此，新增这些集体操作不会引入死锁。用户在初始化消息传递接口时会提供推理某个操作输出所需的任何信息，这与指定一对一消息的方式类似。

例如，要描述运行示例第35 行的 `Gather` 操作，用户必须指定最后一个`nt`th 屏障是一个 `Gather` 操作，并提供所收集信息的精确函数规范（为简洁起见，图 2 中未展示）。调用 `Gather`时，除了满足常规屏障所需的条件外，用户还必须证明每个进程发送的数据与所指定结果中对应的部分相匹配，并且相关缓冲区可读写。随后，`Gather`的规约保证根进程接收到的数据与上述指定内容完全一致。

终止性。 在第2.3节列出的所有正确性属性中，证明终止性所需的工作最少。一旦确立了无死锁性，并且验证器成功排除了数组越界访问等运行时错误，循环和递归便是终止性仅剩的障碍。按照 Dafny 中的标准做法，我们通过指定一个每次迭代都会递减且下界为零的正量，来证明循环和递归会终止。对于图2中贯穿全文的示例里的外层 for 循环，该量为 `nt-n`，也就是剩余迭代次数。

证明工程方面的考量。 自动化验证中常见的一个问题是脆弱性：提供给验证器的信息越多，验证查询就越有可能超时。为解决这个问题，我们采用 Dafny 工程师此前提出的方法 [7, 25]，将大多数函数——尤其是涉及量词的函数——设为不透明，除非明确揭示其定义，否则不让验证器看到这些定义。对于某些此类函数，我们还会

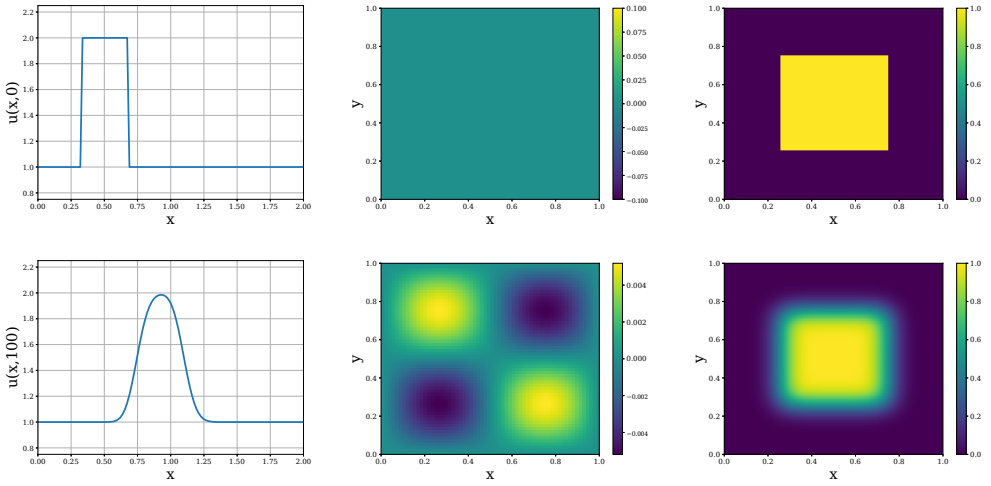

图 6. 偏微分方程模拟基准测试。上排：初始条件；下排：DafnyMPI 计算的 100 个时间步后的解。从左至右：线性对流、泊松、热扩散

使用专门的引理来指定其行为，而不是使用后置条件；这同样能让用户更好地控制验证器在任意时刻可以访问哪些事实。我们建议 DafnyMPI 的使用者也采用相同的方法，以避免证明出现脆弱性。

6 评估

为了评估 DafnyMPI 是否达成其设计目标，我们实现并验证了三种常见的数值偏微分方程求解器。我们从证明相关辅助代码的数量这一角度评估证明工程工作量，并从并行实现的性能这一角度评估运行时性能。结果表明，DafnyMPI 能显著提升运行时性能，加速比最高可达五倍，同时将新增的证明负担控制在可管理的范围内，尽管与顺序实现相比，证明相关代码的数量有所增加。所有实验均在一台搭载 M3 处理器、配备 48 吉比字节内存的 macOS 设备上进行；Dafny 求解器配置为在验证时使用不限数量的虚拟核心。

