面向并行编程模型的动态契约分析

Yussur MustafaOraji科学计算研究所，达姆施塔特工业大学¨at Darmstadt达姆施塔特，德国 0009-0004-9922-3112

Alexander H¨uck科学计算研究所，达姆施塔特工业大学¨at Darmstadt达姆施塔特，德国 0000-0003-1931-773X

Christian Bischof科学计算研究所，达姆施塔特工业大学¨at Darmstadt达姆施塔特，德国 0000-0003-2711-3032

arXiv:2603.03023v1 [cs.DC] 3 Mar 2026

摘要—高性能计算中的并行编程依赖于 MPI 等底层应用程序编程接口，要求用户手动管理同步和资源。现有多种正确性检查工具可帮助开发无错误代码，但大多针对单一编程模型，限制了其适用范围。我们此前开发的静态分析工具 CoVer 采用基于契约的方法，使用户能够指定自定义错误检查规则，并支持新兴或非传统编程模型，而无需开发大量新工具。然而，静态分析无法充分推理指针别名或间接控制流等依赖运行时的行为。为此，我们推出 CoVer-Dynamic，这是一种动态分析扩展，复用 CoVer 的契约语言，提供统一的静态-动态验证框架。通过在运行时强制执行相同的契约，CoVer-Dynamic 提升了分类准确率，在标准化的 MPI 和 OpenSHMEM 基准测试中消除了假阳性，同时还能检测出仅靠静态分析无法发现的错误。评估结果表明，CoVer-Dynamic 的表现始终优于先进的正确性检查器 MUST，平均速度提升 2 倍。最后，我们的结果显示契约语言的表达能力存在局限，这促使我们在未来工作中支持更多错误类别。

索引词—**正确性、动态分析、消息传递接口、契约**

I. 引言

在 Fortran、C 和 C++ 等高性能计算（高性能计算）语言中，这些语言仍是主流。在这些环境中，分布式内存计算通过消息传递接口 [1], 、OpenSHMEM [2], 和 NCCL [3] 等低级应用程序接口进行管理。尽管这种组合具有高性能，但正确管理同步、资源生命周期和内存使用的重担落在了开发人员身上。任何使用违规都可能导致隐蔽的数据损坏、死锁或程序崩溃。

为缓解这些问题，人们开发了多种正确性检查工具。这些工具大致可分为主要在运行时执行分析的动态工具 [4], [5], , 以及仅分析源代码的静态工具 [6], [7], [8]。

静态工具通常能够全局查看源代码，因此可以检测到例如仅在运行时由特定测试输入触发的执行路径中存在的数据竞争，

作者谨此感谢德国联邦教育、研究、技术与航天部以及德国联邦州（http://www.nhr- verein.de/en/our-partners）对本工作/项目的支持。本工作/项目是国家高性能计算（NHR）联合资助计划的一部分。

```
1 布尔值 选项 = 输入();消息传递
2 接口_接收(buf, ..., req);如果 (
3 选项)buf[0]++; 消息传递接口_
4 等待(&req, ...);
5
```

(a) 当基于输入的选项为 true 时会触发数据竞争。

```
1 整型*缓冲区1， *缓冲区2;
2 // 缓冲区设置省略
3 消息传递接口_Irecv(buf1, ..., req);
4 buf2[0]++;
5 消息传递接口_Wait(&req, ...);
```

(b) 若 buf1 与 buf2 重叠，则会触发数据竞争

图1：难以自动检测的数据竞争示例

参见图1a。相比之下，动态工具虽然只能分析实际执行的特定代码路径，但通常能够检查运行时依赖属性，例如内存访问和缓冲区重叠，以检测数据竞争等问题，参见图1b。

此外，上述正确性工具通常针对单一并行编程模型，例如面向消息传递接口的检查器，如 [4], [8]。因此，将此类工具扩展为支持新兴模型可能会带来巨大的实现负担。尽管已有工作试图为（静态）分析提供统一抽象 [7], [9], , 但这些方法仍依赖针对有限几种并行模型硬编码的转换层。

为了解决这种模型特异性问题，我们此前推出了 CoVer [6], , 这是一款旨在将分析逻辑与特定并行模型的使用要求解耦的工具。CoVer 使用一种语言和模型无关的契约系统，以声明式方式指定正确性要求。CoVer 并不将针对消息传递接口或 OpenSHMEM 等 API 的任何 API 使用规则硬编码到分析器中，而是通过基于字符串的 API 注释的契约语言规范来定义正确性要求。这样便可对多个并行模型进行正确性检查，而不依赖于它们各自的特定 API。然而，由于 CoVer 是静态检查器，它仍然不可避免地受限于无法解析指针别名等问题，也无法处理其他运行时依赖条件。

为了解决这些局限，我们提出了 CoVer-Dynamic，将 CoVer 框架的适用范围扩展到动态分析领域。通过在静态和动态环境中复用相同的契约定义，我们构建了一个统一的验证框架。因此，CoVer-Dynamic 兼具动态工具的优势，同时保留了原始 CoVer 工具中与模型无关的契约。

此外，借助共享契约基础，我们支持一种混合工作流，以降低运行时开销。在这种工作流中，首先使用静态分析识别潜在的错误位点，随后通过有针对性的运行时插桩，在动态环境中验证这些问题。与我们之前评估过的静态—动态工具组合不同 [10]，CoVer-Dynamic 通过在静态和动态分析中维护统一的契约规范，避免了因并行模型API或错误类别支持不匹配而导致的不一致。此外，尽管此前混合分析仅限于数据竞争检测，但对契约的共同理解使耦合分析方法能够用于所有错误类别。

总而言之，我们的主要贡献如下：

- 一种名为 Cover-Dynamic 的、基于动态契约的并行编程模型错误检测方法。
- 支持静态工具 CoVer 的现有契约语言，从而可以完整复用这些契约。
- 与原始 CoVer 实现无缝互操作，实现有针对性的低开销插桩。

本文其余部分的安排如下。第二节介绍并行模型中的常见错误，并描述原始 CoVer 实现及其契约语言。第三节介绍 CoVer-Dynamic，以及动态执行期间重用静态契约所面临的具体挑战。第四节将我们的工具与最先进的 MPI 正确性检查器 MUST 进行评估 [4]。具体而言，我们还强调，通过静态信息精简动态分析所需插桩的混合工作流，可以显著改善运行时性能。第五节简要讨论相关工作。最后，我们在第六节总结全文。

II. 背景

CoVer-Dynamic 重用了 CoVer 的契约。因此，要讨论 CoVer-Dynamic 的实现，有必要了解 CoVer 的分析方法。本节介绍 CoVer 契约中可表示的一些错误类别，以及 CoVer 用于检测并行代码中这些问题的方法。

