arXiv:2305.03317v1 [cs.DC] 5 May 2023

StarPlat：用于图分析的多用途领域特定语言

NIBEDITA BEHERA，ASHWINA KUMAR，EBENEZER RAJADURAI T，SAI NITISH，
RAJESH PAN-DIAN M，以及RUPESH NASRE，印度马德拉斯印度理工学院。

图用于对多种现实世界现象建模。随着非结构化和半结构化数据不断增长，图算法的并行化已不可避免。然而，由于计算、内存访问和通信本身具有不规则性，图算法传统上难以并行化。为应对这一挑战，人们提出了多种库、框架和领域特定语言（DSL），以减轻用户的并行编程负担；这些用户往往是领域专家。然而，现有的图算法建模框架通常只面向单一架构。本文介绍一种名为 StarPlat 的图领域专用语言，程序员可以用高级形式描述图算法，而系统则能根据同一算法规范为三种不同后端生成代码。具体而言，该领域特定语言编译器可为多核系统生成 OpenMP 代码，为分布式系统生成 MPI 代码，并为众核 GPU 生成 CUDA 代码。由于这三者属于完全不同的并行编程范式，将它们统一到同一种语言之下颇具挑战。本文分享了我们在语言设计方面的经验。编译器的核心是一个中间表示，它为高级程序提供统一表示，并以此为基础分别生成各个后端的代码。我们通过描述四种图算法展示 StarPlat 的表达能力：介数中心性计算、PageRank 计算、单源最短路径和三角形计数。借助一组包含十个大型图的数据集，我们将生成的代码与手工编写的库代码进行性能比较，以展示该方法的有效性。我们发现，在许多情况下，生成的代码能够与基于库的代码相媲美。更重要的是，我们证明了可以根据同一图算法规范，为不同目标架构生成高效代码。

CCS 概念：• **计算方法** → **大规模并行算法**；共享内存算法；**并行编程语言**；• **软件及其工程** → **并行编程语言**。

附加关键词和短语：图算法、领域特定语言、OpenMP、MPI、CUDA

1 引言

如今，图数据结构已成为许多现实应用中用于建模数据关系的组成部分。非结构化和半结构化数据的巨大增长，使得这些图的规模扩大到拥有数十亿条边。因此，要扩展到如此庞大的图规模，并行图分析解决方案已不可或缺。过去二十年见证了硬件在并行处理方面的显著推进，这也推动了软件支持方面的重大发展，无论是以库的形式还是以编程语言的形式。这些硬件和软件架构主要适用于规则代码，因为其中的数据访问、控制流和通信模式都可以静态确定。例如，如今编译器可以自动对规则矩阵计算进行分块，以改善缓存性能或将通信降至最小。

遗憾的是，现有的广泛使用的编译器在图算法方面表现不佳。这是由于稀疏图处理固有的 不规则性 所致：其访问模式取决于输入（而编译时无法获取输入）。另一方面，我们的社区已证明，在多个真实世界的图上，图算法具有足够的并行性，能够让核心保持繁忙 [18]。遗憾的是，手动利用这种并行性

作者地址：Nibedita Behera ,cs20s023@cse.iitm.ac.in;AshwinaKumar,cs20d016@cse.iitm.ac.in; Ebenezer Rajadurai T, ebenezerrajadurai5@gmail.com;Sai Nitish,bsainitishkumar@gmail.com;RajeshPandianM,mrprajesh@cse.iitm.ac.in;RupeshNasre, rupesh@cse.iitm.ac.in, 马德拉斯印度理工学院, 印度。

© 2023

在各种硬件上实现这一点颇具挑战性。程序员需要精通应用领域，才能利用算法属性；需要精通高性能计算（HPC），才能对计算进行建模以适配目标硬件；还需要精通计算系统，才能在给定基础设施上优化整个应用及其支持软件。即使企业和机构能够找到并聘请这样的专家，该专家也未必适合另一个应用领域。一个令人遗憾的现实是，我们有不精通高性能计算或系统的领域专家，也有可能对底层应用领域了解不足的高性能计算专家。

一种可行的提高生产效率的替代方案是使用图处理库或领域特定语言，让领域专家通过应用程序接口或高级构造表达算法，再由库或领域特定语言编译器负责为目标硬件生成高性能代码。如今已有多种此类框架[14, 22, 32, 33]。其中许多框架会部分或完全隐藏并行性的复杂细节，提供调度策略的助记符，并执行程序分析以识别数据竞争，从而生成同步代码。与手工编写显式并行代码相比，通常所需编写的代码量会显著减少。

现有框架（库或图领域特定语言）的一项局限是，它们针对特定的硬件架构。例如，Ligra [29] 是专为共享内存系统设计的图处理框架。Greenmarl [14] 主要面向多核设备，并通过 Pregel API 支持分布式实现。Gunrock [32] 是用于 GPU 图处理的 CUDA 库。GraphIt [33] 是一种面向共享内存和众核系统的领域特定语言。很少有框架能面向多个后端（例如 Kokkos [30]）。为获得最佳性能，并适应当今使用的各类高性能计算硬件，库往往会将自身限制在某种特定目标上。多核并行允许数据由所有正在运行的线程共享，而集群级并行则要求显式通信；此外，为了获得最佳性能，众核并行还需要分层计算和 SIMD 执行。领域专家虽然了解这些基本差异，但他们应当能够针对某种架构进行指定，而不是编写这些内容，以实现所需的性能。因此，理想情况下，领域特定语言应在设计中涵盖不同后端的特性。这样既有利于简化并提高代码生成效率，也可避免修补原本为其他并行架构设计的语言。

在这项工作中，我们提出了一种名为 StarPlat 的图领域专用语言，用户可以使用其中高级的图专用构造来提供图问题的算法规范，并根据同一算法规范为多个后端生成代码（目前包括多核、分布式和众核）。这些构造经过精心设计，旨在将并行化相关的具体实现从用户面前抽象出来，同时确保生成高性能的图代码。StarPlat 编译器会将这些构造中包含的高级信息转化为针对不同架构的具体实现。编译器将领域专用语言代码转换为 C/C++。在多核环境中，它使用 OpenMP 并行性；在分布式环境中使用 MPI；在众核架构中则使用 CUDA。在同一领域特定语言编译器中涵盖这三种截然不同的后端颇具挑战，我们在本文中重点介绍了这些挑战。我们坦承，StarPlat 借鉴了现有库和领域特定语言所提供的见解，从这些框架中吸收了一些构造（同时也提供了我们自己的构造），并生成了性能可与这些框架相媲美的代码。具体而言，我们通过讨论四种图算法展示了 StarPlat 的多功能性：介数中心性 (BC)、PageRank (PR)、单源最短路径 (SSSP) 和三角形计数 (TC)。