6.1 基准测试

我们使用 DafnyMPI 为三种不同的偏微分方程实现了数值求解器，每种方程都采用不同的通信模式。图 6 展示了使用 DafnyMPI 对每种偏微分方程模拟 100 个时间步后的求解结果。图 1 概述了各求解器的 MPI 通信模式。基准测试如下：

- **线性对流**。 如第2节所述，我们实现了用于求解一维线性对流偏微分方程的迎风格式。该方程描述了以恒定速度传播的波，如图6左栏所示。请注意，虽然线性对流的解析解能够完全保持波形不变，但数值解会引入扩散，无法完全保持初始条件中的尖锐边缘。图1 左上方的示意图显示，在 MPI 版本中，每个进程都必须在每次迭代时将其计算域中的最右值发送给相邻进程。

• **泊松**。 我们实现了用于求解带狄利克雷边界条件的二维泊松方程的雅可比方法。雅可比方法在每次迭代中根据某点的四个邻居更新该点的值。对于 MPI 实现，我们将计算域划分为大小相等的水平条带。每个进程都必须将其条带的顶部和底部行发送给相邻进程，并接收对应的两行边界。图1 右上方的示意图展示了这种通信模式（由于空间限制，图中使用列而不是行）。泊松方程的解是一个函数，其二阶空间导数（拉普拉斯算子）等于另一个作为输入提供的函数。经过多次迭代后，可以预期该解收敛到平衡态，因此进程必须在每次迭代时进行通信，以检查是否已达到收敛状态。后者通过对 MPI_ALLREDUCE的集体调用来处理。中间栏

f

图 6 展示了经过 100 次迭代后泊松方程的解，其中我们将初始条件设为零矩阵，并将方程的右端项设为一个函数$f(x,\ y)$ = 正弦$(2\pi x)$ 正弦$(2\pi y)$，其在离散计算域上的取值由外部实现的三角函数预先计算得出。

• **热扩散**。 我们实现了四阶龙格-库塔方法（四阶龙格-库塔法），并将其应用于二维热扩散方程。图6 右侧一栏用热力图展示了方程描述的行为。随着模拟的进行，最初位于图中心的热量逐渐消散。四阶龙格-库塔方法根据某个区域内的点来计算一个点的值，该区域包含曼哈顿距离不超过 4 的所有点。对于消息传递接口实现，我们再次将计算域划分为大小相等的水平条带。不过，与泊松问题不同的是，各进程必须一次交换 4 行数据，这使问题变得更加复杂。图1 底部的示意图展示了这一过程（与之前一样，我们用列代替行）。请注意，进程发送给两个邻居的数据可能会有重叠，而且发送操作可以并发进行。然而，接收操作必须使用独立的缓冲区。

6.2 证明工程工作量

为了深入了解其中涉及的工程工作量，表 3 报告了上述三个基准测试各自的代码行数（LOC）。尽管这只是间接衡量方式，但 LOC 大致反映了验证代码各部分所投入的工作量。我们分别列出每个基准测试中规约（Spec）、顺序实现（Seq）和并行 MPI 实现（Par）的代码行数。对于热扩散和泊松基准测试，我们还列出顺序实现与并行实现共享的代码库大小（Shared），其中分别包含用于实现雅可比和四阶龙格-库塔技术的若干方法。不出所料，由于复杂性增加，并行版本需要更多代码行来实现。

幽灵代码与可执行代码。 表3 还报告了各组文件中属于幽灵代码的代码行所占百分比。幽灵代码是不可执行的代码，其唯一用途是辅助证明。从图中可以看出，并行版本中幽灵代码的比例显著增加，这反映了证明复杂度的提高。同时，并行实现中的幽灵代码比例与顺序版本中某些较复杂子程序的比例相当。例如，四阶龙格-库塔方法（表3中的热扩散基准测试的共享子类别）需要大量辅助证明代码，以将函数规范与实现中使用的滚动数组操作联系起来。因此，幽灵代码占该共享组件的81%，接近同一基准测试并行实现中的89%。