A. 错误类别

并行编程模型提供了多种通信范式，例如点对点通信、集合通信和远程内存访问通信，但使用这些通信方式开发代码时，往往会导致一些微妙的错误。本节以消息传递接口和 OpenSHMEM 为例，介绍其中一些错误。此处介绍的错误与 CoVer 论文 [6] 中所用的错误相同。

a) 缺少初始化或终结：许多基于库的并行编程模型要求在进行通信前显式初始化，并在执行结束后进行终结。省略这些步骤往往会导致程序崩溃。

b) 局部数据竞争：使用非阻塞通信时，API 调用会在通信执行之前结束。因此，在通信明确完成之前，必须避免对该缓冲区进行读取或写入访问。否则，

```
1  int src = 4;
2  #if defined(USE_消息传递接口)
3        消息传递接口_Isend      req1);
4  #else
5        shmem_int_put_nbi(global_buf, src);
6  #endif
7  src      ; // 错误：数据竞争!
```

图2：本地数据竞争示例：通信完成前，源数据就已被写入。

```
1  消息传递接口_Isend(buf      );
2  MPI_Isend(buf2,     req1);// 错误：请求句柄丢失!
```

图3：请求泄漏示例。由于重复调用，第一个句柄丢失。

通信与本地内存访问之间没有顺序保证，缓冲区值和/或通信值将变得未定义。我们将这类错误称为局部数据竞争。

MPI 和 OpenSHMEM 的示例如图2所示。MPI 和 OpenSHMEM 例程都会将 src 中的数据发送到目标进程。然而，在确保通信完成之前，src 就已被写入。因此，实际发送的数据未定义。为简洁起见，后续代码示例将仅使用 MPI。

c) 句柄生命周期：与并行编程模型交互时，有些结构通常以不透明句柄的形式表示。例如，消息传递接口为数据类型、通信器、窗口和请求定义了句柄，而 Open-SHMEM 使用团队和上下文句柄。

这些句柄是资源，必须正确创建和释放。有些句柄还需要更复杂的设置或销毁过程；例如，自定义 MPI 数据类型在创建后、用于通信之前，还必须显式提交。句柄管理方面的问题归入句柄生命周期错误类别。

一种更难检测的 MPI 请求泄漏是句柄生命周期错误的子类，如图3所示。每次通信调用都会由 MPI 分配一个请求。若在两次调用之间没有完成操作，就对两个调用使用同一个请求句柄，那么原有的请求句柄就会被覆盖，导致底层请求泄漏。

d) 常见远程内存访问错误：远程内存访问（RMA）允许直接单边访问进程间内存。不过，在使用 RMA 通信之前，必须先进行一些设置。MPI 尤其如此：MPI 要求事先创建一个用于通信的 RMA 窗口。OpenSHMEM不要求这样做，因为全局变量和静态变量始终被视为可用于通信的暴露内存；但在其他情况下，必须使用 OpenSHMEM 内存管理例程。MPI 还要求通信在某个纪元内进行，而纪元可根据所用的同步方法创建；栅栏和锁定-解锁中只能使用其中一种（不允许混用）

```
1   消息传递接口_窗口_栅[illegible]in);2   消息传递接口_窗口_锁_all(_
win);// 错误：混合同步！3   [illegible]递接口_Get(&buf,win);
```

图 4：MPIRMA 中混合同步错误的示例（栅栏+锁）；每个窗口在同一时间只能使用一种同步方法。

```
1   int 消息传递接口_Finalize() CONTRACT( 前置条件 { call!(MPI_Init) });
2
3   int消息传递接口_Get(...) CONTRACT(
4     前置条件 { call!(MPI_Init) }
5     POST {
6       no!  (读取!(*0)) 直到!          (调用_tag!(远程内存访问_complete,$:7)),
7       否! (写入!(*0)) 直到! (调用_tag!(远程内存访问_complete,$:7))
8     });
9
10  int 消息传递接口_Win_栅栏(int 断言, 消息传递接口_Win win)
11    合约( TAGS { 远程内存访问_complete(1) });
12  int 消息传递接口_Win_unlock_all(消息传递接口_Win win)
13    CONTRACT( TAGS { 远程内存访问_complete(0) });
```

图 5：契约定义示例。

或 Post-Start-Complete-Wait（PSCW）。

图 4 给出了此类错误的一个示例。这里同时使用了栅栏和锁同步，这是不允许的，并且可能导致未定义行为。

B. 使用 *CoVer* 进行验证

CoVer 是一款静态正确性检查器，旨在验证使用并行编程模型编写的代码。它定义了一种契约语言，用于指定应用程序编程接口函数的要求，并通过过程间数据流分析对这些要求进行验证。

本节介绍 CoVer 的核心概念，这些概念对于理解 CoVer-Dynamic 的介绍十分重要。契约语言尤其关键，因为 CoVer-Dynamic 重用了这些定义。

1) 契约语言：考虑一个简单的情况：尝试检测 MPI 中缺少初始化的错误，即未调用 MPI_Init（这是一个简化示例，不考虑线程处理或 MPI 会话支持）。要将此情况编码为契约，必须将其重新表述为一项要求：使用 MPI 调用时，必须事先调用 MPI_Init*must* 。

图 5 展示了为 MPI_Finalize 编码这一要求的契约。MPI_Finalize 的函数声明附带了这项新要求的注释。PRE 表示该要求在调用 MPI_Finalize 的调用点之前生效，而 call! 是必须执行的操作。

再考虑一个更复杂的示例：检测局部数据竞争。该要求是：在每次调用非阻塞通信函数之后，不得进行读写操作，直到通过另一次应用程序编程接口调用确认该调用已完成。与前一个示例类似，这种逻辑结构可以直接转换为契约语言。

然而，这还不够：操作都被禁止，但仅限于针对任何读取或写入操作。

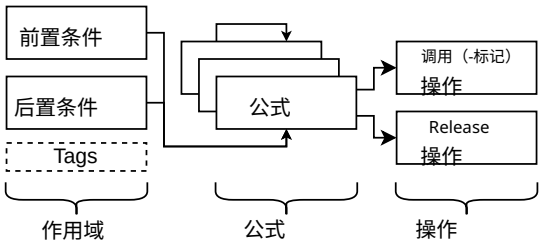

图 6：契约语言结构概览。

通信缓冲区的操作。此外，不仅必须调用某种完成函数，还必须明确完成通信：如果是点对点调用或集体调用，则使用相应的 MPI 请求；如果是 MPI RMA 操作，则使用 MPI RMA 窗口句柄。

为了正确指定这些限制，CoVer 允许指定调用点的参数是否必须与相应操作匹配。图 5 给出了一个要求示例：该要求禁止 MPI_Get 发生数据竞争，同时还要求完成 MPI 初始化。后一项检查只需使用 PRE 块即可完成；前一项则通过两个拆分的要求实现，分别禁止读取和写入内存访问。这些要求位于 POST 作用域中，表示在每个 MPI_Get 调用点之后均启用该要求。对于内存访问，给出的参数为 *0 ，这表示在 completioncall 完成之前，禁止显式访问由第一个参数指向的内存；这与 MPI 语义相符，因为 MPI_Get 的第一个参数是通信缓冲区。