本工作的技术贡献如下：

- StarPlat[1] 是一种用于图分析的领域特定语言，允许用户编写静态图处理的高级算法规范，既能体现并行性意图，又与目标架构解耦。

[1] https://github.com/nibeditabh/StarPlat

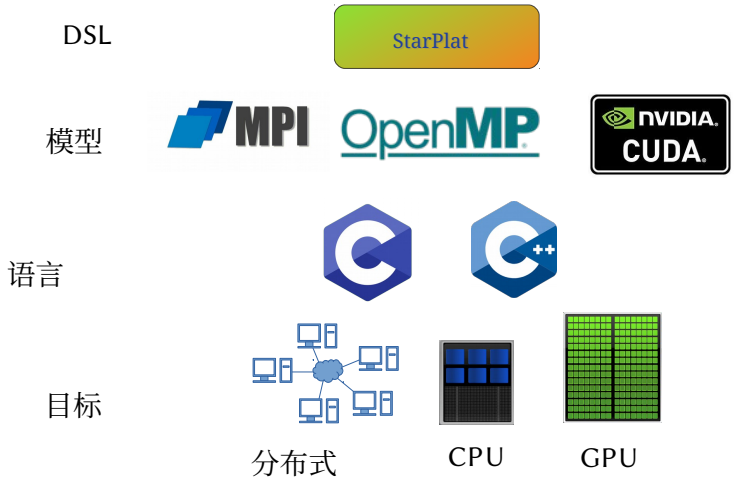

图 1. StarPlat 编译器概览

- 一种在各后端之间通用的中间表示，能够捕获与构造相关的所有必要信息，以便将其转换到目标架构。

- 一种代码生成方案，用于将中间表示转换为适用于多核（使用 OpenMP）、分布式（使用 OpenMPI）和众核（使用 CUDA）后端的高效代码。
- 对生成的介数中心性、PR、单源最短路径和三角形计数代码在多种流行图和不同目标硬件上的性能进行分析，并展示其相较于手工构造的框架所具备的竞争力。

本文其余部分安排如下：第 2 节介绍语言规范和中间表示。第 3 节描述各后端将领域专用语言代码转换时采用的代码生成方案。第 4 节概述 StarPlat 为实现高效代码生成而采用的后端特定优化。第 5节讨论各后端生成代码的实验评估。第6节讨论图分析领域的相关工作。我们总结经验，并在第 7节作出结论。

2 STARPLAT语言与前端

StarPlat 的高层设计理念是尽可能减轻用户编写并行化构造的负担，同时通过提供聚合相关的构造和提示，实现与手工调优代码相媲美的性能。在少数情况下，如果抽象与性能之间必须取舍，我们会有意识地优先考虑抽象。这是因为该语言主要面向领域专家（而非高性能计算专家）。

从一开始，这门语言的设计目标就是抽象掉硬件。这颇具挑战性，因为不同后端之间差异很大。但我们发现，算法层面存在一些共性，并通过特定构造将其编码，从而将其转换为相应的后端代码。例如，在 OpenMP 中需要基于锁的同步时，在 MPI 中也需要通信，而在 CUDA 中则需要无锁同步。

StarPlat 提供了与静态图算法相关的各种抽象和数据类型，例如 Graph、节点、边、propNode（用于表示节点属性）等。程序员采用过程式风格编写程序，并使用 forall 等聚合迭代构造以及其他固有的并行操作，提示可进行并行化的机会。编译器决定是否利用这种并行化（例如嵌套 forall）。许多算法可以看作

反复执行直至收敛的迭代过程。收敛准则取决于具体应用。此类迭代过程通过 afixedPoint 构造来描述。StarPlat 采用“开箱即用”的理念，并为各种数据类型提供了多种实用函数。在内部，每个函数均针对目标硬件实现。

2.1 编译器概述

与通用编译器一样，StarPlat 的编译器分为前端和后端，如图 2 所示。前端负责确保 StarPlat 实现的语法和语义一致性。前端会根据提供的输入代码构建抽象语法树（AST）。所有后端共用同一棵 AST，并在编译器解析阶段为每个构造填充元数据。随后，系统会根据用户希望生成的目标代码，将其交给相应的代码生成器。

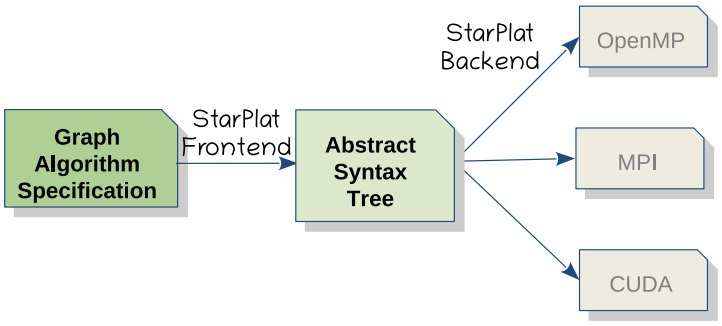

图 2. StarPlat 编译器的处理流程

2.2 语言概述示例：单源最短路径

单源最短路径计算（SSSP）旨在求出从指定源节点到图中其他每个节点的最短路径。并行单源最短路径计算可以建模为一个迭代过程，如图 3 所示。这是 Bellman-Ford 算法的一种变体。

1 函数 computeSSSP (图 g , 节点 s r c) { 2 节点属性 < 整数 > 距 离 ; 3 节点属性 <布尔 >修
改
e d
nbr
nbr

```
13 }    }    }    }
```