Ta表 3. li 的数量　　代码行数和验证时间作为　　证明复杂度的度量。

	线性对流			泊松			热扩散					总计
	规范序列		并行规范共享序列				并行规范共享序列				Par	
LOC	209	106	379	384	126	153	1626	328	406	137 1516		5370
% 幽灵代码	100	45	65	100	72	55	91	100	81	57	89	87
耗时 验证（秒）	3	3	25	4	6	4	148	3	11	4	154	371

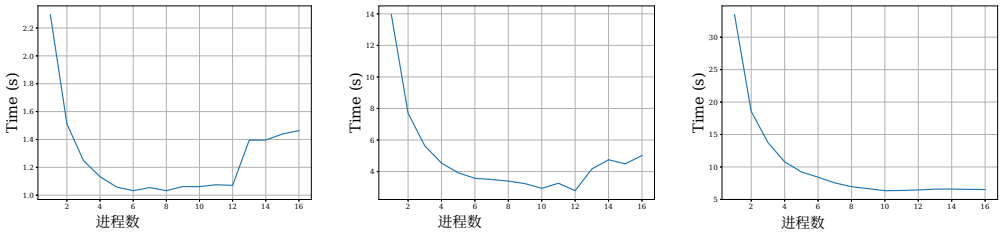

图 7. 运行　　时间随进程数变化的情况。从左到右：线性对流、泊松、热扩散

。

验证与编译速度。 验证所有基准测试大约需要6分钟，而使用Dafny内置编译器将代码编译为Python则不到20秒。 热扩散是验证时间开销最大的基准测试，如表3所示。这主要是因为在验证过程中，需要多次在二维表示和一维表示之间转换消息负载，以确保发送的数据在内存中连续存储。

6.3 运行时性能

鉴于证明一个 MPI 程序正确需要投入大量精力，我们希望确保所编写的程序相较于相应的顺序实现，确实能带来预期的性能提升。因此，我们使用 Dafny 编译器将全部三个基准测试转译为 Python，并以不同的进程数运行它们。[1] 图 7 展示了这项实验的结果。所有时间测量值均为三次运行的中位数。单进程运行时，我们采用顺序实现。正如预期，增加进程数会缩短运行时间；在热扩散基准测试中，加速比最高可达顺序版本的五倍。然而，通常情况下，进程数超过某个阈值后，收益会逐渐减少；对于线性对流和泊松，给 MPI 程序分配 12 个或更多进程后，运行时间甚至会增加。我们推测，运行时间增加是因为实验是在一台有 16 个核心的机器上进行的，因此为程序分配 12 个核心可能会干扰后台运行的其他进程，从而延长总运行时间。这种影响在线性对流基准测试中最为明显，因为就运行时间而言，它是所有基准测试中负载最轻的。

[1]为简化对程序结果的推理，我们当前的证明假设计算域大小可以被进程数整除。此假设不会影响通信模式或无死锁性的证明。当计算域大小不能被进程数整除时，我们会在测量运行时间前将其向上取整至最接近的可整除值。在我们的基准测试中，所有计算域大小都可以被 16 整除，因此使用 1、2、4、8 和 16 个进程进行的实验不受此调整影响。

7 讨论与未来工作

与证明助手的比较。DafnyMPI 系统的初始原型曾部分使用 Rocq 证明助手进行了形式化。为此，我们开发了一种领域专用语言，其设计类似于第 3.1 节中的核心演算，并为解释器编写了规约。尽管我们仍然认为这种方法可行，但在 Rocq 中验证线性对流基准测试期间，我们发现，利用 Dafny 内置的构造可以大幅减轻证明工程负担。比较 Rocq 中线性对流示例的无死锁性部分证明（约 1000 行，即使其中有许多证明目标被暂时承认）与 DafnyMPI 中完整的程序正确性证明（694 行），这种优势尤为明显。虽然使用其他验证框架实现与 DafnyMPI 类似的推理机制既可行又有用，但我们将此任务留待未来工作。

浮点运算。 在整个代码库中，我们使用 Dafny 的 `real` 类型来模拟浮点数。尽管这种方法在验证中很常见，但它并未考虑浮点下溢，因此无法精确模拟 IEEE-754 或其他浮点标准，也无法防止由此可能导致的除零错误。已有多项研究探讨如何自动验证使用浮点运算的程序 [1, 11], 这或许也是 DafnyMPI 未来工作的一个方向。