MPI RMA 调用可以通过多种不同机制完成，例如通过栅栏或锁定—解锁。为了同时涵盖这两种机制，相应函数会被标记为 rma_complete，并将存放 RMA 窗口的参数索引写入该标签。随后，MPI_Get 的合约会使用 call_tag! 操作，而不是使用以该标签为目标的 call!。因此，只要窗口句柄匹配，任一函数都可以完成通信；通过从 $（标签中的参数）到 MPI_Get 的第八个参数的映射即可确保这一点，因为第八个参数存放窗口句柄。

图 6 展示了合约的一般结构。合约是附加到 API 函数的字符串，其最高层级由作用域构成。该语言定义了三种可能的作用域：PRE、POST 和 TAGS。PRE 和 POST 可以包含要求，这些要求必须分别在 API 函数的每个调用点之前或之后成立，而 TAGS 作用域则可用于如图所示的 call_tag! 操作。这些要求使用（可能嵌套的）公式进行指定，其中操作是原子字面量。在局部数据竞争示例中，两个要求通过合取给出（即两者都必须满足），不过也可以指定更复杂的公式。

2) 工具架构：CoVer 基于 LLVM 中间表示 [11] (LLVM IR)上的编译时遍次实现，并通过一个轻量级编译时包装器支持跨翻译单元的分析。分析过程可以

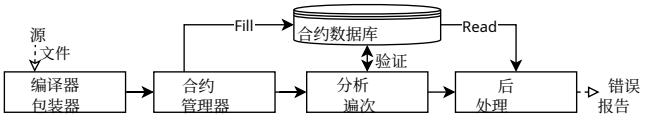

图 7：CoVer 架构概览，摘自 [6]

分为四个部分：(i) 调用编译器包装器，(ii) 合约管理器，(iii) 运行每个分析遍次，以及 (iv) 合约后处理。图 7 给出了概览。

这些合约以字符串注解的形式，通过 C/C++ 函数声明属性给出。为简化实现，该工具提供了一个宏，用于隐藏特定的属性语法；图 5 展示了 CONTRACT 宏的用法。

编译器包装器会拦截所有编译步骤，并强制底层编译器提供 LLVM IR，而不是生成目标文件。编译进入链接步骤后，所有 IR 文件会被链接在一起，随后包装器运行合约管理器和分析遍次，从而使分析能够跨翻译单元进行。

合约管理器会处理所有合约注释，并为每个注释生成一种合约树表示。合约树是一种基于对象的格式，用于存储每个合约的结构，即前置条件和后置条件、（嵌套的）公式及其包含的操作。此外，还会保存公式的类型，类型包括：(i) 合取，要求所有子公式均得到满足；(ii) 析取，要求至少一个子公式得到满足；或 (iii) 异或，要求恰好一个子公式得到满足。所有合约以及每个函数与相应标签之间的映射，都会保存到全局合约数据库中，供分析遍次使用。

CoVer 包含三个分析遍次：PreCall、PostCall 和 Release。PreCall 和 PostCall 会检查函数是否在每个调用点之前或之后被调用。如果前置条件或后置条件中分别包含调用操作，就会使用它们。目前，Release 仅用于释放操作，而释放操作始终位于后置条件中。它会检查被禁止的操作是否发生在释放操作之前（例如图 5 中的数据竞争检查）。这些遍次不会对整个公式运行，而只取决于其中包含的原子操作。完成后，结果会写回合约数据库。

现在，所有操作的满足状态都已知，后处理遍次便可以解析公式。它会递归检查每个嵌套公式是否满足，直至顶层。如果顶层公式被违反，

错误会生成并报告给用户。

III. 实现动态契约分析

我们的目标是创建一种新的动态工具，使其能够检查与静态工具 CoVer [6] 相同的契约。通过重用契约定义，现有定义仍然有效，从而让两种工具能够执行相同的检查，同时发挥动态分析的优势。

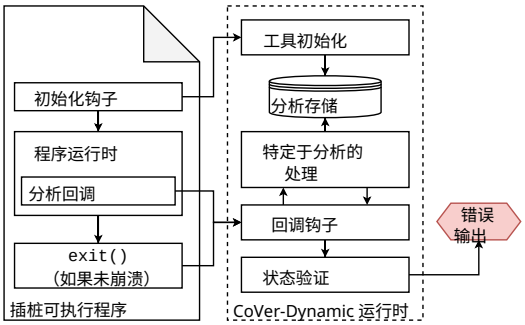

图 8：动态分析库在运行时的概览。

A. 动态契约分析的挑战

CoVer 的契约原本并非用于动态环境，因此这种新方法在实现上面临一些挑战。本节将讨论这些挑战，即契约定义的可用性、对与模型无关的函数调用拦截的需求，以及实时错误报告。

a) 契约可用性：一个基本问题是契约的可访问性。虽然可以静态地从注解中读取这些契约，但程序一旦编译完成，它们便不可用。在内部，注解以函数声明属性的形式给出，这些属性仅与编译过程相关，因此会在运行时之前被丢弃。因此，我们的动态工具需要另一种读取契约的方式。

b) 调用拦截：此外，为实现与模型无关的分析，新的动态工具必须完全不感知所使用的编程模型。正确性检查器常见的拦截点（例如用于消息传递接口的 PMPI 接口）因这一限制而不可行。

c) 实时错误报告：最后，另一个障碍是契约的结构。它们以公式的形式设计，静态工具可以在后处理步骤中解析这些公式。动态地做到这一点要困难得多：程序可能有多个退出点，而崩溃可能导致所有分析数据丢失。由于崩溃在有缺陷的代码中很常见（这也是使用正确性检查器的主要原因），该问题尤为隐蔽，因为崩溃可能正是由该工具试图检测的特定缺陷引起的。

虽然我们也可以将分析数据保存到磁盘上并进行事后分析，但这会带来显著的开销（尤其是在集群使用的并行文件系统上），因此不可行。

我们已在工具中解决了上述每个问题，具体细节将在后续章节中介绍。

B. 编译时插桩

由于 CoVer 已经会解析源代码中的合约，我们决定将编译时插桩实现为一个可选的编译过程（通过在其编译器包装器中添加一个选项启用），然后将合约信息添加到可执行文件中。动态工具的运行时本身构建为一个静态

```
// 回调函数声明
void CB_Initialize(int* argc, char*** argv,
                   ContractDB_t* DB);

// ... -> 可变参数: 对于调用点中的每个参数,
// 首先是参数大小,                然后是实际参数
void CB_Function(bool isRel, 无效* 函数,
                 整数 数值_参数,     ...);

无效CB_内存(布尔值isRel,无效* buf,          布尔值 是写入);
```

