arXiv:2303.13954v1 [cs.DC] 24 Mar 2023

PGAS 程序中不规则内存访问模式的编译器优化

Thomas B. Rolinger[1], [2]✉Christopher D. Krieger[2],和 Alan Sussman[1]

[1] 美国马里兰大学，科利奇帕克，马里兰州，美国{tbrolin, als}@cs.umd.edu[2] 物理科学实验室，科利奇帕克，马里兰州，美国 krieger@lps.umd.edu✉ 通讯作者：tbrolin@cs.umd.edu

摘要。 不规则内存访问模式给分布式内存系统带来了性能和用户生产力方面的挑战。它们可能导致细粒度远程通信，而且访问模式往往只有到运行时才能确定。分区全局地址空间（PGAS）编程模型通过为用户提供一种类似于单一共享地址空间的分布式内存系统视图，应对这些挑战。然而，这种视图往往会使程序员编写出导致细粒度远程通信的代码，从而造成性能不佳。以往研究表明，通过手动应用优化，可以提升使用高级 PGAS 语言 Chapel 编写的不规则应用的性能。然而，手动应用此类优化会削弱 Chapel 和 PGAS 模型所带来的生产力优势。我们提出了一种基于检查器-执行器的编译器优化，可自动为 Chapel 程序执行远程数据复制。尽管其他 PGAS 语言也已有类似的编译器优化，但 Chapel 的高级特性（例如隐式处理器亲和性）为编译器优化带来了新的挑战。我们在两个不规则应用上评估了该优化的性能。结果表明，在采用低延迟互连的 Cray XC 系统上，总运行时间最多可缩短 52 倍；在采用 InfiniBand 互连的标准 Linux 集群上，最多可缩短 364 倍。这表明，在不牺牲用户生产力的前提下，也能实现显著的性能提升。</output>

关键词： 分区全局地址空间、Chapel、不规则应用、编译器优化

1 引言

实现能够有效利用分布式内存系统的并行软件，对程序员而言面临诸多挑战。具体来说，要将现有的串行应用程序或共享内存并行化应用程序改造为可在分布式环境中运行，通常需要程序员投入大量精力来协调数据分布和通信。分区全局地址空间（PGAS）模型

试图通过为程序员提供一种视图来应对这些挑战：在这种视图中，分布式内存系统看起来就像一个共享地址空间。PGAS 模型已在多种语言和库中实现，例如 UPC [11], GlobalArrays [14] 和 Chapel [6]。在 PGAS 语言中，数据分布和通信的相关细节通常会对程序员隐藏起来。例如，在 Chapel 中，只需将数组指定为“分布式”，数据便会自动映射到系统各处，并隐式执行远程通信。因此，PGAS 模型鼓励程序员以共享内存的方式编写代码，同时旨在确保代码在分布式内存系统上具有良好性能，而无需重写程序。

不规则内存访问模式常见于执行图分析 [13], 稀疏线性代数 [20] 和科学计算操作 [10]的应用中。这类访问模式给分布式内存系统上的用户生产力带来了重大挑战，因为直到运行时才能确定访问模式，这使得协调远程进程之间的通信变得困难。不过，借助分区全局地址空间模型，编码不规则通信模式会更加简单，因为可以通过单边通信来实现。不幸的是，由于不规则内存访问会产生细粒度的远程通信，这类代码的性能会受到显著影响。虽然可以手动绕过分区全局地址空间模型提供的抽象以获得更好的性能，但这样做会大幅降低用户生产力。

本文介绍了一种编译器优化的设计与实现，该优化可自动将 检查器-执行器技术 [16] 应用于使用 Chapel 语言编写的 PGAS 程序中的并行循环。检查器会在运行时执行内存访问分析，以确定循环中对目标数组的远程通信需求。执行器会复制远程数据并运行原始循环，同时将远程访问重定向至复制的本地副本，以避免重复的远程通信。我们的编译器优化会自动识别候选循环和数组访问，然后执行代码变换来构建检查器和执行器例程。因此，用户无需修改原始代码即可获得显著的性能提升。我们的工作贡献如下：

– 为 Chapel 程序设计并实现了一种基于检查器-执行器技术的编译器优化，专门针对分布式数组的不规则内存访问。据我们所知，这是 Chapel 编译器中首个此类优化。– 探讨了 Chapel 的独特特性（例如隐式处理器亲和性）与该编译器优化的关系。尽管检查器-执行器技术早已应用于其他分区全局地址空间语言 [16,15,17], [18,1], ，但由于 Chapel 的高级特性，我们在 Chapel 中的设计需要采用不同的方法。– 在两种不规则应用和两种不同的分布式内存系统上评估了该优化的性能。结果表明，在采用低延迟互连的 Cray XC

系统上，性能最高可提升 52 倍；在采用 InfiniBand 互连的标准 Linux 集群上，最高可提升 364 倍。

本文其余部分的组织如下：第 2 节概述 Chapel。第 3 节介绍我们的编译器优化并讨论其细节。第 4 节评估该优化在两个不规则应用中的性能，附录则提供更多性能结果。第 5 节介绍与本文相关的先前工作。最后，第 6 节总结全文并讨论未来工作。