运行时间。 由于我们在 Dafny 中模拟了大多数数组操作，因此与使用 Python 的 Numpy 等原生库相比，运行时性能有所下降。这在一定程度上是我们有意为之：我们选择尽量缩小信任基，仅依赖 Numpy 执行单元素的设置和获取操作。原则上，我们可以在运行时将逐元素加法、切片和滚动操作等的实现替换为相应的 Numpy 操作，从而显著提升性能。

8 相关工作

相关工作主要有几个方向。

MPI 的动态分析与模型检查。 研究人员开发了多种用于验证 MPI 的系统。ISP [32], 、其后继 DAMPI [33], 以及 MOPPER [10]等实用工具会在受控调度器下执行 MPI 程序，或以其他方式分析先前执行的结果。其他工具，包括 TASS [28], 、MPI-SV [6], 和 CIVL-C [24], ，则将符号执行与模型检查相结合，以探索所有可能的消息交错。这些方法所需的证明工程不及 DafnyMPI，但仍可能遭遇状态爆炸，尽管已有技术能够剪枝搜索空间 [31]。

锁顺序。锁的排序此前已被用于在多种验证场景中确保无死锁性。一个强制规定锁获取和接收全局顺序的系统已为 Chalice 验证器实现 [22] ，而一个支持条件变量的扩展已为 VeriFast 实现 [13]。近期工作 [15] 使用更复杂的排序拓扑来管理高阶锁。这些研究聚焦于共享内存并发和动态线程创建场景中的锁，在这些场景中，创建新线程时可以建立或转移通信义务。相比之下，我们针对 MPI 环境：进程数任意但在整个执行过程中固定，这意味着程序员可以将整个通信拓扑描述为进程 ID 的函数，而无需显式管理每个进程的通信。此外，MPI 中的数据传输可以相对于发起进程异步发生；我们的操作语义通过消息负载缓冲区，以及点对点操作的初始化阶段与完成阶段之间的分离，明确地刻画了这一行为

。DafnyMPI 还允许同时发送和接收，并提供了一种便捷方式来集成集合通信操作。

使用 *Dafny* 实现并发。 已有多个项目探索了 Dafny 环境中的并发。

DafnyInfoFlow 项目 [29] 增加了对并发推理的支持，也采用了 rely-guarantee 方法，但其主要目标是防止信息从私有变量中泄露。相比之下，DafnyMPI 关注无死锁性、终止性和函数等价性。

Armada [23] 是一种工具和编程语言，其操作语义以 Dafny 建模，并支持验证类 C 并发代码。由于 Armada 并非 Dafny 库，而是一种独立语言，因此 Armada 程序无法像 DafnyMPI 那样直接复用 Dafny 构造。此外，Armada 明确面向低层共享内存并发，这与消息传递接口（MPI）属于根本不同的场景。

其他方法。 以 Iris [17] 和 Steelcore [30],为例，并发分离逻辑（CSL）通过资源（堆）、资源所有权以及用于堆谓词的分离蕴涵等共享内存语义扩展了霍尔逻辑 [3]。CSL 直接对堆建模，而 DafnyMPI 则将这些细节抽象掉，从而减轻证明工程的负担，但也限制了其对消息传递接口（MPI）程序的适用性。CSL 也不支持活性或无死锁性，通常需要补充其他逻辑（如 LinearActris [16],）来确立这些性质。

多位研究人员开发了旨在防止数据竞争和死锁等常见并发错误的类型系统。例如，基于会话类型的 TyPiCal [19], 、基于依赖类型的 CLL [12],，以及基于精化类型的异步液体分离类型 [18], 。与 DafnyMPI 不同，这些方法并不推理程序等价性。

最后，研究人员还为并发程序开发了各种其他形式化验证框架，例如 VCC[8], VerCors[2]。DafnyMPI 与这些框架的主要区别在于其核心设计。VCC 要求为 C 代码添加不变式以及前置条件或后置条件；VerCors 则使用分离逻辑注释验证 Java 代码。