图9：由CoVer生成的对动态工具的回调函数。

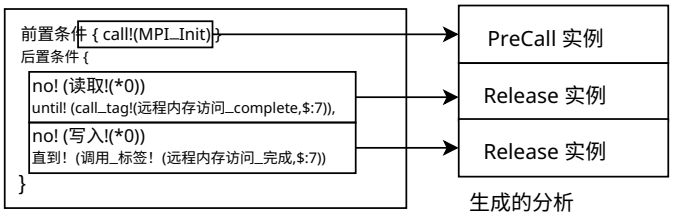

图10：为图5中给出的合约创建的分析实例。

库，该库会链接到可执行文件中，并使用专门添加的回调函数执行分析。

工具结构的概述见图8，额外的插桩函数列于图9。不过，isRel、argc和argv参数仅会在第III-D节中被引用，并用于实现 [10]中所述静态-动态耦合工作流的覆盖率警告。

*a) CoVer-Dynamic*的初始化：一旦CoVer完成契约解析，它们便被编码为C风格结构体表示。所有契约被打包到一个数据库中，并作为全局变量嵌入可执行文件。

在程序起始处插入一个对动态工具的回调，并传入指向该数据库的指针（图9中的CB_Initialize）。该函数接收一个指向全局契约数据库结构体的指针；该结构体先前已通过静态插桩添加到可执行文件中，并作为动态工具接收插桩契约信息的主要途径。

为此，还需要进行一些额外处理。例如，某些操作可能会引用函数名（例如call!(消息传递接口_Init)）。动态工具无法链接任何编程模型。因此，即使函数名可用，该工具也无法检查该函数是否被调用，因为它无法识别该函数。CoVer-Dynamic并不理解消息传递接口，而只掌握合约字符串。在运行时，它无法检查任何被调用的函数是否为消息传递接口_Init，因为名称已不再可用，仅剩一个函数指针。我们的解决办法是：不仅在编译时存储函数名，还在每个调用操作的表示中存储指向该函数的指针。这样，在运行时它就可以用作相应函数的标识符。

b) 拦截 *API* 调用：为了拦截对编程模型的调用，我们决定实现一个回调系统。合约中提及的每个函数，或附加了合约的每个函数，都会被添加到“相关函数”列表中。然后，

其中每个函数调用点都会创建一个指向动态工具的回调（图9中的 CB_Function）。

该回调函数是合约中提及的所有函数的通用处理函数，因为无法为不同函数定义不同的回调：由于编程模型与验证工具严格独立，因此无法在分析前为回调构造特定的函数目标。因此，作为通用处理函数，为了提供调用点所需的所有信息，它还必须支持可变参数，以适应不同插桩函数的参数数量。

回调的第一个和第二个参数分别是相关函数的指针及其参数个数。随后，对于相关函数调用点中使用的每个实参，回调首先接收该参数的大小，然后接收参数本身。指定每个实参的大小至关重要，否则动态工具将无法知道应为每个实参读取多少字节。这使动态工具能够访问每个相关函数的参数值以进行分析，并且契约中引用的函数可以与第一个参数中的指针进行比较。

c) 附加插桩：还会为局部内存访问插入回调函数，以存储类型（读/写）以及被修改内存位置的指针（图9中的CB_Memory）。这对于推理局部数据竞争等情况是必要的，否则该工具无法检测主机程序的局部内存访问。该回调很简单，其参数为被访问内存的指针，以及该访问是写入还是读取。

最后，我们也通过 atexit 函数捕获程序退出；程序（隐式地）调用 exit 后，该函数会提供一个回调。这对于错误输出来说并非 必要，因为如前所述，如果程序崩溃，错误输出会阻止该工具报告错误；但捕获 PostCall 操作的语义则需要这样做。它表明所有仍处于待处理状态的 PostCall 操作均已遭到违反；例如，如果在程序退出时尚未验证合约 POST {call!(MPI_Finalize)}，那么显然程序缺少终结操作。

C. 运行时交互

编译时插桩插入合约和回调后，我们会将可执行文件与包含回调函数定义的静态库链接起来。这就是动态工具的核心：在运行时，该库会执行必要的检查，以验证带注释的合约。

1）管理分析实例：在初始化回调期间，针对合约数据库中的每个合约，递归检查合约公式中所有出现的契约操作（例如 PreCall、PostCall、Release）。每找到一个操作，就会创建相应分析对象的实例，并将其添加到分析存储中。对于图 5 中附加到 MPI_Get 的示例合约，

```
前置条件 {
    call_tag!(纪元_栅栏_create,$:7) MSG "栅栏历元" ^
    call_tag!(纪元_锁_create,$:7)                MSG "锁定历元" ^
    call_tag!(纪元_pscw_create,$:7)              MSG "PSCW 历元"
}
```

图11：MPI_Get 合约示例，要求存在一个开放纪元，同时禁止混合同步。

这可以在图 10 中看到。每个分析实例只负责一个操作。因此，对于图 10 所示的示例，给定的 PreCall 实例会检查 MPI_Init 是否在 MPI_Get 之前被调用。

执行此项分析所需的信息由插桩回调函数提供。所有尚未得出结论（既未违反也未满足）的分析都会收到每个回调的通知。随后，各项分析可以独立判断这些信息是否与自身相关，并根据当前程序状态返回分析已满足、已违反或（仍）未知。

例如，调用 MPI_Send 与图 10 中给出的 PreCall 操作无关，但调用 MPI_Init 会使其满足，而调用 MPI_Get 则会使其违反。尽管在调用 MPI_Init 之前调用 MPI_Send 同样是错误的，但由于合约和分析实例彼此隔离，因此会由另一个分析对象负责检查 MPI_Init 的这一错误。

所有返回已确定状态（无论是已满足还是已违反）的分析都会从分析存储中移除，因此不再接收回调。这样可以避免不必要且开销高昂的函数调用，从而提升工具性能。为进一步提升性能，分析还可以直接声明哪些回调类别与自身无关。例如，对于任何 PreCall 操作，内存访问都无关紧要，因为只有函数调用才相关。此外，对于 Release 操作，只有当其中被禁止的操作是内存访问（例如局部数据竞争）时，内存访问才相关；如果被禁止的操作是函数调用，则内存访问无关紧要，例如检查是否使用尚未提交的 MPI 数据类型进行通信。这样，相应的分析就可以完全不接收内存回调，从而降低开销。由于插桩涵盖了所有 内存访问，涉及的回调数量极其庞大，因此避免内存回调尤为重要。优化需求以及插桩对运行时的影响将在第 IV-B 节中进一步阐述。

2）状态验证：如第 III 节所述，一旦违反合约，就必须立即输出错误。然而，不能仅仅因为某个操作被违反就报告错误。契约公式可以用（嵌套的）合取、析取和异或关系来指定，这意味着违反某个操作并不一定会导致整个合约被违反。指定混合 RMA 同步的合约时（参见第 II-A 节）就能看出这一点，图 11 展示了一个简单示例。若恰好一个调用操作得到满足，则给定的合约便得到满足。如果一个都没有满足，则没有开启任何纪元；如果有多个得到满足，则发生了混合同步。静态 CoVer 分析会在后处理中解析公式，但动态工具无法保证存在统一出口点。因此，一旦公式被违反，就必须立即检测并报告。