2 Chapel 语言概述

Chapel 是一种实现分区全局地址空间模型的高级语言，旨在支持大规模、高效的并行计算，并提供分布式数组、远程通信以及数据并行和任务并行的构造。本节简要概述 Chapel，重点介绍与我们工作最相关的特性。关于 Chapel 更深入的介绍，请读者参阅 Chamberlain 等人的工作 [6]。

2.1 术语：任务、线程与 locale

Chapel 通过并行执行多个任务 来实现并行度。任务是一组概念上可以并行执行、但实际上可能并行执行也可能不并行执行的计算。任务由任务调度层实现，该层提供线程以供调度 Chapel 任务。对于分布式内存编程，Chapel 引入了*locale*这一概念，它被定义为任务可以在其上执行的一组机器资源。实践中，一个 locale 对应集群中的一个完整计算节点；程序启动时会指定程序运行所用的 locale 数量。Chapel 程序最初会在 locale 0 上启动一个任务。随后，并行循环和其他构造会创建可在各个 locale 上执行的任务；这些任务完成后，最终会汇合为 locale 0 上的单个任务。这不同于 UPC [11], 等其他分区全局地址空间语言；它们采用单程序多数据（SPMD）模型，在程序启动时确定并行度。

2.2 域和数组

Chapel 将数组拆分为语言中的两个一等对象：域和数组本身。域是索引集的一种表示，可用于定义数组中的索引，或循环迭代的索引。域定义完成后，可用于声明数组。对域的修改（例如添加或删除索引）会传播到所有基于该域定义的数组。清单 1 第 1–2 行展示了一个简单示例：定义域 D，使其包含从 0 到 5 的索引，然后在该域上声明一个整数数组 data。与我们的工作尤为相关的是关联域，它类似于

字典。清单 1 第 4–7 行声明了一个存储字符串的关联域 C，并通过 += 运算符添加键“foo”。关联数组 dict 在 C 上声明，提供从字符串到实数的映射。关联域还默认支持并行安全修改。

```
1 var D = {0..5};// 矩形域2 var data : [D]int;34
va
di
```

清单 1: Chapel 的域和数组

```
1 变量 D =
n(
ar
ar
```

清单 2: Chapel 中的块分布域/数组

清单 1 中的代码仅声明了位于单个 locale 上的域和数组。我们的工作聚焦于分布式数组，其数据根据某种分布策略分散在多个 locale 上。Chapel 提供了若干内置分布策略，例如块分布、循环分布和块循环分布。清单 2 展示了一个简单示例，其中声明了一个块分布域以及与之对应的分布式数组。块分布会将数组划分为连续数据块，并将每个数据块分配给一个 locale。底层分布实现会自动处理索引重映射，使用户可以通过 arr[i] 访问数组元素，而无需指定该索引属于哪个块。此外，Chapel 会执行隐式远程通信，这意味着用户访问分布式数组的远程元素时，与访问本地元素的方式相同。清单 2 第 4–6 行展示了这一点：这些行将 arr[i] 的值设为执行该任务的 locale 的 ID（here.id）。循环将在 locale 0 上执行所有迭代，因此，访问任何不在 locale 0 上的 arr 元素都会导致通信。

2.3 Forall 循环

Chapel 通过 forall 循环提供数据并行，该循环允许将循环迭代并行化并分布到整个系统中。迭代如何映射到核心/locale，取决于 forall 循环迭代的对象，即迭代对象。非分布式数组和域具有

默认定义的迭代器 会将迭代划分为连续数据块，并将每个数据块分配给一个任务。随后，这些任务会在单个 locale 上并发执行。对于分布式数组和域，默认迭代器通过在存储该次迭代数据元素的 locale 上执行循环的相应迭代，同时提供共享内存和分布式内存并行。请看列表 3 中的示例，该示例遍历分布式数组 arr 的域。此循环类似于列表 2 第 4–6 行的非并行循环，但它会在存储 arr[i] 的 locale 上执行 i^{th} 迭代。因此，此处的 id 将返回存储 arr[i] 的 locale 标识符。这样可以避免列表 2 中的代码会产生的远程通信。因此，forall 循环的迭代对象会隐式控制计算的执行位置。我们将此称为控制 forall 的区域亲和性，并指出这是 Chapel 的一项特性，与 UPC 等语言中更显式的亲和性控制结构不同。此外，用户可以定义自己的自定义迭代器，以改变 forall 循环的并行化方式。

```
1 var D = newBlockDom({0..15});
2 变量 arr: [D]整数;
3 forall i 属于 arr.domain{
      arr[i] = 这里.id;
4
5 }
```

列表 3: Chapel 中的 forall 循环

3 编译器优化

本节介绍针对不规则内存访问的检查器-执行器编译器优化的设计与实现。我们特别关注 forall 循环中形如**A**[**B**[i]] 的只读访问，其中**A** 和**B** 是数组，且**A** 是分布式数组。不过，如果满足某些条件，我们也支持更复杂的非仿射表达式（见第 3.3 节）。该优化的总体目标是有选择地复制**A** 中被远程访问的元素，使其在循环执行期间能够在本地访问。从内存消耗和通信开销两方面来看，完全复制**A** 的代价都可能高得难以承受；然而，只有在循环执行时才能得知哪些元素会被远程访问。因此，我们设计了这种编译器优化，在运行时创建检查器，以确定哪些访问是远程访问。