9 结论

我们介绍了 DafnyMPI，这是一种新颖的验证库，它将对消息传递接口（MPI）并行程序的形式化推理引入为顺序代码设计的语言 Dafny。我们的方法能够验证 MPI 的核心性质，包括无死锁性，以及与顺序规约的函数等价性，而且无需自定义并发逻辑。通过利用 Dafny 内置的验证能力，并在此基础上添加 MPI 专用不变式，我们认为该方法兼具严谨性、易用性和可扩展性。

我们的形式化模型表明，只需对消息标签、屏障和缓冲区使用施加一小组前置条件，便足以保证无死锁性。这些前置条件通过核心演算和一组 Dafny 断言加以强制执行。偏微分方程求解器上的实验结果证明了我们方法的可行性：尽管证明工程仍然颇具挑战，但经验证的并行实现相比其顺序版本展现出显著的加速比。

展望未来，我们计划扩展 DafnyMPI，以支持更丰富的 MPI 特性（例如通信器、通配符），更无缝地融入科学计算编程生态系统，并进一步减少证明工程工作量。我们希望 DafnyMPI 能为经验证的高性能计算领域的未来研究奠定基础，并推动形式化方法在正确性至关重要、但传统技术难以实际应用的领域中得到采用。

数据可用性声明

DafnyMPI 及相关基准测试可通过 Zenodo 资源获取 [9]。

致谢

我们感谢匿名审稿人提供的宝贵反馈。本研究部分由美国国家科学基金会通过编号为 2313998 和 2513872 的奖项资助。

参考文献

[1] Rosa Abbasi、Jonas Schiffl、Eva Darulova、Mattias Ulbrich 和 Wolfgang Ahrendt。2021。KeY 中 Java 浮点程序的演绎验证。载于系统构造与分析工具和算法国际会议。 242 261 10 1007 978 3 030 72013 1_13 – 。doi：。/ - - - - Stefan Blom 和 Marieke Huisman。。用于验证并发程序的 VerCors 工具。载于形式化方法国际研讨会。[2] 2014. 127 131 10 1007 978 3 319 06410 9_9 – 。doi：。/ - - - - Stephen Brookes 和 Peter W. O'Hearn。。并发分离逻辑。ACM SIGLOG News 3（2016），47–65。doi:10.1145/2984450.2984457[3] 2016. Franck Cassez、Joanne Fuller、Milad K Ghale、David J Pearce 和 Horacio MA Quiles。2023。Dafny 中以太坊虚拟机的形式化语义与可执行语义。载于形式化方法国际研讨会。571–583。doi:10.1007/978-3-031-27481-7 32_[5] Aleks Chakarov、Jaco Geldenhuys、Matthew Heck、Michael Hicks、Sam Huang、Georges-Axel Jaloyan、Anjali Joshi、K. Rustan M. Leino、Mikael Mayer、Sean McLaughlin、Akhilesh Mritunjai、Clement Pit-Claudel、Sorawee Porncharoen- wase、Florian Rabe、Marianna Rapoport、Giles Reger、Cody Roux、Neha Rungta、Robin Salkeld、Matthias Schlaipfer、Daniel Schoepe、Johanna Schwartzentruber、Serdar Tasiran、Aaron Tomb、Emina Torlak、Jean-Baptiste Tristan、Lucas Wagner、Michael W. Whalen、Remy Willems、Tongtong Xiang、Tae Joon Byun、Joshua Cohen、Ruijie Fang、Juny- oung Jang、Jakob Rath、Hira Taqdees Syeda、Dominik Wagner 和 Yongwei Yuan。2025。形式化验证的云规模授权。载于2025 年 IEEE/ACM 第 47 届软件工程国际会议（ICSE）。2508–2521。doi:10.1109/ICSE55347.2025.00166[6] Zhenbang Chen、Hengbiao Yu、Xianjin Fu 和 Ji Wang。2020。MPI-SV：一种用于 MPI 程序的符号验证器。载于 ACM/IEEE 第 42 届软件工程国际会议论文集：伴随论文集（韩国首尔）。 93 96 10 1145 3377812 3382144 – 。