图3. StarPlat 中的单源最短路径计算

函数 compute_SSSP 接受两个参数：一个图 g 和源顶点 src。语句 2 声明了一个名为 dist、类型为 propNode 的变量。propNode<整数>表示 dist 是与每个顶点关联的整数类型属性。语句 3 创建了一个名为 modified 的布尔类型顶点属性。在语句 4 中，函数 attachNodeProperty 将这两个属性关联到 g 的每个顶点，并通过聚合操作将 dist 的属性值初始化为无穷大（INT_MAX），将 modified 初始化为 false。在语句 5–6 中，为顶点 src 的 dist 和 modified 属性赋予特定值。具体来说，src 的距离设为零，同时将该顶点标记为已修改。语句 8 中的不动点构造确保迭代过程持续执行，直到变量 finished 变为 true。变量 finished 会在每次迭代中更新，并决定是否需要进行下一次迭代。当没有顶点被修改时，即达到不动点。为此，对每个节点的 modified 属性执行或运算，并将结果赋给 finished。语句 9 中的迭代构造 forall 指定了对图中节点进行并行迭代。此外，可选的过滤子句允许处理满足特定条件的顶点子集。语句 10 中的 forall 构造指定并行迭代顶点 v 的所有邻居，并对每个邻居 nbr 松弛相应的边（语句 11–12）。语句 12 中的多重赋值构造执行多项任务，是 DSL 提供的抽象的一个恰当示例（借鉴自 Green-Marl [14]）。右值 Min(nbr.dist, v.dist + e.weight) 会被赋给 nbr 的 dist 属性；如果经过 v 的备选距离小于 nbr 当前的距离，则该值为 v.dist 与 e.weight 之和。如果更新了 nbr.dist，就会设置 nbr 的 modified 属性。这种构造允许将相关语句作为一个整体处理（准确地说，作为临界区），并将其转换为受某种同步机制保护的更新逻辑，以防止数据竞争。

我们强调，单源最短路径代码在结构上与教科书中描述的算法相似，只做了少许改动。因此，StarPlat 简化了并行化背景下图算法的实现、分析和修改。该代码的多核转换将采用与 StarPlat 代码类似的结构，并插入 OpenMP 编译指令；分布式版本的差异在于跨设备分散和汇集数据所涉及的复杂性；而众核版本则需要预先将 forall 内使用的数据从 CPU 传输到 GPU。

现在，我们将深入探讨 StarPlat 的各种构造。

2.3 语言构造

我们首先讨论各种数据类型，然后介绍迭代方案和归约。

2.3.1 数据类型。 StarPlat 支持将图论概念作为一等类型，例如图、节点、边、节点属性、边属性等。它还支持基本数据类型：整数、bool、long、float 和 double。

Graph 数据类型封装了独立图的操作和属性。这些属性包括其节点、边、节点数、边数等。StarPlat 以压缩稀疏行（CSR）格式存储图，该格式既具备邻接列表的存储优势，也由于采用偏移量而支持在设备间无缝传输。该数据类型还便于在节点和边级别收集和操作信息。由于节点和边与图紧密关联，Graph 可以通过各种库函数方便地支持这些操作。例如，根据 neighbors(u) 的语义，对于有向图，它返回所有出边邻居；对于无向图，它返回所有邻居。对于有向图，图类型还提供一个函数 nodesTo()，用于返回指向某节点的入边邻居。这要求 Graph 还维护一份 CSR 表示，用于存储

其转置，这对于需要在输入图的转置版本上进行计算的算法（例如介数中心性）很有用。

```
1 函数 foo(图 g, 节点属性 <整 数 >modified, 节点属性 <整 数 >data){2 节点集合 <
g>
邻
```

图 4. 展示 StarPlat 中各种数据类型的示例程序

节点和边本身都可以关联属性。在单源最短路径问题中，顶点的距离可视为节点属性，由相应的算法计算得出。同样，在介数中心性计算中，每个节点的介数中心性值也可视为一种属性。 节点属性 数据类型用于声明图中节点的属性，并可指定其类型。图类型提供的 attachNodeProperty函数会将此属性绑定到图的节点上，并在提供初始值时初始化属性值。我们图 3 中的单源最短路径代码第 2 行3 声明了一个节点属性 距离 ，其类型为 整数。 attachNodeProperty 将距离绑定到图，并可选择为每个顶点初始化距离属性（例如，在图 3 中初始化为无穷大）。同样，propEdge 数据类型与边关联，其语义与 节点属性 相同。 attachEdgeProperty 函数会将该属性绑定到图的边上。

StarPlat 还提供了集合类型，例如列表、节点集合和边集合。列表允许存在重复元素，而节点集合和边集合分别存储唯一的节点和边。图4中的第2行展示了一个使用示例。在节点集合与边集合之间进行区分，可以在基于顶点的代码和基于边的代码中选择相关实现。

*2.3.2 并行化与迭代方案。*forall 是 StarPlat 中的一种聚合结构，可以并行处理一组元素。它的顺序对应形式是一个简单的for语句。目前，StarPlat 支持基于顶点的处理[2]。并行forall 支持对各种范围进行迭代（例如，整个图中的节点或某个节点的邻居，如图3第9 和10行所示）。

在图g.nodes() 上调用函数g 会返回一个可进行迭代的节点序列。要迭代某个节点u的邻居，函数 g.neighbors(u)、g.nodesTo(u) 和g.nodesFrom(u) 会返回类似的序列。forall 的主体可以通过包含一个筛选结构，根据节点标签或节点属性上的某个布尔表达式，有选择地对满足条件的节点执行。图4中的第3行展示了使用节点属性modified的用法。

考虑到若干图算法可以使用单个外层并行循环进行良好表示，并且嵌套并行循环的分析较为复杂，目前 StarPlat 仅支持外层并行。因此，领域特定语言中的嵌套 forall 会在目标代码中产生一个并行的外层循环和一个串行的内层循环。有人可能会认为，对顶点执行外层循环、对邻居执行内层循环可以从嵌套并行中受益，我们对此表示赞同。然而，（i）基于边的并行（将予以支持）可以很好地处理此类计算；（ii）由于我们的目标是大型图，即使是基于顶点的处理也具有足够的并行性，可以让资源保持繁忙。