该优化针对执行多次的 forall 循环，这样检查器阶段的开销便可分摊到优化后循环（即执行器）的多次执行中。这类模式常见于稀疏迭代求解器 [10], 分子动力学模拟 [17]以及某些图分析应用 [5]。图 1 概述了 Chapel 编译器遍次的高层结构，虚线框内阴影标出的遍次对应于该优化执行静态分析和代码变换的阶段。

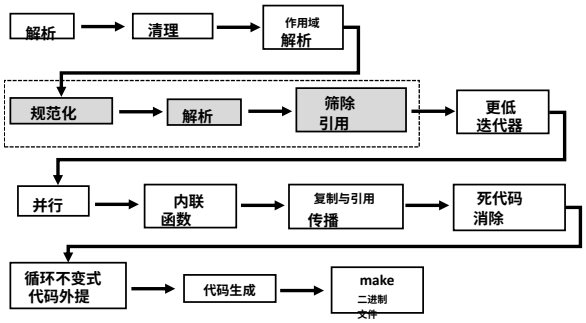

图 1： Chapel 编译器的高层概览。虚线框内的阴影阶段表示我们的优化会在这些阶段执行代码变换和静态分析。

截至目前，Chapel 编译器大约有 40 个遍次，但为简化图示，图 1 省略了其中大多数。

3.1 高层概览

我们的优化仅作用于 Chapel 程序的中间表示，我们称之为抽象语法树（AST）。优化过程始于规范化阶段，此时 AST 尚未经过大量修改，因此更容易执行代码变换任务。优化会检查程序中的每个 forall 循环，并查找形式为 **A**[**B**[i]] 的访问。不过，数组访问在 AST 中没有专属语法，这意味着 **A**[**B**[i]] 会被表示为调用表达式，无法与函数调用区分。为解决这一问题，不规则访问候选表达式会被替换为编译器原语，并在解析阶段进行处理；届时，优化便能确定**A** 和**B** 是否为数组。如果 forall 循环中存在候选访问，优化就会将该 forall 循环克隆为检查器循环和执行器循环。

在编译的解析阶段，会解析函数调用和类型，使优化能够执行大量必要的静态分析，以确保在规范化阶段执行的代码变换有效。如果发现任何问题会导致优化无效，则会移除代码变换，并将其替换为原始的 forall 循环。否则，表示 **A**[**B**[i]] 的原语会被替换为库调用，以检查或执行该访问，具体取决于该原语位于检查器循环还是执行器循环中。哪些访问属于远程访问由检查器在运行时确定。优化所执行的静态分析仅确定该访问的形式为 **A**[**B**[i]]，且 **A** 是一个分布式数组。

最后，在剔除引用阶段结束时，优化会执行剩余的分析，以确保这些转换有效；分析重点是定位对不规则访问所在循环中相关数组和域的修改（写入）。

这类写入可能会改变**A**[**B**[**i**]] 访问模式，因此需要重新运行检查器运行时分析，以确保执行器复制了正确的元素。剔除引用阶段会解析函数参数的意图，参数的意图指的是该参数是按值传递还是按引用传递。此阶段会根据过程是否读取或写入参数，在静态分析中将参数的意图设为不同的值。数组的写入通过 Chapel 过程调用来执行，数组作为参数传入；这使优化能够检查数组的意图，从而确定数组是否会被写入。

3.2 代码转换

清单 5 展示的代码在功能上等同于将我们的优化的 AST 级转换应用于清单 4 中的程序后得到的输出。接下来，我们将介绍代码转换的具体细节，以及它们如何支持选择性数据复制。

```
1 全体i 属于B.域{
2   C[i] = A[B[i]];
3 }
```

清单 4： 具有不规则内存访问模式的 forall 循环示例

```
    1 如果 doInspector(A,B){2
in
in
ins
ins
所
11
```

代码清单 5：优化对代码清单 4 中的代码执行代码转换后的输出

Chapel 内部数据结构：对于正在优化的 forall 循环中的分布式数组 **A**，我们会为每个 locale 添加一个 Chapel 记录（即 C 结构体），其中包含该 locale 上有关 **A**[**B**[**i**]] 的远程通信信息。我们将这些记录的完整集合称为通信计划。通信计划本质上是一组关联数组，在给定 locale 发起的**A**[**B**[**i**]] 属于远程访问时，这些数组会将**B**[**i**] 映射到**A**[**B**[**i**]] 。如果**A** 关联了多个正在优化的 forall 循环，优化过程会为**A** 创建不同的通信计划，并将其分别关联到每个 forall 循环。

检查器循环：编译器优化首先将原始 forall 循环克隆为两个副本，一个将转换为检查器，另一个将转换为执行器。对于检查器循环（第 3–5 行），优化过程会将原始 forall 循环的迭代器替换为自定义迭代器（inspectorIterator）。这个自定义迭代器会在每个 locale 上创建一个任务，串行执行原始 forall 循环中属于该 locale 的部分。尽管这会降低默认迭代器提供的总体并行度，但能够关闭通信计划中底层关联数组的并行安全性检查，通常可以提升性能。生成循环结构后，优化过程会将原始表达式 C[i] = A[B[i]] 替换为对 inspectAccess 的调用；我们创建此过程来执行内存访问分析。请注意，原始循环中任何与**A**[**B**[i]] 访问无关的代码都会被移除，以尽可能精简检查器。此外，inspectAccess 的调用不会实际访问**A**，而是查询**B**[i] 是否会对**A**进行远程访问，因为远程访问开销很大。最后，优化过程会在检查器循环之前插入对 inspectorPreamble 的调用（第 2 行），以初始化部分通信调度内部结构。