doi：。/ 。Joseph W. Cutler、Michael Hicks 和 Emina Torlak。2024。提高 Dafny 中类型安全证明的稳定性。载于 Dafny 2024 - POPL 2024。https://popl24.sigplan.org/details/dafny-2024-papers/3/Improving-the-Stability-of-Type- Safety-Proofs-in-Dafny [8] Markus Dahlweid、Michal Moskal、Thomas Santen、Stephan Tobies 和 Wolfram Schulte。2009。VCC：并发 C 程序的基于契约的模块化验证。载于2009年第31届软件工程国际会议伴随卷。429–430。doi:10.1109/ICSE-COMPANION.2009.5071046[9] Aleksandr Fedchin、Antero Mejr、Hari Sundar 和 Jeffrey S. Foster。2025。论文配套材料“DafnyMPI：用于验证消息传递并发程序的 Dafny 库”。doi: 10 5281 17654850 。/zenodo。Vojtundefinedch Forejt、Saurabh Joshi、Daniel Kroening、Ganesh Narayanaswamy 和 Subodh Sharma。2017。用于发现 MPI 程序中通信死锁的精确预测分析。ACM Trans. Program. Lang. Syst. 39（2017）。doi:10.1145/3095075[11] Clément Fumex、Claude Marché 和 Yannick Moy。2017。浮点程序验证的自动化。载于经验证的软件。理论、工具与实验：第9届国际会议，VSTTE 2017，德国海德堡，7月 *22 23, 2017,* *9* 102 119 10 1007 978 3 319 72308 2_7 - 修订后的精选论文。– 。doi：。/ - - - - Deepak Garg 和 Frank Pfenning。。类型导向并发。载于并发理论国际会议。[12] 2005. 6 20 10 1007 11539452_5 – 。doi：。/ Jafar Hamin 和 Bart Jacobs。。无死锁监视器。载于欧洲程序设计研讨会。Springer，415–441。doi:10.1007/978-3-319-89884-[13] 2018. 1_15[14] Jonas Kastberg Hinrichsen、Jesper Bengtson 和 Robbert Krebbers。2019。Actris：分离逻辑中的基于会话类型的推理。POPL 4 2019 1 30 10 1145 3373096 （），– 。doi：。/ Jules Jacobs 和 Stephanie Balzer。。高阶无泄漏且无死锁锁。ACM 编程语言会议录[15] 2023. 7, 2023 1027 1057 10 1145 3571229 POPL（），– 。doi：。/ Jules Jacobs 、 Jonas Kastberg Hinrichsen 和 Robbert Krebbers。。无死锁分离逻辑：线性性为依赖型高阶消息传递带来进展性。POPL[16] 2024. 8 2024 10 1145 3632889 （）。doi：。/ Ralf Jung、Robbert Krebbers、Jacques-Henri Jourdan、Aleš Bizjak、Lars Birkedal 和 Derek Dreyer。2018。Iris 从基础开始：高阶并发分离逻辑的模块化基础。函数式编程杂志[17] 28 2018 1 73 10 1017 0956796818000151 （），– 。doi：。/S Johannes Kloos、Rupak Majumdar 和 Viktor Vafeiadis。2015。异步液体分离类型。载于第29届欧洲面向对象编程会议（ECOOP 2015）。396–420。doi:10.4230/LIPIcs.ECOOP.2015.396[18]

[19] Naoki Kobayashi。2006。一种用于无死锁进程的新型类型系统。载于 CONCUR 2006——并发理论：第17届国际会议，CONCUR 2006，德国波恩，2006年8月27日至30日。论文集17。233–247。doi:10.1007/ 11817949 16_[20] K Rustan M Leino。2010。Dafny：用于功能正确性的自动程序验证器。载于程序设计、人工智能与推理逻辑国际会议。