这是图 8 中状态验证 步骤的任务。每当一次分析得到结论后，库就会进入状态验证阶段。不能只对违反的分析实例执行此步骤，否则异或公式可能无法被判定为无效；多个“已满足”的契约会导致更广泛作用域中的契约违反。

简要来说，状态验证步骤总是从操作层级开始，而操作是一个原子公式。错误会沿公式树向上逐级传播；只要这些错误不是良性的，且未被析取或异或视为与更广泛的正确性无关，顶层作用域公式就必然会被标记为违反。此时，程序状态表明确实发生了契约违反，并执行错误输出。

具体而言，一旦某个分析得到满足或被违反，工具就会进入状态验证阶段。如果父公式的类型是合取，且当前公式被违反，那么父公式也必然被违反。因此，工具会将父公式判定为无效，并使用父公式重新运行状态验证。对于析取，只有在不存在状态未知的兄弟公式时才会发生这种情况；对于这些兄弟公式，程序运行时间还不足以判定其违反或满足。对于异或，工具会检查是否有多个子公式已得到满足，或所有子公式都已被违反。

如果当前公式已经是顶层公式（即完整的前置条件或后置条件），且它被违反，则可以确定已经发生错误。随后，工具将通过递归检查契约公式的子项来输出错误，并报告每一处违反。

请看图 11 中给出的混合同步检测合约示例。如果某个子公式（例如负责栅栏的子公式）被违反，由于另外两个兄弟公式的状态未知，整个合约的状态也未知。如果锁完成公式得到满足，整个合约仍处于未知状态；最后一个兄弟公式得到满足会导致父公式违反，而该公式被违反则会使父公式得到满足。由于这是异或关系，因此应当恰好满足一个子公式，而不能满足多个。

D. 静态-动态耦合

动态工具可能带来显著的运行时开销；此外，如果执行过程中未访问特定代码路径，它们也可能无法报告错误，见图1a。我们可以通过静态-动态耦合来避免这些问题，如 [10] 所示：

- 通过静态引导的过滤，可显著降低开销
- 可以采用基于覆盖率的方法检查尚未访问的位置

由于静态工具通常更倾向于产生假阳性而非假阴性，动态工具带来的开销可以

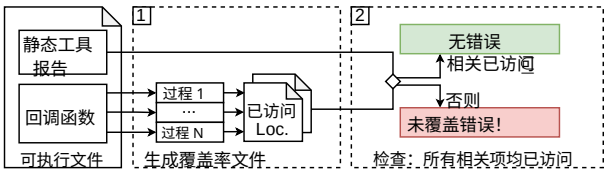

图12：覆盖率错误生成过程：(1) 正常执行，包括转储已访问位置。(2) 收集已访问位置，并与静态报告中提及的位置进行比较。

通过仅对静态工具报告错误的位置进行插桩而大幅降低。

此外，由于静态工具会检查并报告整个代码中的问题，动态工具可以检查静态工具报告的所有位置是否都实际被访问过，从而避免因包含错误的位置未被访问而漏报错误。例如，如果静态工具检测到图1a中的问题，但动态工具未访问相关位置，就会报告覆盖率错误。

1) 过滤式插桩：过滤式插桩很容易实现：由于插桩是在使用编译包装器进行 CoVer 分析后立即执行的，因此只需在错误报告未引用某个代码位置时，有选择地省略该位置的插桩即可。为增强通用性，当其他工具的输出采用 [10]中定义的交换格式时，CoVer-Dynamic 也可以读取这些工具的报告。

2) 基于覆盖率的错误报告：实现覆盖率错误略为复杂；该实现的概览见图12。首先，必须让 CoVer-Dynamic 知道哪些位置是相关的（在静态工具报告中被引用），哪些不是。使用插桩过滤时，这一点是隐式的：不会存在无关的插桩。然而，尽管静态工具通常比假阴性更容易产生假阳性，假阴性仍然可能出现。因此，为确保高准确率，默认仍采用完整插桩。作为替代，我们为插桩回调函数扩展了另一个参数（在图9中记为 isRel），用于指示该回调的位置是否被静态工具在错误报告中引用。利用这些信息，CoVer-Dynamic 随后可以收集运行时被标记为相关的所有位置。

遗憾的是，仅凭这一点还不够：给定一段 MPI 代码，其中进程秩 0 和 1 分别遇到一个各自唯一的相关位置，两个进程秩都会独立地针对另一进程的对应位置报告覆盖率错误。MUST 通过消息传递接口 [10] 收集每个进程遇到的位置列表。然而，由于 CoVer-Dynamic 与模型无关的特性，这里无法做到这一点。

相反，在程序退出时，CoVer-Dynamic 会为每个进程写入一个具有相同前缀的唯一文件，其中包含每个已访问相关位置的信息（图12中的步骤1）。

为了生成覆盖率信息，可执行文件会再次（串行）启动，并使用特殊的 --cover:check-coverage 标志（图12中的步骤2）。程序参数在启动时由 CoVer-Dynamic 捕获（见图9），从而使其能够在实际程序执行之前识别该标志。随后，CoVer-Dynamic 跳过实际程序的执行，转而从先前生成的文件中收集每个已访问位置。由于它在同一可执行文件中执行，嵌入的相关位置列表仍然有效，因此可以将先前实际访问并写入磁盘的位置与静态工具报告的位置进行比较。如果二者不匹配，则会生成相应的覆盖率错误，之后程序退出。

假设图1a中，进程1和进程2的选项均为 false。数据竞争不会实际发生，但静态工具会报告第2行（MPI调用）和第4行（冲突的内存访问）存在问题。使用 CoVer-Dynamic 时，它会将第2行标记为已访问，但不会标记第4行。因此，使用覆盖率检查标志执行时，该行将不会出现在任何覆盖率文件中，从而产生覆盖率错误。反之，如果某个秩确实执行了第4行，它本身就会报告数据竞争，并将第4行标记为已访问。汇总覆盖率信息时，所有相关位置都会被标记为已访问，因此不会报告覆盖率错误，从而避免重复或无效的报告。

在这里，基于契约的方法的一项显著优势显现出来：静态-动态耦合的原始论文仅关注本地数据竞争检测 [10]。但我们在 CoVer-Dynamic 中的实现完全独立于错误类别，只要求对契约定义有共同理解。由于 CoVer 和 CoVer-Dynamic 始终如此，这种方法便能以透明的方式将耦合工作流的适用范围扩展到任意错误类别，只要这些类别能够用契约语言表示即可。