启用/关闭检查器：优化的关键步骤之一是确定检查器何时应执行。检查器首次运行后，除非内存访问模式发生变化，否则无需再次运行。静态分析阶段（见第 3.3 节）会确定此类变化何时发生，而优化会插入对某个过程的调用，以设置与 **A** 和 **B** 关联的标志，指示应执行检查器。第 2 行的 doInspector 调用会检查这些标志，第 6 行的 inspectorOff 调用则会关闭这些标志。

执行器循环：对于执行器循环，优化会将原始的 **A**[**B**[i]] 访问替换为对 executeAccess 的调用（第 10 行），将远程访问重定向到副本。优化还会在执行器循环之前插入对 executorPreamble 的调用（第 8 行），用 **A**中的当前值初始化复制元素，确保执行器使用最新的值。虽然这确实会产生远程通信，但无论在循环中访问某个远程元素多少次，优化都只会对其通信一次。因此，远程访问的开销会分摊到多次本地访问中。

3.3 静态分析

静态分析有两个主要目标：（1）检测代码转换可能导致程序结果与原始程序不同的情形；（2）仅在可能获得性能提升时才应用优化。每个目标都提出了不同的要求，这些要求必须在编译时解析；若任何一项要求未得到满足，优化就会回退到原始代码。我们简要介绍为实现这两个目标而执行的分析，随后讨论为提高优化通用性而开发的其他分析。

程序结果： 由于优化由编译器自动应用，因此必须检测代码转换可能产生与原始代码不同的程序结果的情形。可能导致程序结果不同的主要情形，是在必要时未运行检查器，从而导致通信计划过时。为了确定检查器何时需要执行，优化必须能够静态推理 forall 循环的区域亲和性，以及远程访问的源 locale 和目标 locale，适用于 **A**[**B**[i]]。以下标准概述了为此执行的静态检查：

1. forall 循环必须遍历分布式数组或分布式域。2. forall 循环不能嵌套在另一个 forall 循环内，也不能嵌套在任何可能创建多个并行任务的其他语句内。3. **i** 在**A**[**B**[i]] 中的索引必须是包含**A**[**B**[i]],的循环的循环索引变量，并且该循环必须遍历域或数组。4. 在 forall 循环内，既不能修改**A** 或**B**，也不能修改它们的域。

检查（1）确保优化能够分析 forall 循环的区域亲和性（即各次迭代将在哪些 locale 上执行）。检查（2）确保不会有多个任务同时执行整个 forall 循环，否则可能导致通信计划发生并发更新。检查（3）确保优化能够静态分析**i** 对**B**的索引访问，尤其是**i** 的值何时会改变。如果循环所遍历的数组/域被修改，就会发生此类变化。检查（4）确保检查器分析的**B** 中的值，与执行器（以及原始循环）中使用的值相同。将此检查应用于**A**，可以避免向**A**的复制元素写入所带来的复杂性，以及随后必须将这些值传播回原始元素的问题。如果满足这些检查条件，优化就会确定对**A**、**B**、它们的域，以及循环中涉及的其他各种数组/域所做的所有修改。如果这些对象被修改，就表明**A**[**B**[i]] 可能呈现出不同于上次执行检查器时的访问模式；变化可能体现在访问**A** 时使用了不同的索引，也可能体现在访问操作由不同的 locale 发起。发现修改后，优化会设置相应的标志，以便重新运行检查器并更新通信计划。

程序性能：优化的目标是提升输入程序的运行时性能。因此，静态分析会尝试判断该优化是否可能带来性能提升。此分析可概括为：确定 forall 循环是否会执行多次，而无须每次都执行检查器，否则将产生显著开销。以下标准概述了为此执行的静态检查：

(a) forall 循环必须嵌套在外层串行循环中（即 for、while 等）。（b）在 forall 循环所嵌套的外层循环中，不能修改**B** 及其域。

(c) 在 forall 循环所嵌套的外层循环中，不能修改**A**的域。

检查 (a) 可确保 forall 循环很可能会执行多次，但无法保证一定如此。检查 (b) 和 (c) 可确保每次运行 forall 循环时都不会执行检查器。请记住，代码转换阶段的一部分是在对**B**、其域或**A**的域进行修改后“启用”检查器。此类修改可能会改变**A**[**B**[i]], 的内存访问模式，因此需要再次执行检查器。允许在外层循环中修改数组**A** ，因为其值的任何变化都会通过 executorPreamble 传播到副本中。不过，修改**A**的域可能会通过在**A**中添加或删除元素，改变访问模式**A**[**B**[i]] 。