[2]我们正在添加对基于边的处理的支持，这需要更改底层数据表示。压缩稀疏行（CSR）存储格式适用于基于顶点的处理。

对于图处理领域特定语言，遍历成为基本构建块。StarPlat 提供了广度优先遍历这一构造，它借鉴自 GreenMarl [14]。

```
iterateInBFS(v 属于 g.nodes()从 root) {...}
```

`iterateInBFS` 从给定的根节点开始对图执行 BFS 遍历。底层处理按层进行，并并行遍历特定层中已访问的节点。访问某一层中的节点时，会执行主体语句并生成下一组待访问节点。可以使用 `filter()` 构造有选择地探索已访问节点的邻域。同样，`iterateInReverse` 以层同步的方式执行反向 BFS 遍历，并在主体语句的计算过程中提取每一层的并行性。请注意，`iterateInBFS` 是使用 `iterateInReverse`的前提，因为前者会构建后者要遍历的 BFS DAG。函数 `neighbors`、`nodesTo` 和 `nodesFrom` 在 `iterateIn...`构造中使用时，其含义会略有变化：它们对应的是 BFS DAG 中的邻居，而不是原始图 `G`。我们发现，这种语义变化符合我们编写代码时的自然直觉。我们在介数中心性计算中对此进行了说明（附录 A）。

2.3.3 归约。 归约是常见的并行编程原语之一，有助于实现高效计算。在领域特定语言中指定归约还有助于传达同步的必要性。然而，直接将归约作为一种语言构造来支持，并不完全符合 StarPlat 的设计理念。因此，作为一种折中方案，StarPlat 允许使用某些相关的 C 运算符（例如 `+=`）来表达归约。这一“技巧”使我们能够保留抽象，同时仍然实现生成代码的高效性。StarPlat 支持的归约运算符列于表 1中。

运算符	归约类型
+=	Sum
*=	乘积
++	计数
&&=	All
||=	Any

表 1. StarPlat 中的归约运算符

我们在图 5 中展示了归约的用法。在代码中引入归约（第 7 行）可确保`accum` 变量在并行区域结束时得到确定的结果。请注意，第 5 行涉及线程局部变量 `count`，因此不需要归约。另一方面，如果支持嵌套并行，`count` 也需要进行归约。StarPlat 中的归约运算符会转换为基于库的实现，以便在目标后端执行归约。[3]

```
<escape>1 int 累加器 = 0;2 对所有 (v 在 g . 节点 ( ) ) {3 int 计数 = 0;4 对所有
(nl
</e
```

[3]目前，编写正确运算符的责任在于领域特定语言的用户。我们正在为 StarPlat 添加程序分析功能，以免用户再为此操心。

```
7累加  += 计数 ; / / 赋值形式的求和归约8  }
```

图 5. 归约示例

2.3.4 不动点和最小值/最大值构造。 图算法的多种解法都是迭代式的，并根据节点属性的条件收敛。StarPlat 提供了不动点构造，可简洁地指定这一过程。其语法包含一个布尔变量和一个基于节点属性的布尔表达式，用以构成收敛条件，如下所示。

不动点 直到(变量：收敛表达式) {...}图 8中的第 3 行在单源最短路径的规格说明中使用 不动点 构造来定义收敛条件。循环会持续迭代，直到至少一个节点的 已修改属性被设为 true。

StarPlat 提供了最小值和最大值构造，可根据比较准则执行多重赋值。这对于单源最短路径等基于更新的算法很有用：在满足所需条件时更新节点属性，同时处理潜在的数据竞争。

在单源最短路径计算中，最小值 构造用于在第 12 行编码松弛准则。当经过顶点 v 的替代距离小于 nbr 当前距离时，便会更新邻居节点的距离。使用 距离 属性并根据此比较指定的更新操作，通过 最小值 构造会将 已修改 属性更新为 True。

StarPlat 还提供 minWt 和 maxWt 聚合函数，用于查找最小和最大的边权。 .

2.4 抽象语法树（AST）

构成该语言的每个有意义的非终结符都在最高层表示为 ASTNode。ASTNode 是抽象语法树中所有节点的父类。ASTNode 派生出子类节点，用于表示具体类型，例如语句节点和表达式节点。语句节点是所有语句类型的父类，例如赋值、声明、控制流、过程调用等。由于解析器中构造的定义方式，这些节点有时可以由其他节点组成。例如，forallStmt 的构造由一个主体、一个迭代空间（由迭代器和范围函数定义），以及一个可选的、作用于迭代项的过滤表达式组成。在 AST 中，forallStmt 的主体表示为语句节点，迭代器表示为标识符节点，范围表示为 proc_callExpr，过滤表达式表示为表达式 节点。其他节点类型包括 Identifier、反向广度优先迭代、类型、归约调用、fixedPoint 语句、形式参数等。

大多数节点的数据在解析阶段就已填充。有关符号类型的数据，则是在已构建的抽象语法树上再进行一次遍历时添加的。给定目标后端后，会向构造的抽象语法树填充与该特定后端代码生成相关的其他信息。代码生成前的分析阶段会执行分析，以实现优化且高效的代码生成。其中一些优化是目标后端特有的。将节点划分为不同的子类，便于在代码生成器中为不同类型的节点维护相应的处理程序。代码生成器构成编译器的后端部分。与主要独立于目标平台的前端不同，后端与目标并行化平台紧密耦合。