348 370 10 1007 978 3 642 17511 4_20 – 。doi：。/ - - - - K

[21] Rustan M Leino。。2017 使用 Dafny 进行易用的软件验证。IEEE Software（），– 。doi：.1109/MS。2017.4121212[22] K Rustan M Leino、Peter Müller 和 Jan Smans。2010。无死锁通道与锁。载于欧洲程序设计研讨会。Springer，407–426。doi:10.1007/978-3-642-11957-6_22[23] Jacob R Lorch、Yixuan Chen、Manos Kapritsos、Bryan Parno、Shaz Qadeer、Upamanyu Sharma、James R Wilcox 和 Xueyuan Zhao。2020。Armada：高性能并发程序的低成本验证。载于第41届ACM SIGPLAN程序设计语言设计与实现大会论文集。197–210。doi:10.1145/ 3385412.3385971[24] Ziqing Luo、Manchun Zheng 和 Stephen F Siegel。2017。使用 CIVL 验证 MPI 程序。载于第

24 1 11 10 1145 3127024 3127032 届欧洲MPI用户组会议论文集。– 。doi：。/ 。

[25] Sean McLaughlin、Georges-Axel Jaloyan、Tongtong Xiang 和 Florian Rabe。。2024 增强证明稳定性。载于 Dafny 2024 - POPL 2024。https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability [26] 消息传递接口论坛。1994。MPI：消息传递接口标准 1.0 版。https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps 访问日期：2025-05-04。[27] Salman Pervez、Ganesh Gopalakrishnan、Robert M Kirby、Robert Palmer、Rajeev Thakur 和 William Gropp。2007。用于验证 MPI 程序正确性的实用模型检测方法。载于并行虚拟机与消息传递接口最新进展：第14届欧洲PVM/MPI用户组会议。344–353。doi:10.1007/978-3-540- 75416-9 46_[28] Stephen F Siegel 和 Timothy K Zirkel。2011。基于 MPI 的并行程序自动形式化验证。ACM SIGPLAN Notices

46 2011 309 310 10 1145 1941553 1941603 （），– 。

[29] doi：。/ 。Graeme Smith。。2023 一种基于 Dafny 的线程局部信息流分析方法。载于2023年 IEEE/ACM 第11届软件工程形式化方法国际会议（FormaliSE）。86–96。doi:10.1109/FormaliSE58978. 2023.00017[30] Nikhil Swamy、Aseem Rastogi、Aymeric Fromherz、Denis Merigoux、Danel Ahman 和 Guido Martínez。2020。SteelCore：一种用于具有副作用的依赖类型程序的可扩展并发分离逻辑。POPL 4（2020）。doi:10.1145/ 3409003[31] Sarvani Vakkalanka、Michael DeLisi、Ganesh Gopalakrishnan、Robert M Kirby、Rajeev Thakur 和 William Gropp。2008。为 MPI 程序实现高效的动态形式化验证方法。载于欧洲并行虚拟机/消息传递接口用户组会议。

248 256 10 1007 978 3 540 87475 1_34 – 。doi：。/ - - - -

[32] Sarvani S. Vakkalanka、Subodh Sharma、Ganesh Gopalakrishnan 和 Robert M. 2008 Kirby。。ISP：一种用于模型检查 MPI 程序的工具。载于第13届 ACM SIGPLAN并行编程原理与实践研讨会论文集（美国犹他州盐湖城）。

285 286 10 1145 1345206 1345258 – 。doi：。/ 。Anh Vo、Sriram

[33] Aananthakrishnan、Ganesh Gopalakrishnan、Bronis R. de Supinski、Martin Schulz 和 Greg Bronevetsky。2010。一种用于 MPI 程序的可扩展分布式动态形式化验证器。载于2010年 ACM/IEEE高性能计算、网络、存储与分析国际会议论文集。

1 10 10 1109 2010 7 – 。

[34] doi：。/SC。。Anh Vo、Sarvani Vakkalanka、Michael DeLisi、Ganesh Gopalakrishnan、Robert M Kirby 和 2009 Rajeev Thakur。。实用 MPI 程序的形式化验证。ACM SIGPLAN Notices

44 2009 261 270 10 1145 1594835 1504214 （），– 。

[35] doi：。/ 。使用 Intel MPI库的大型 MPI 标签。2025 。
https://www.intel.com/content/www/us/en/developer/articles/technical/large-mpi-tags-with-the-intel-mpi.html 访问日期：2025-04-12。

收稿日期：2025年7月10日；录用日期：2025年11月6日