非仿射表达式分析： 到目前为止，我们主要讨论的是**A**[**B**[i]]形式的访问。不过，只要满足某些条件，该优化也支持更复杂的非仿射表达式，例如**A**[**B**[**i*j**]%**k**+**1**]。具体而言，这些表达式必须是立即数之间的二元运算，或是由遍历数组/域的循环产生的变量之间的二元运算。此要求是为了让优化能够静态分析对**A** 和**B** 的访问何时以及如何发生变化。实现方法是查找对产生表达式中所用变量的数组/域所做的修改。请注意，除本优化之外，Chapel 当前的编译器不会进行仿射/非仿射表达式分析。

过程间分析与别名分析：该优化执行过程间分析与别名分析，以支持前文所述的静态检查。过程间分析从包含 forall 的函数开始计算调用图，并检查是否存在无效调用路径。如果缺少外层串行循环，或者存在会创建多个并行任务的外围语句，我们就将该调用路径视为无效。检测到无效调用路径时，编译器会插入标志，这些标志在运行时设置，以便沿无效路径暂时“关闭”优化。此外，过程间分析还会检测任意嵌套的函数调用过程中对相关数组/域的修改。别名分析是必要的，因为 Chapel 允许用户创建数组/域的引用，其行为类似于 C 中的指针。为解决这一问题，我们开发了静态检查，用于检测此类引用并确定原始数组/域。该分析支持任意长度的别名链（例如，`var arr = ...; ref a1 = arr; ref a2 = a1`），优化会检测链中任意引用上的修改。

4 性能评估

为展示第 3 节所述编译器优化带来的性能收益，我们评估了两个不同分布式内存系统上运行的两个不规则应用。这些应用采用高层次方式实现，符合 Chapel 的设计理念：将数据分布和通信细节与

算法设计分离。因此，我们的示例应用与标准共享内存实现非常相似，也能代表分区全局地址空间模型的直接使用。我们的目标是展示，这些 Chapel 程序的性能会因隐式的细粒度远程通信而受损，但通过自动优化可以显著改善，且无需用户修改程序。这样，用户既能享受 Chapel 带来的高效开发优势，也能获得良好的性能。

4.1 实验设置

为了进行评估，我们在 Cray XC 集群和基于 InfiniBand 的集群上开展实验。对于 Cray XC，我们使用最多 64 个通过 Aries 互连网络连接的节点，每个节点配备两颗 22 核 Intel Xeon Broadwell处理器和 128 GB DDR4内存。对于 InfiniBand 集群，我们使用最多 32 个通过 FDR InfiniBand互连连接的节点，每个节点配备两颗 10 核 Intel Xeon Haswell处理器和 512 GB DDR4内存。在 Cray XC 上，Chapel 使用 ugni通信层和 Aries 通信基底构建。在 InfiniBand 系统上，Chapel 使用 GASNet通信层和 ibv通信基底构建。两个系统上的所有应用程序均使用 ---fast 标志编译。对于每项实验，我们多次运行给定的应用程序，并测量总运行时，其中包括检查器开销。结果为这些试验的平均值。我们观察到，各次试验之间的运行时变化不超过 4%。

4.2 应用：NAS-CG

```
1 forall行 属于Rows {2 变量 累加器 : 实数 = 0;
3
va
]
```

代码清单 6: NAS-CG 的 forall 循环

共轭梯度（CG）方法用于求解方程 $Ax = b$，其中 A 是一个对称正定矩阵，通常规模较大且稀疏；x 和 b 是向量。非结构化优化问题和偏微分方程可以使用迭代共轭梯度法求解。在评估中，我们采用 NAS-CG 基准测试规范和数据集 [3]。表 1 描述了所评估的问题规模，其中每种问题规模都对应于 A 矩阵的大小。NAS-CG 的每次迭代总共执行 26 次稀疏矩阵-向量乘法（SpMV），这是优化所关注的内核，详见代码清单 6。该实现使用标准的压缩稀疏行（CSR）格式表示 A，其中 Rows 是一个块分布式记录数组

这些记录包含指向存储非零数据值的分布式数组的偏移量。这种方法与 NAS-CG 的 Fortran+OpenMP 实现 [3]非常相似。我们关注的不规则访问位于第 4 行：x[col idx[k]]。对于优化，检查器只需执行一次，因为所有 SpMV 操作的内存访问模式都保持不变。关于复制带来的内存存储开销，我们观察到内存使用量平均增加了 6%。

表 1： NAS-CG 数据集

名称	行数	非零元素	密度（%）	稀疏矩阵-向量乘法次数
C	150k	39M	0.17	1950
D	150k	73M	0.32	2600
E	9M	6.6B	0.008	2600
F	54M	55B	0.002	2600

表 2： 针对表 1 中 NAS-CG 问题规模进行优化后，相较于代码清单 6 中未经优化的代码所实现的运行时加速比。缺失值表示该问题规模所需内存过大，无法执行；“不适用”值表示系统不支持指定数量的节点/locale。

locale（节点）	Cray XC C	Cray XC D	Cray XC E	Cray XC F	InfiniBand C	InfiniBand D	InfiniBand E	InfiniBand F
2	3.2	2.8	—	—	8.9	6	357	—
4	3.6	3.4	17.5	—	15.8	10.4	345	—
8	5.7	6.2	36.7	—	115	127	364	—
16	8.6	11	22.5	—	238	330	258	270
32	6.4	8.4	34	52.3	160	240	195	165
64	4.1	4.9	16.7	25.4	NA	NA	不适用	不适用
几何平均值	5	5.5	24.1	36.4	57.3	57.5	296	211