3 STARPLATCODE 代码生成器

代码生成阶段主要支持为三种目标后端生成高效代码：共享内存、分布式和众核平台。每种后端生成的代码还会使用特定的后端库

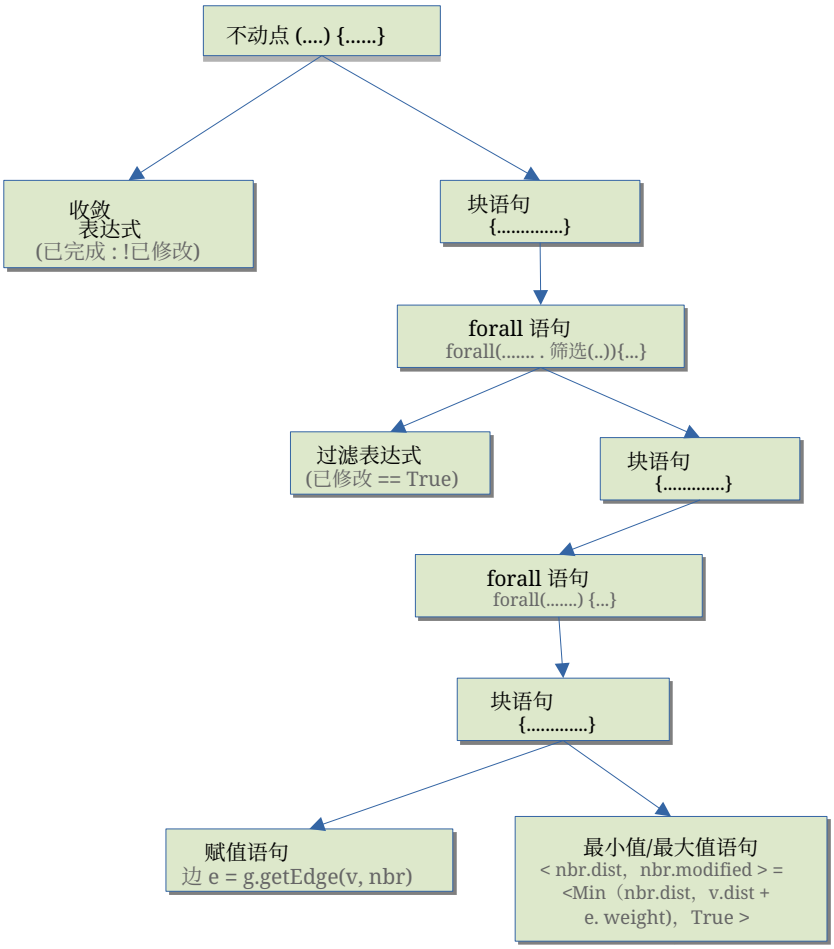

图 6. SSSP 的抽象语法树不动点构造（见图3

以处理 StarPlat 代码中直观指定的并行处理。共享内存实现使用 OpenMP 作为线程库，分布式内存实现使用 MPI，众核实现使用 CUDA。

3.1 图表示与存储

尽管图可以采用多种格式存储，我们更倾向于使用满足以下条件的格式：

- 最好能够在所有后端上运行
- 适用于图处理中常见的以顶点为中心的算法
- 可以轻松拆分

最后一项偏好不仅是出于 MPI 后端的考虑，也源于我们未来计划扩展 StarPlat，以支持多 GPU 和异构后端。由于基于偏移量的格式符合我们的需求，我们可以选择压缩稀疏行（CSR）或坐标列表（COO）格式，而 CSR 符合第二项偏好。因此，我们选择了 CSR。CSR 也广泛应用于稀疏图处理框架中。StarPlat 拥有一个后端图处理库，负责从输入文件加载图，并以 CSR 格式存储图。

OpenMP 后端在图的存储方面不存在任何问题。对于 MPI 后端，图由秩 0 进程加载，并采用简单的分块方式分发给各 MPI 进程，因此每个进程都包含本地和远程顶点。只有本地顶点的信息可以直接访问，而远程顶点的信息则需要通过显式的发送与接收进行通信。MPI 后端维护本地顶点 ID 与其对应的全局顶点 ID 之间的映射，以确保正确处理以及数据的发送和接收。例如，一个较大的全局顶点 ID 在某个 MPI 进程中仍可能映射到本地 ID 0。CUDA 后端需要将图及其相关属性完整地从主机传输到 GPU（使用多次 cuda Memcpy 调用）。由于这些算法处理的是静态图，因此在处理结束时，只需将修改后的属性信息从 GPU 传回主机。

在各种后端之间无缝管理图表示和属性存储，可使领域用户免于处理复杂且容易出错的图。

3.2 总体流程

在 OpenMP 后端中，最外层的 forall 会被转换为以 #pragma omp parallel for 为头部。输入图会被解析，并在生成的代码中存储为 C++ 类；节点和边则表示为整数（作为其 ID）。节点属性和边属性会被转换为属性声明中指定类型的数组。我们发现，与另外两个后端相比，生成 OpenMP 代码相对容易，这是因为：(i) 只有一个设备；(ii) 采用共享内存范式，所有线程的变量都位于同一内存地址空间中；(iii) 采用基于 pragma 的处理，无需修改顺序代码。

MPI 后端生成遵循批量同步处理 (BSP) 风格的代码，其中每次迭代都包含一个计算步骤，随后是一个通信步骤。在计算阶段，各个 MPI 进程执行主循环代码，更新其本地顶点的属性，并将这些属性标记为待发送到相应全局顶点，存入发送缓冲区。在通信阶段，各进程接收来自其他进程的这些更新，将其转换为本地顶点，并更新属性。与 OpenMP 类似，节点和边的属性/属性值会被转换为数组。

与 OpenMP 和 MPI 不同，CUDA 后端需要将 `forall` 循环转换为 GPU 内核，而其作用域有所不同。这会导致某些变量无法使用，因此需要显式分配（`cudaMalloc`）并传输（`cudaMemcpy`）。此外，`forall` 循环体可能包含局部变量，这些变量在 CUDA 中会转换为线程局部变量。为区分这两类变量，CUDA 后端会对抽象语法树进行基本分析。[4] 除每个最外层的 `forall` 外，属性值的初始化也需要转换为单独的内核（而在 OpenMP 和 MPI 中，这是一段循环）。内核启动配置设为固定的每个线程块的线程数（生成的代码中为 1024，这是我们实验中平均表现最佳的设置），线程块数量则与顶点数量成正比。

[4]全面的程序分析正在进行中，旨在从抽象语法树层面对算法进行分析，其中包括识别数据竞争，并通过适当的同步 o 或通信来解决这些问题。 r

除此之外，样板代码由计时函数调用组成，用于计量初始化、不动点处理以及适用情况下的数据传输时间。

3.3 邻域迭代

遍历邻域通常嵌套在遍历图顶点的循环中（可能带有筛选条件，如图 3 所示）。对于 OpenMP 后端，生成的代码片段如下所示。

```
1 #pragma omp p a r a l l e l for 2 for ( int v = 0 ; v < g . num_节点 ( ) ; v ++) { 3 i f ( ! m o d
i f
N (
```