IV. 评估

我们进行了多项测试，以验证新的动态契约分析的有效性。这些测试分为两部分：（i）使用现有的 RMARaceBench [12] 和 MPI-BugBench [13]（第 1 级）测试套件进行的分类质量测试，以及（ii）使用代理应用 LULESH [14]，PRK Stencil [15] 和 TeaLeaf [16]进行的开销分析。我们将 CoVer-Dynamic 与最先进的正确性检查器 MUST [4] v1.11.2 进行比较，其中包括在 [10]中引入的静态-动态耦合修改。

所有测试均在 Lichtenberg II 集群上运行，分类质量测试使用 OpenMPI 5.0.5，开销分析使用 OpenMPI 4.1.6。这是因为 MUST 与进程秩数量较多的较新 OpenMPI 版本不兼容，在我们的测试中导致了死锁。我们使用 Clang 编译器（开发版）22 来编译测试和工具。所生成的工具源代码、脚本和结果可在 [17] 获取。

A. 分类质量

使用 5 种工具配置分析了 MPI-BugBench 第 1 级和 RMARaceBench 中的所有测试，共 458 项：(i) CoVer-Dynamic、(ii) CoVer、(iii) CoVer-filtered、(iv) MUST、(v) MUST 筛选。后缀 -filtered 指的是

表 I：分类质量结果。

RMARaceBench （消息传递接口本地数据竞争）								
	真正例	真负例	假正例	假负例	NC 真阳性	NC 假阳性	超时	A
CoVer-Dynamic	35	31	0	4	3	0	0	[illegible]
CoVer 34 28			3	8	-	-	0	0.85
CoVer-filtered	32	31	0	7	3	0	0	0.90
MUST	39	27	1	3	-	-	3	0.94
MUST 筛选	33	28	3	7	2	0	0	0.86

MPI-BugBench （II-A 中的错误类别）								
	真正例	真负例	假正例	假负例	NC 真阳性	NC 假阳性	超时	A
CoVer-Dynamic	16	23	0	2	6	0	0	0.96
CoVer 24 22			1	0	-	-	0	[illegible]
CoVer-filtered	16	23	0	2	6	0	0	0.96
MUST	21	22	1	2	-	-	1	0.93
MUST 筛选	13	22	1	3	0	0	8	0.90

RMARaceBench （OpenSHMEM 测试）								
	真正例	真负例	假正例	假负例	NC 真阳性	NC 假阳性	超时	A
CoVer-Dynamic	28	24	0	5	2	0	0	[illegible]
CoVer 28 22			2	7	-	-	0	0.85
CoVer-filtered	25		0	8	2	0	0	0.86

TP↑：真正例，TN↑：真负例，FP↓：假正例，FN↓：假负例。其中，最佳结果以绿色突出显示。TO↓：超时（30 秒，以红色突出显示），A↑：准确率↑：越高越好，↓：越低越好

由于动态工具覆盖率警告，其他静态工具错误覆盖率阳性

采用定向插桩的动态分析：仅对静态分析工具（此处为 CoVer）判定可能导致错误的代码位置进行插桩。这属于耦合分析工作流，该工作流以牺牲准确率为代价来降低运行时开销。

测试执行后，会被归入以下类别之一：(i) 真阳性（TP），(ii) 真阴性（TN），(iii) 假阳性（FP），(iv) 假阴性（FN），(v) 超时（TO）。此外，还使用两种未覆盖分类来表示由静态-动态耦合引起的情况（见第III-D节）。当动态工具发现自己未访问静态工具标记为包含错误的位置时，它可能会报告覆盖率错误。这样可以通过提供额外的错误报告来提高准确率。作为静态工具，CoVer 不会报告覆盖率警告。只有在同时过滤插桩时，MUST 才会报告覆盖率警告（这两项功能绑定在一起）。对于 CoVer-Dynamic，无论是否过滤插桩，都可以生成覆盖率警告。未覆盖分类分为 NC-TP（未覆盖，真正例）和 NC-FP（假阳性）。最后，准确率（A）定义为正向结果（TP、TN、NC-TP）占测试总数的比例。

我们的分析结果见表 I。不过，需要注意的是，该表不包括针对远程 RMA 数据竞争、无效参数（例如使用 NULL、错误的 MPI 常量等）、数据类型不匹配和死锁检测的测试。目前，这些内容无法用 CoVer 定义的契约语言表示，并会导致显而易见的假阴性。尽管如此，我们仍执行了这些测试，以确保不会报告假阳性（实际并未出现假阳性）；不过，为避免已知的功能缺口影响结果，这些测试未列入表 I。未来将考虑扩展对这些错误类别的支持。

CoVer-Dynamic 的分类质量结果总体上令人满意。动态分析取得了最高的总准确率。此外，所有测试中均未报告任何假阳性。这正是动态分析相较于纯静态分析的主要优势：消除假阳性。与静态版本相比，CoVer-Dynamic 还能够检测出额外的错误。例如，CoVer 无法检查函数指针，因此无法检查这些位置的过程间控制流。这对 CoVer-Dynamic 而言并非问题，因此它能够正确报告这些路径上的错误。

与 CoVer（包括 CoVer-Dynamic）相比，MUST 能够报告更多真阳性错误。这是因为 MUST 与消息传递接口紧密集成，因此能够更深入地理解错误行为。由于无法用契约语法表示而被排除的错误类别就是一个例子，MPI-BugBench 测试中更高的真阳性率也是如此。例如，这是因为测试中包含了在缓冲区中间进行内存访问的数据竞争。由于 CoVer 不理解消息传递接口如何确定待发送缓冲区的大小（该大小由 MPI 数据类型、元素数量、步长等信息编码），因此它无法判断内存访问是否冲突，也就不会报告错误。由于 MUST 内置了对 MPI 调用语义的理解，因此它可以轻松判断是否存在冲突访问，并在需要时报告错误。

然而，这种紧密集成的缺点有两个方面，都源于由此产生的内在依赖。MUST 难以通过某些测试，有时会陷入死锁并导致超时。由于 MUST 与消息传递接口紧密耦合，因此对执行环境非常敏感。诸如消息传递接口版本不同或网络传输方式变化等因素，都会对该工具的运行能力产生显著影响。

这对 CoVer-Dynamic 并不是问题，因为其分析完全不会干扰并行编程模型。例如，MUST 通过 PMPI 接口拦截消息传递接口调用，并在内部使用消息传递接口调用来获取有关程序状态的额外信息；而 CoVer-Dynamic 则添加自己的回调函数，除了提供的合约注释之外，对消息传递接口一无所知。

此外，紧密集成有助于分析消息传递接口代码，却妨碍了将其扩展到其他并行编程模型。我们开展的 OpenSHMEM 评估就体现了这一点：由于 MUST 根本不支持它，因此无法报告任何错误。虽然也有工具可以检测 OpenSHMEM 中的数据竞争，例如 RMASanitizer [5], ，但它们同样仅适用于一组特定的已知编程模型。