表 2 展示了相对于代码清单 6 中所示未优化代码，优化在各系统上实现的 NAS-CG 运行时加速比。我们观察到两个系统上都获得了显著加速，但 InfiniBand 系统上的加速尤为明显，因为与 Cray XC 上的 Aries 互连网络相比，细粒度远程通信的延迟更高。在两个系统上，这些加速都源于稀疏矩阵-向量乘法内核中大量的远程数据复用，这归因于基准测试生成矩阵的稀疏模式。该优化只需读取一次远程元素，之后在 forall 循环的其余部分中便可在本地访问该元素。若不进行优化，由于访问模式稀疏且不规则，每次访问远程元素很可能都要承担远程访问的完整延迟开销。此外，在 NAS-CG 基准测试期间，稀疏矩阵-向量乘法内核会执行多次，因此检查器开销得以摊销。我们观察到，在所有 locale 数量下，检查器所占运行时间的比例在 Cray XC 上平均为 3%，在 InfiniBand 系统上为 2%。

除了相对加速比之外，这项优化还显著缩短了总体运行时间，如附录中的表 5 和表 6 所示。我们观察到性能很差

未优化代码的运行时性能和可扩展性都很差，这是因为代码清单 6 所示的直接实现需要隐式地进行细粒度远程通信。另一方面，随着 locale 数量增加，优化后的代码通常会运行得更快，因为该优化能够利用远程数据复用。总体而言，这些结果表明，我们的优化很有用：它能够自动缩短运行时间，同时不牺牲用户生产力。

4.3 应用：网页排名

```
1 对所有v 属于 Graph{2 变量 val  = 0.0;3 对于 i属于v.offsets{4 引用
t
v.
```

列表 7： 网页排名的 forall 循环

网页排名 [5] 是一种迭代图算法，可为图中的每个顶点提供重要性度量。列表 7 展示了网页排名内核，也是优化的目标，其中关注的不规则内存访问发生在第 4 行。与 NAS-CG 类似，我们使用 CSR 数据结构表示图。分布式数组 Graph 存储与顶点对应的记录，每个顶点有两个重要性度量：pr write 和 pr read。这使得内核可以通过在一次迭代期间将一个值视为只读，实现直接的并行处理，并且与 GAP 基准测试套件的实现 [4]十分接近。与 NAS-CG 不同，网页排名还增加了一项复杂性：在目标数组中存储的是记录，而非基本类型数据（即 int、real 等）。该优化会识别这一特点，并且只复制 forall 循环中访问的字段，即 pr read 和 out degree。

我们在 SuiteSparse 矩阵集合 [8], 中的两个真实网页图上评估网页排名，相关信息见表 3。最右侧一列表示在容差值为 $1e$-7、阻尼因子为 0.85（清单 7 第 7 行中的 d）的条件下收敛所需的迭代次数。我们选择这些数值，是为了与 Neo4j [19], 这一开源图数据库所采用的数值保持一致。每次网页排名迭代都会执行一次清单 7 中整个 forall 的操作；由于执行期间图不会发生变化，检查器只需执行一次。网页排名优化的内存存储开销为 40–80%，远高于我们在 NAS-CG 中观察到的开销。对于 NAS-CG，所关注的复制数组仅占所需总内存的一小部分。而对于网页排名，相比之下所关注的数组要大得多，因此内存的相对增幅也更大

存储开销。不过，该优化带来的内存存储开销显著低于完全复制所需的开销。

表 3： 网页排名的数据集

Name	顶点数	边数	密度（%）	迭代次数
webbase-2001	118M	992M	7.1e−6	33
sk-2005	51M	1.9B	7.5e−5	40

表4： 表 3 中各图在网页排名中的优化运行时加速比（相对于未优化实现）。「不适用」表示系统不支持指定数量的节点/locale。

locale（节点）	Cray XC		InfiniBand	
	webbase-2001	sk-2005	webbase-2001	sk-2005
2	0.88	1.2	5.2	2
4	0.98	1.6	8.6	7.1
8	0.97	1.3	12	6
16	0.94	1.7	9.6	5.4
32	1.3	1.4	4.5	4.2
64	1.2	2.1	NA	NA
几何平均值	1.04	1.5	7.3	4.5

表 4 展示了表 3 中两个图应用该优化后的运行时加速比，附录中的表7和表8则列出了未应用和应用该优化时的运行时间。与 NAS-CG 的观察结果一样，由于 Aries 互连网络，InfiniBand 系统上的加速比高于 Cray XC。不过，与 NAS-CG 相比，整体加速比明显较低。这主要是因为 PageRank 内核的执行次数少于 NAS-CG 内核，而且与 NAS-CG 相比，这些图的数据重用明显较少。因此，检查器开销更大，执行器带来的性能提升也更小。基于这些原因，在 Cray XC 上，webbase-2001 图直到使用 32 个 locale 时才实现加速，此时远程数据复用达到峰值。此外，由于这些图具有高度不规则的特性（其度分布遵循幂律），随着 locale 数量增加，运行时性能会因图在系统中的分区方式而波动。分区方式可能改变元素所在的位置，导致原本频繁访问的远程元素变为本地元素。对于未优化的代码，这种影响更显著，因为优化通过复制只需承担一次远程访问的开销。尽管如此，该优化的可扩展性总体上与未应用优化的代码的可扩展性相当。