图7. 为邻域迭代生成的 OpenMP 代码

MPI 代码非常相似，但会根据进程秩遍历预定义的范围。

```
1 mpi : : communicator world ; 2 myrank = world . rank ( ) ;
3 n
/ ( f
s t
<
r t
11
```

图8. 为邻域迭代生成的 MPI 代码

CUDA 代码需要将内核调用与内核处理分开。内核调用会设置启动配置（线程块数量以及每个线程块的大小），并传递适当的参数（图、属性、算法特定变量，例如单源最短路径中的源节点，以及内部使用的其他杂项变量）。内核使用一维线程标识符，并利用复制到 GPU 的 CSR 遍历邻居。请注意，与 OpenMP 和 MPI 不同，这里没有遍历顶点的循环；该操作由启动内核时使用的并行线程处理。

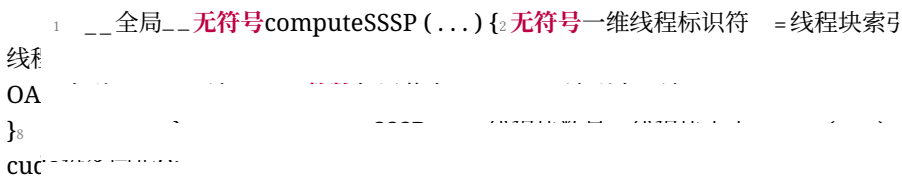

图 9. 邻域迭代生成的 CUDA 代码

对于有向图和无向图，以及入邻居（g.nodesTo(v)）的情况（后者要求使用反向邻接关系（g.revIndexofNodes[v]）），循环边界函数会相应地进行调整。

3.4 归约

回顾 StarPlat 支持的归约操作（例如第 7 行）。标量上的归约会转换为 OpenMP 的`reduction`子句，如下所示。OpenMP 归约实现会为每个线程创建并更新归约变量的私有副本。在并行区域结束时，各线程的本地更改会累积到全局变量中。这显著降低了为确保共享变量操作正确性而需要处理数据竞争的成本。

```
#pragma omp parallel for reduction(+:accum)
```

CUDA 中的情况会因驻留线程块数量的限制而变得复杂，这可能导致内核阻塞。解决这一问题的一种方法是从主机端调用归约操作，并将其作为独立内核执行（例如使用`thrust::reduce`）。但这需要先终止内核，返回主机端并调用归约操作，再返回主机端，然后调用另一个内核来完成`forall`循环中的其余处理。这不仅增加了代码生成的复杂性，还会降低整体处理效率。因此，我们依靠原子操作来生成在功能上等效的代码（例如，在图 5 的示例中使用 atomicAdd ）。

```
atomicAdd(&accum, prop[nbr]);
```

3.5 BFS 遍历

iterateInBFS 构造会展开为并行广度优先搜索。这个单一构造会展开为约 40 行 OpenMP 代码、60 行 MPI 代码和 40 行 CUDA 代码。因此，我们仅展示并讨论下面代码片段中的关键部分。

如图中的代码片段所示，OpenMP 后端维护一个向量的向量，用于按层跟踪顶点（图 10）。广度优先搜索有一个外层`while`循环和两个内层循环：一个遍历某一层的顶点（在第 6 行并行化），另一个将某一层中已探索的顶点添加到相应的层向量中（第 18 行），该层向量充当前沿。并行循环会检查一个顶点的邻居，并判断它们当前的距离是否尚未设置（值为 -1），然后

将其设置为 1 + 当前顶点的距离。为避免数据竞争，此更新需要通过 CAS（比较并交换）以原子方式执行（第 9 行）。

```
1 层级节点 [阶段 ].push_back(根节点); 2 ... 3 当(bfs计数 > 0){ 4 ...
5 #
点
bfs
编
主
();
```

图 10. 为 iterateInBFS 构造生成的 OpenMP 代码

MPI 后端运行着一个类似的外层`while`循环，并逐层遍历顶点，但仅针对本地顶点集合。一旦某个顶点的邻居节点得到更新，该顶点就会被推送到发送缓冲区（第14行）。在每一层结束时，通过一次全对全调用交换本地顶点和远程顶点的层级值（第17行）。不同层之间采用 BSP 风格的 MPI 屏障进行分隔（第23行）。

```
1 当(is_finished(startv,endv,active)){ 2
...
ba
```

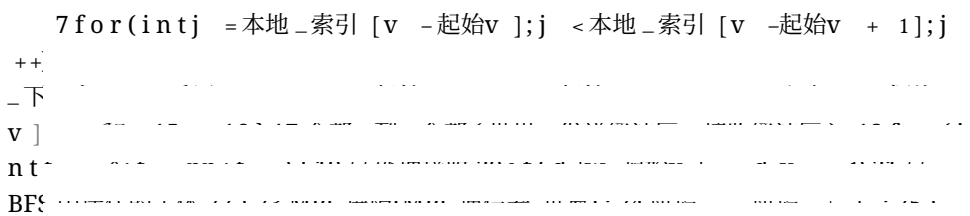

图 11. 为 iterateInBFS 构造生成的 MPI 代码

CUDA 后端对`iterateInBFS` 构造提出了挑战，因为与其他两个后端不同，生成的代码需要同时在主机和设备上运行。外层`do-while` 循环在主机上运行（第1行），其中会调用 GPU 上逐层执行的广度优先搜索内核（第13行）。由于循环在主机上运行，为确保循环条件能够推动循环继续执行，标志（`finished`）需要在设备之间反复复制（第2 行和8行），作为参数传递给内核，并在顶点层级发生变化时于内核中更新。



```
12 13 g l o b a l v o i d 广度优先搜索 ( . . . ) { 14 . . . 15 i f ( d l e v e l
[ u
d i
d _
o u
22 /
```

图 12. iterateBFS 构造的 CUDA 等效代码

3.6 最小值/最大值构造

回顾一下最小值构造在单源最短路径（第12行）中的用法。在 OpenMP 中，通过基于原子CAS（比较并交换）的这些操作的原子实现，可以实现对节点属性的条件更新；这些实现以 StarPlat 库的形式提供。单源最短路径中最小值构造生成的 OpenMP 代码如下。

```
1 整数d i s t _n e w =d i s t [v ] +权重 [e ];2 布尔
值n
n e
值
a l
mo
```