CoVer(-Dynamic) 的基于契约的方法便于扩展到更多编程模型，本文以 OpenSHMEM 为例进行了演示。由于分析所需的所有信息都编码在合约注释中，因此这些扩展

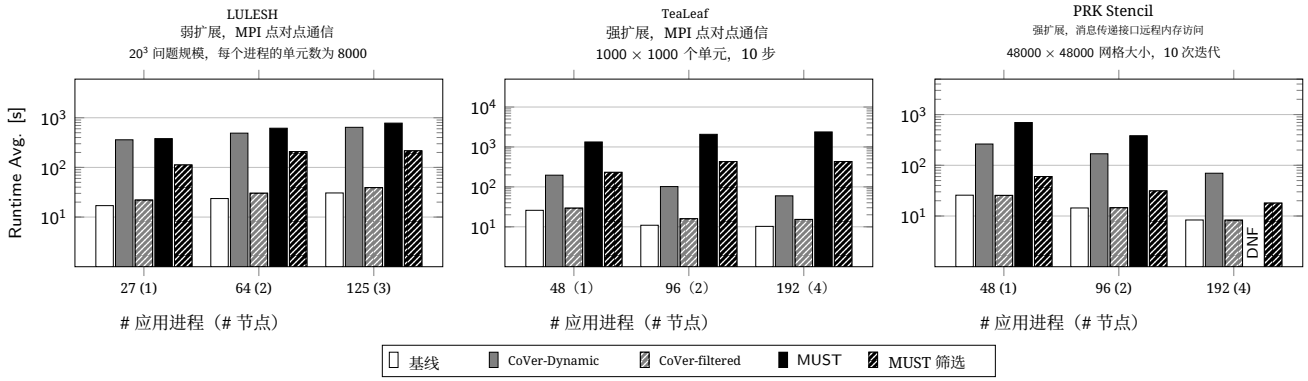

图 13：开销分析结果，包括相对于基线的倍数，时间轴采用对数刻度（未完成：未能完成，1 小时后发生死锁）。

只需为必要的 API 函数添加相应规则即可完成。

B. 开销分析

在开销分析中，我们运行了代理应用 LULESH [14],a C++冲击流体动力学模型，该模型使用非阻塞 MPI 点对点通信；还运行了 TeaLeaf [16],a C++热传导迷你应用，该应用使用集合通信和非阻塞点对点通信；以及 PRK Stencil [15],，这是一个使用 MPI RMA 通信实现简单模板核的 C 程序。

实验在计算节点上运行，这些节点配备两颗英特尔至强白金 9242 处理器（96 个核心）、384 GB 主存和 InfiniBand 互连。

每个代理应用均使用 CoVer-Dynamic 和 MUST 运行，并分别采用完整插桩和过滤式插桩；这些测试重复了五次。对于 TeaLeaf 和 Stencil，我们分别在一个、两个和四个节点上重复实验；对于 LULESH，则分别在一个、两个和三个节点上进行，因为 LULESH 只接受进程秩数量为立方数的配置。

然而，MUST 在运行死锁检测的同时进行过滤式插桩时会遇到问题，导致 MUST 自身发生死锁。因此，我们在使用过滤式插桩运行 MUST 时禁用了死锁检测。此外，在三个节点上运行未经过滤的 Stencil 时，MUST 总是发生死锁；该配置被标记为未完成（DNF）。最后，MUST 需要额外的工具进程才能运行（也可以使用线程，但运行一小时后仍超时）。因此，与基线和 CoVer-Dynamic 相比，MUST 运行时需要消耗更多计算资源。

结果绘制于图 13（对数刻度）中。所有工具都会显著拖慢代理应用的执行速度。不过，对于 CoVer-Dynamic，过滤几乎可以消除由此产生的开销，因为对每个内存位置进行插桩是导致性能减速的主要原因。对于 MUST，过滤也非常有效。但由于 MUST 在进程秩和工具进程之间有大量通信，其效果并不那么明显。PRK Stencil 的结果可以证明这一点：CoVer-Dynamic 的开销可以忽略不计。由于 CoVer 未报告错误（误报），因此没有进行内存插桩。即便如此，MUST 仍然因进程秩之间的通信而产生了较高的运行时开销。此外，与 [10] 中仅使用数据竞争检测的静态-动态耦合方法的原始测试不同，我们启用了 MUST 的大部分功能。

增加工具进程的数量可以改善 MUST 的运行时表现，但代价是在计算集群上消耗更多资源。此外，与 CoVer-Dynamic 相比，MUST 还可以运行其他分析，例如死锁检测（如前所述，过滤式插桩运行时已禁用）、针对 MPI 远程内存访问的远程数据竞争检测以及参数检查；这些分析可能会对运行时造成更大的影响。

即使没有进行静态筛选，CoVer-Dynamic 在所有测试配置下的完成速度仍显著更快。对于 TeaLeaf 和 Stencil，CoVer-Dynamic 的速度至少比 MUST 快 2×，而运行 LULESH 时则快 20%。

在 TeaLeaf 上，CoVer-Dynamic 在两个节点时的开销有所增加，在四个节点时又有所下降。考虑到 MUST 带来的异常高开销，可以肯定地说，TeaLeaf 尤其无法在我们的集群配置上良好运行，因此可视为一个异常值。

CoVer-Dynamic 和 MUST 在所有测试过的代理应用中都未报告假阳性。这正是开发动态契约分析的主要原因，因为 CoVer 确实在 LULESH 和 TeaLeaf 中报告了许多假阳性。PRK Stencil 是唯一未报告假阳性的代码。Stencil 是规模最小的代码，因此没有使用许多可能导致 CoVer 静态分析出错的复杂代码结构。

V. 相关工作

借助 MUST [4]，RMASanitizer [5]，或 ITAC [18] 等工具，可以进行动态正确性检查。在这些工具中，只有 RMASanitizer 支持消息传递接口以外的并行编程模型。这些工具都通过插桩代码并拦截运行时“已知与错误检测相关”的 API 调用，进而检查是否存在错误行为。然而，与 CoVer-Dynamic 不同，这些工具仅适用于预先已知的并行编程模型和应用程序接口。如果不修改工具本身，就无法扩展这些工具以支持不同的并行编程模型或更新的 API 调用。

除 CoVer [6] 外，其他静态工具还包括 SPMD IR [7] 和 CIVL [19] 等。CoVer 和 SPMD IR 依靠经典的数据流分析来检测编程错误，而 CIVL 则采用结合符号执行的模型检测方法。这些工具都支持多种并行编程模型。然而，除 CoVer 外，使用这些工具仍需修改工具本身才能支持其他编程模型。静态工具普遍存在的问题是误报结果风险较高，这通常是由复杂的代码结构以及静态指针别名分析的困难所导致的。

VI. 结论

在高性能计算中，分布式内存计算借助低级并行编程模型得以实现，但这些模型的使用容易出错。为此，CoVer [6] 和 MUST [20] 等正确性工具可以通过揭示数据竞争、死锁或资源生命周期问题等（非确定性）错误，帮助开发者编写和维护无错误的代码。