5 相关工作

多年来，人们一直在研究用于不规则内存访问的运行时优化技术，其中与我们的工作尤其相关的是 检查器-执行器 技术 [16,9,17]。Das 等人 [7] 提出了一种与我们类似的检查器-执行器优化，但这项工作早于分区全局地址空间模型，因此由于编程模型设计和实现存在根本差异，无法直接应用。对于分区全局地址空间语言，Su 和 Yelick [18] 开发了一种与我们类似的检查器-执行器优化，但针对的是 Titanium 语言。不过，Titanium 基于 Java，其执行模型以及用于并行循环的语言构造都与 Chapel 不同。因此，静态分析的整体方法也有所不同。Alvanos 等人 [1,2]介绍了一个用于分区全局地址空间语言 UPC [11],的检查器-执行器框架，该语言同样与 Chapel 不同。UPC 使用单程序多数据模型，并要求使用显式构造来控制并行循环的处理器亲和性。与 Titanium 类似，这些差异使其优化所采用的静态分析方法显著不同。不过，我们计划在未来的工作中探索他们的一些技术。针对 Chapel 的编译器优化方面，Kayraklioglu 等人 [12] 提出了一种优化方法，用于聚合对分布式数组的远程访问。不过，他们的优化并未专门针对不规则内存访问，因此静态分析和代码变换的方法显著不同。此外，第 4 节中评估的应用并不适用于他们的聚合优化。

6 结论与未来工作

尽管分区全局地址空间模型为编写分布式不规则应用提供了用户生产力方面的优势，但由此产生的代码往往因细粒度远程通信而运行时性能不佳。在本研究中，我们介绍了一种针对 Chapel 程序的编译器优化，专门处理并行循环中的不规则内存访问模式，并自动应用代码变换来复制远程访问的数据。我们证明，该优化在 Cray XC 系统上可实现最高 52 倍的运行时加速，在 InfiniBand 系统上可实现最高 364 倍的加速。为此，我们表明，在不牺牲用户生产力的前提下，也能获得显著的性能提升。

未来，我们计划改进编译器优化框架，并解决其中的一些局限，例如优化同一循环中的多个不规则访问。我们还计划设计其他编译器优化，以便在无法应用选择性数据复制时提供替代方案；目前，选择性数据复制仅适用于只读数据。未来的优化将专门针对写操作。

致谢

我们谨此感谢 Chapel 团队的 Brad Chamberlain、Engin Kayraklioglu、Vass Litvinov、Elliot Ronaghan 和 Michelle Strout 在使用 Chapel 编译器方面给予的指导

以及提供 Cray XC 系统的使用权限，该系统用于我们的性能评估。

附录

本附录提供了第 4 节实验的更多性能结果。表 5 展示了 NAS-CG 未优化实现的执行运行时间，表 6 展示了 NAS-CG 自动优化实现的运行时间。由于未优化代码处理问题规模 E 和 F 所需的时间过长，我们根据平均迭代运行时间推算其总运行时间。NAS-CG 的每次迭代都执行相同的计算和通信；我们注意到，在问题规模 C 和 D 上，各次迭代之间的差异不超过 2%。表7和表8展示了 PageRank 应用程序的类似数据。

表 5： 在表 1 所列问题规模上运行 NAS-CG 未优化实现的执行运行时间（分钟）。缺失值表示问题规模所需内存过大，无法执行；“不适用”表示系统不支持指定数量的节点/locale。问题规模 E 和 F 的运行时间根据单次迭代运行时间推算。

	Cray XC				**InfiniBand**			
locale（节点）	C	D	E	F	C	D	E	F
2	1.8	3.9	—	—	9	15	2.3e+5	—
4	1.7	3.4	2882	—	12	17	1。7e+5	—
8	2.2	4.8	3069	—	58	131	1e+5	—
16	2.7	6.1	1005	—	96	242	5.7e+4	4.7e+5
32	1.8	4	845	9927	64	156	3。1e+4	2.4e+5
64	1.2	2.6	285	3501	不适用	不适用	NA	NA

表 6： 在表 1 所列的问题规模上执行时，NAS-CG 自动优化实现的运行时间（分钟）。缺失值表示执行该问题规模需要的内存过多；“不适用”值表示系统不支持指定数量的节点/locale。问题规模 E 和 F 的运行时间根据单次迭代运行时间推算得出。

	Cray XC				**InfiniBand**			
Locale（节点）	C	D	E	F	C	D	E	F
2	0.6	1.4	—	—	1.1	2.6	655	—
4	0.5	1	165	—	0.7	1.7	489	—
8	0.4	0.8	84	—	0.5	1	279	—
16	0.3	0.6	45	—	0.4	0.7	221	1732
32	0.3	0.5	25	190	0.4	0.6	156	1437
64	0.3	0.5	17	138	不适用	不适用	不适用	NA