图 13. 为最小值构造生成的 OpenMP 代码

在 MPI 后端中，邻居节点可能是本地节点，也可能是远程节点。如果是本地节点，每个进程都可以更新其属性，而无需同步。如果是远程节点，要更新的属性值会存储在发送缓冲区中，最终在通信阶段发送给该节点所属的进程。

```
1 i f ( nbr >= s t a r t v && nbr <= endv ) { // l o c a l 2 i n t e =边
0
ht
```

```
6if(dist[nbr-startv]>dist_new){7dist[nbr-startv]=dist_new;
8m
owr
to
```

CUDA 后端处理最小值/最大值构造的方式与 OpenMP 后端完全相似，区别在于：(i) 使用的变量是 GPU 副本；(ii) CUDA 已原生支持原子指令（原子最小值操作和原子最大值操作）。

```
1intnbr=gpu_edgeList[edge];2inte=edge;3intdi
st
st
st
in
```

3.7 不动点构造

不动点构造会转换为一个以构造中提供的不动点变量为条件的 while 循环。通常，收敛依据的是节点属性，但也可以是任意计算。此代码在主机上生成，因此三个后端的模板都相似。不过，与之前一样，MPI 代码需要根据各进程所处理的顶点 ID，收集相关信息，以设置完成标志。类似地，CUDA 代码会在 GPU 上更新一份完成标志的副本，然后通过 cudaMemcpy 将其复制到主机。

```
1当(!finished){2finished=为真;34如果(...){//最小值-构造扩展5
...
的
```

4 优化

有些优化可以嵌入手工编写的代码中，而 StarPlat 的代码生成器无法做到这一点。例如，如果某个算法的不同变体在不同后端上的表现更好，我们就无法根据同一算法规范自动生成这些变体（例如，基于推送与拉取的处理方式 [2]），尽管该领域特定语言能够分别指定这些变体。在实现层面，CUDA 中的持久化内核 [11] 可以缩短算法在大直径图上的执行时间。然而，StarPlat 会在主机上为各个后端生成 不动点 构造的代码。尽管如此，我们观察到，许多优化都可以轻松地嵌入 StarPlat 生成的代码中：

- 算法优化可以直接嵌入领域特定语言规范中，该规范是所有后端的通用输入。例如，附录中的图21 展示了一种基于拉取的单源最短路径算法。
- 后端特定优化可以嵌入相应的后端中，因为这类优化不会在领域特定语言规范中明确提及，而是会被抽象隐藏。

下面将讨论后端特定优化。

4.1 OpenMP

避免 `iterateInBFS` 中的伪共享。 在广度优先搜索抽象中，多个线程在特定层发现的节点会被推入图遍历所需的数据结构。通过为每个线程分配本地更新空间，缓解了此场景下因共享使用同一数组而导致的伪共享。

高效的不动点计算。 不动点构造会针对单个布尔节点属性的特定条件收敛。收敛状态的变化通过一个布尔不动点变量进行跟踪；理想情况下，在分析完所有节点的属性值后更新该变量。对此更新流程进行了优化：更新任意节点的属性值时，同时更新不动点变量。由于硬件会以原子方式处理多个线程对布尔变量的更新，因此不会造成性能损失。

使用内置原子操作。 原子操作已证明开销较低。因此，生成的代码使用 GCC 编译器提供的内置函数实现原子操作，以便线程能够独占访问单条语句，而无需使用锁。

4.2 MPI

通信聚合。 更新远程邻居节点的距离时，MPI 代码会采用通信聚合优化。处理节点不会向远程顶点发送多条消息，而是发送一条包含本地最小值的消息。

使用 Boost。 Boost MPI 库提供的打包/解包机制，在发送和接收向量、映射等 STL 容器以及用户定义的数据类型时，比调用 MPI 库函数具有更好的性能。

快速的基于索引的划分。 StarPlat 目前使用基于索引的划分方式，在多个 MPI 进程间分配图。与使用 Metis 等划分器相比，我们的方法缺点是由于存在多个分区间边，通信量会增加。不过，这种划分方式速度快，也使我们能够根据 MPI 进程的数量任意划分图。目前，该实现假设所有进程拥有相同数量的顶点。因此，我们会为最后一个进程补充临时顶点。[5]

[5]我们当然计划利用 Metis 及其并行变体，以支持 MPI、多 GPU 和异构后端。

4.3 CUDA

优化主机—设备传输。我们对抽象语法树（AST）进行初步程序分析，以识别需要在设备之间传输的变量。例如，由于图是静态的，因此在内核结束时无需将其从 GPU 复制回 CPU。相比之下，经过修改的属性则需要传输回来。同样，完成标志在 CPU 上设置，在 GPU 上有条件地设置，并在不动点处理中再次由 CPU 读取。因此，该变量需要来回传输。forall 局部变量会作为仅设备变量生成。

或归约中的内存优化。我们编写不动点构造的方式是：使用一个经过修改的属性来计算不动点。从高层来看，只要任意顶点的修改标志被设置，就需要进行另一次迭代。这本质上是一种逻辑或运算。StarPlat 利用这一点生成一个单独的标志变量，由多个线程并行设置（对于基本类型，这依赖于硬件提供的原子性）。与在设备间传输修改标志数组相比，管理这个标志在时间和内存方面的成本都更低。

5 实验评估

为了定量评估 StarPlat，我们实现了四种常用算法：介数中心性（BC）、PageRank（PR）、单源最短路径（SSSP）和三角形计数（TC）。我们还使用其他用于比较的框架编写并优化了这些算法。图 3展示了 StarPlat 的 SSSP 代码，另外三种算法的代码则见附录。每段代码约 30 行，领域专家或学生只需付出这些工作量，就能为多核环境、分布式系统和 GPU 生成这四种算法的高效实现。[6]