CoVer 通过契约语言实现与模型无关的正确性检查，用户无需开发新的分析工具即可指定应用程序编程接口使用规则。这种设计使 CoVer 能够扩展到其他（新兴的）并行编程模型。不过，其静态分析在精度方面存在局限，例如数据竞争分析。

在这项工作中，我们介绍了 CoVer-Dynamic，这是对 CoVer 现有契约框架的动态分析扩展。通过集成动态检查，CoVer-Dynamic 提高了验证准确率，同时避免了重复开发工具。动态分析减少了假阳性，并能够检测难以通过静态方式解决的错误，例如涉及指针别名或间接控制流的错误。

我们的评估将 CoVer-Dynamic 与最先进的工具 MUST 以及原始 CoVer 工具进行了比较。结果表明，CoVer-Dynamic 在标准化的消息传递接口和 OpenSHMEM 正确性基准测试中均实现了较高的分类准确率。CoVer-Dynamic 始终未报告任何假阳性，分类准确率最高可达 95%。关于运行时开销，在最坏情况下，由于必须进行内存插桩，与基线相比，测得的开销最高为 21×。尽管如此，其开销始终低于 MUST。静态与动态工具的结合能够显著降低这一开销，最高可降至基线的性能水平。不过，如前所述 [10]，我们的分类结果也证实，这可能会降低检测准确率。

最后，CoVer 现已同时具备静态分析和动态分析能力，下一步将扩展其契约语言的功能。我们的测试表明，例如，目前无法使用我们的契约语言表达死锁检测。因此，未来的工作将着重扩展对更多错误类别的支持，以拓宽该工具的适用范围。

参考文献

[1] 消息传递接口论坛，“MPI：消息传递接口标准第 4.1 版”，2023 年。 [在线]。可访问：https://www. mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[2] OpenSHMEM 委员会，“OpenSHMEM：应用程序接口第 1.5 版”，2020 年。 [在线]。可访问：http://openshmem.org/site/sites/default/site files/OpenSHMEM-1.5.pdf[3] 英伟达，“英伟达集合通信库 (NCCL) 文档——NCCL 2.25.1 文档”，2025 年。 [在 线]。可访问：https://docs.nvidia.com/deeplearning/nccl/archives/ nccl 2251/user-guide/docs/index.html [4] T. Hilbrich、M. Schulz、B. R. de Supinski 和 M. S. M¨uller，“MUST：一种在 MPI 程序中进行可扩展运行时错误检测的方法”，载于《高性能计算工具 2009》。施普林格，2010 年，第 53–66 页。[5] S. Schwitanski 、 Y. M. Oraji 、 C. P¨atzold 、 J. Jenke 、 F. Tomski 和 M. S. M¨uller，“RMASanitizer：远程内存访问应用中的数据竞争通用运行时检测”，载于第 53 届国际并行处理会议论文集，丛书 ICPP ’24。美国计算机协会，2024 年，第 833–844 页。[6] Y. M. Oraji、S. Schwitanski、A. H¨uck、J. Jenke、S. Kreutzer 和 C. Bischof，“使用契约验证 MPI API 使用要求”，载于《消息传递接口最新进展》。施普林格，2026 年，第 54–72 页。[7] S. Burak、I. R. Ivanov、J. Domke 和 M. M¨uller，“SPMD IR：统一 SPMD 与多值中间表示，并展示其在集合通信静态验证中的应用”，载于《消息传递接口最新进展》。施普林格，2025 年，第 3–20 页。[8] A. Droste、M. Kuhn 和 T. Ludwig，“MPI-checker：MPI 静态分析”，载于第二届高性能计算 LLVM 编译器基础设施研讨会论文集，丛书 LLVM ’15。美国计算机协会，2015 年，第 1–10 页。[9] S. Burak、S. Schwitanski、F. Tomski、J. Domke 和 M. M¨uller，“扩展 SPMD IR，以支持远程内存访问模型和静态数据竞争检测”，载于《消息传递接口最新进展》。施普林格，2026 年，第 143–164 页。[10] Y. M. Oraji、S. Schwitanski、S. Burak、C. Bischof 和 M. M¨uller，“耦合静态与动态 MPI 正确性工具以优化准确率和开销”，载于高性能计算、网络、存储与分析国际会议 SC ’25 研讨会论文集，丛书 SC Workshops ’25。美国计算机协会，2025 年，第 188–197 页。[11] C. Lattner 和 V. Adve，“LLVM：用于长期程序分析与转换的编译框架”，载于代码生成与优化国际研讨会，2004。CGO 2004，2004 年，第 75–86 页。[12] S. Schwitanski、J. Jenke、S. Klotz 和 M. S. M¨uller，“RMARaceBench：用于评估远程内存访问程序数据竞争检测工具的微基准测试套件”，载于高性能计算、网络、存储与分析国际会议 SC ’23 研讨会论文集，丛书 SC-W ’23。美国计算机协会，2023 年，第 205–214 页。[13] T. Jammer、E. Saillard、S. Schwitanski、J. Jenke、R. Vinayagame、A. H¨uck 和 C. Bischof，“MPI-BugBench：评估 MPI 正确性工具的框架”，载于《消息传递接口最新进展》。施普林格，2025 年，第 121–137 页。[14] I. Karlin、J. Keasler 和 J. Neely，“LULESH 2.0 更新与变更”，技术报告 LLNL-TR-641973，1090032，2013 年。[15] R. F. Van der Wijngaart 和 T. G. Mattson，“并行研究内核”，载于 2014 年 IEEE 高性能极限计算会议 (HPEC)，2014 年，第 1–6 页。[16] “TeaLeaf”，布里斯托大学高性能计算小组，2025 年。 [在线]。可访问：https://github.com/UoB-HPC/TeaLeaf [17] Y. M. Oraji，“《并行编程模型的动态契约验证》配套材料。” [在线]。可访问：https://doi.org/10.5281/zenodo.18152064[18] P. Ohly 和 W. Krotz-Vogel，“MPI 自动化正确性检查：如果有一个神奇选项会怎样？”，载于第 8 届 LCI 高性能集群计算国际会议论文集，2007 年，第 19–25 页。[19] S. F. Siegel、M. Zheng、Z. Luo、T. K. Zirkel、A. V. Marianiello、J. G. Edenhofner、M. B. Dwyer 和 M. S. Rogers，“CIVL：并发中间验证语言”，载于高性能计算、网络、存储与分析国际会议论文集，丛书 SC ’15。美国计算机协会，2015 年，第 1–12 页。[20] T. Hilbrich、J. Protze、M. Schulz、B. R. de Supinski 和 M. S. M¨uller，“使用 MUST 进行 MPI 运行时错误检测：死锁检测的进展”，《科学编程》，第 21 卷，第 3–4 期，第 109–121 页，2013 年。