表 7： 使用表 3 中的图执行未优化的网页排名实现时的执行运行时间（分钟）。“不适用”表示系统不支持指定数量的节点/locale。

locale（节点）	**Cray XC** webbase-2001	sk-2005	**InfiniBand** webbase-2001	sk-2005
2	1.1	4.1	14.6	14
4	0.9	3.8	14.6	28
8	0.7	4.4	14.9	23
16	0.6	7.3	9.7	29
32	0.6	5.7	6.9	26
64	0.7	10	NA	NA

表 8： 在表 3 中的图上运行自动优化的网页排名实现时的执行运行时间（分钟）。“不适用”表示系统不支持指定数量的节点/locale。

Locale（节点）	**Cray XC** webbase-2001	sk-2005	**InfiniBand** webbase-2001	sk-2005
2	1.2	3.3	2.8	6.9
4	1	2.4	1.7	3.9
8	0.8	3.4	1.2	3.9
16	0.7	4.2	1.1	5.5
32	0.5	4.2	1.5	6.2
64	0.6	4.7	NA	NA

引用

1. Alvanos, M., Farreras, M., Tiotto, E., Amaral, J.N., Martorell, X.：改进 PGAS 环境中的通信：UPC 中的静态与动态合并。在：第 27 届国际 ACM 超级计算会议（ICS'13）论文集。第 129–138 页。计算机协会（2013）。 https://doi.org/10.1145/2464996.2465006, https://doi.org/10.1145/2464996.2465006 2. Alvanos, M., Tiotto, E., Amaral, J.N., Farreras, M., Martorell, X.：利用共享数据局部化降低统一并行 C 程序中检查器-执行的开销。《并行计算》**54**，2–14（2016）。https://doi.org/https://doi.org/10 002 3 1016 2016 03 . / j.parco. . . . Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.：NAS 并行基准测试 2.0。技术报告，NASA 艾姆斯研究中心技术报告 NAS-95-020（1995）4. Beamer, S., Asanovi´c, K., Patterson, D.：GAP 基准测试套件（2015）。https://doi.org/10 03619 5 48550 1508 . /ARXIV. . . Bianchini, M., Gori, M., Scarselli, F.：深入了解网页排名。ACM 互联网技术汇刊**5**(1)，92–128（2005）6. Chamberlain, B.L., Callahan, D., Zima, H.P.：并行可编程性与 Chapel 语言。《国际高性能计算应用杂志》**21**(3)，291–312（2007）7. Das, R., Uysal, M., Saltz, J., Hwang, Y.S.：分布式内存架构上不规则科学计算的通信优化。《并行与分布式计算杂志》**22**(3)，462–478（1994）8. Davis, T.A., Hu, Y.：佛罗里达大学稀疏矩阵集。《ACM 数学软件汇刊》**38**(1)，1:1–1:25（2011年12月）。https://doi.org/10 2049663 9 1145 2049662 . / . . Ding, C., Kennedy, K.：通过运行时的数据与计算重组提升动态应用程序的缓存性能。在：ACM SIGPLAN 1999年编程语言设计与实现会议论文集。第 229 301670 10 241 1999 10 1145 301618 – 页。ACM（）。https://doi.org/ . / . . Dongarra, J., Heroux, M.A., Luszczek, P.：高性能共轭梯度基准测试：一种用于高性能计算系统排名的新指标。《国际高性能计算应用杂志》**30**(1)，3–10（2016）11. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.：UPC：分布式共享内存编程，Wiley 并行与分布式计算丛书，第 40 卷。John Wiley & Sons （ 2005 ） 12. Kayraklioglu, E., Ronaghan, E., Ferguson, M.P., Chamberlain, B.L.：Chapel 编译器中的基于局部性的优化。载于：Li, X., Chandrasekaran, S.（编），《并行计算语言与编译器》。第 3–17 页。Springer（2021）13. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.：并行图处理中的挑战。《并行处理快报》**17**(01)，5–20（2007）14. Nieplocha, J., Harrison, R.J., Littlefield, R.J.：全局数组：面向高性能计算机的非一致性内存访问编程模型。《超级计算杂志》**10**(2)，169–189（1996）15. Rauchwerger, L., Padua, D.：LRPD 测试：对循环进行推测性运行时并行化，并采用私有化和归约并行化。《IEEE 并行与分布式系统汇刊》**10**(2)，160–180（1999）。https://doi.org/10.1109/71.752782

16. Saltz, J.H., Mirchandaney, R., Crowley, K.：循环的运行时并行化与调度。《IEEE计算机汇刊》**40**(5)，603–612（1991）17. Strout, M.M., Carter, L., Ferrante, J.：运行时数据与迭代重排序的编译时组合。在：ACM SIGPLAN 2003年编程语言设计与实现会议论文集。第 91–102 页。ACM（2003）18. Su, J., Yelick, K.：高级语言对不规则计算的自动支持。在：IEEE国际并行与分布式处理研讨会。第 2 卷，第 53b–53b 页。IEEE（2005）19. Webber, J.：Neo4j 编程入门。在：第三届年度系统、编程与应用：软件服务于人类会议论文集。第 217 2384777 20 218 2012 10 1145 2384716 – 页。ACM（）。https://doi.org/ . / . . Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.：新兴多核平台上稀疏矩阵-向量乘法的优化。在：SC'07：2007年ACM/IEEE超级计算会议论文集。第 1–12 页。IEEE（2007）