由于领域专用语言代码相当简短，因此编译是即时完成的。因此，我们重点分析针对各种后端生成的代码的效率。我们将 StarPlat 中特定于后端的代码的性能与现有的最先进的图分析解决方案进行比较（定性比较将在第6节讨论）。StarPlat 的 OpenMP 后端与 Galois [22], Green-Marl [14], 和 Ligra [29]进行比较；其 MPI 后端与 Gluon [8]进行比较；其 CUDA 后端与 GunRock [32] 和 LonestarGPU [5]进行比较。Galois 是一个基于 C++的图分析框架，而 Ligra 基于 C。此外，Galois 还支持图变更，而这并非其他框架关注的重点。与 StarPlat 类似，Green-Marl 是一种为多核后端设计的图领域专用语言。Gluon 是以 C++框架构建的 Galois 分布式版本。LonestarGPU 是手动优化的 CUDA 代码集合，而 Gunrock 是一个基于 CUDA 的库，提供用于图分析的以数据为中心的 API，例如 推进、计算和 筛选。因此，除 Green-Marl 外，我们将 StarPlat生成的代码与手动优化的代码进行比较。

我们的实验使用了十个大型图，涵盖多种不同类型。其中六个是具有小世界特性的社交网络，两个是直径较大且顶点度数较小的道路网络，另外两个是合成图。一个合成图具有均匀随机分布（使用 Green-Marl 的图生成器生成），另一个则具有遵循递归矩阵格式的偏斜度数分布（使用 SNAP 的 RMAT 生成器生成，参数为 a = 0.57, b = 0.19, c = 0.19,d = 0.05）。这些图列于表 2中，并按各类别的边数排序。对于无权图，我们在范围 [1,100] 内均匀随机地为边赋予权重（用于单源最短路径）。

[6]我们认为这非常出色，社区也会对此表示赞赏。

图	缩写	顶点数（百万）	边数（百万）	平均度	最大度	
twitter-2010	TW	21.2	265.0	12	302,779	
soc-sinaweibo	SW	58.6	261.0	4	4,000	
orkut	OK	3.0	234.3	76.2813	33,313	
wikipedia-ru	WK	3.3	93.3	55.4067	283,929	
livejournal	LJ	4.8	69.0	28.257	22,887	
soc-pokec	PK	1.6	30.6	37.5092	20,518	
usaroad	US	24.0	28.9	2	9	
germany-osm	GR	11.5	12.4	2	13	
rmat876	RM	16.7	87.6	5	128,332	
均匀随机	UR	10.0	80.0	8	27	

表2。输入图

我们所有的实验（包括用于比较的基线实验）均在 IIT Madras AQUA集群上运行。每个计算节点的配置如下：Intel Xeon Gold 6248 CPU，配备分布在两个插槽上的 40 个硬件线程，时钟频率为 2.50 GHz，内存为 192 GB，运行 RHEL 7.6操作系统。所有使用 C++ 编写的代码均使用 GCC 9.2 编译，并启用 -O3 优化标志。各后端的版本如下：OpenMP 4.5、OpenMPI 3.1.6、CUDA 10.1.243。程序运行于 Nvidia Tesla V100-PCIE GPU，该 GPU 配备 5120 个 CUDA核心，均匀分布在 80 个时钟频率为 1.38 GHz 的 SM单元上，每个线程块拥有 32 GB 全局内存和 48 KB 共享内存。

5.1 OpenMP

表 3 展示了四种算法在不同框架中的代码运行时间。执行时间仅指算法处理时间，不包括图读取时间，并且是三次运行（对每个后端）的平均值。Galois框架无法加载最大的图 TW（因段错误而退出）。该框架在美国图上对 150 个源点进行介数中心性计算时也失败，并因 PTS 内存溢出错误而退出。

介数中心性。 BC类似于全对最短路径，可以通过从每个顶点出发、将其作为源顶点运行单源最短路径来实现。在大型图上完成BC计算需要数小时，有时甚至数天。根据实现存储每个源顶点的BC结果的方式，内存需求也可能随每个源顶点增加，与 $\sim V$ ~成正比。因此，文献中通常会给出从一个或少数几个顶点运行BC的结果。我们统计了源顶点数量为{20, 80, 150}时的BC结果。每个图的源顶点列表使用随机数生成器生成，并在各框架中保持一致，以便进行公平比较。在道路网络图和部分社交网络图上，StarPlat的BC代码优于Galois代码。Galois框架采用了一种调度策略，可确保线程之间实现更好的负载均衡。但对于工作负载均匀的任务，这种调度会增加运行时开销。Ligra采用了优化的正向广度优先搜索遍历，跳过节点的距离计算，仅计算每个节点的最短路径数量，从而节省并发距离计算所需的同步开销。这项优化使Ligra优于StarPlat和Galois。有趣的是，在大多数输入上，Green-Marl生成的代码优于其他三个框架。Green-Marl的正向广度优先搜索遍历经过优化，并根据特定标准选择每一层的遍历方式。

算法框架			TW	SW	OK	WK	LJ	PK	US	GR	RM	UR	总计
BC	20	Galois	-	**1.840**	8.272	3.377	3.209	1.477	38.488	15.368	**2.861**	29.528	104.420
		Ligra	11.200	13.130	5.936	3.470	4.060	**1.220**	**18.160**	10.430	4.250	4.736	76.592
		GreenMarl	**4.289**	3.492	**5.504**	**2.444**	**2.218**	1.173	18.823	**9.602**	3.187	**3.573**	54.305
		StarPlat	6.530	11.060	7.650	3.300	4.422	1.777	22.250	11.740	5.580	12.830	87.139
	80	Galois	-	**7.269**	50.171	13.548	12.139	5.596	258.447	75.969	**7.435**	85.534	516.108
		Ligra	48.700	48.760	24.100	14.230	16.160	4.550	**73.230**	40.500	10.900	19.130	300.260
		GreenMarl	**18.189**	12.172	**21.094**	**9.353**	**8.578**	**4.107**	75.450	**39.704**	7.863	**14.184**	210.690
		StarPlat	33.240	44.270	32.150	13.050	16.550	6.810	86.400	47.290	18.030	50.260	348.050
	150	Galois	-	**14.598**	78.632	25.403	23.816	11.132	-	138.052	12.365	116.700	420.698
		Ligra	83.400	90.360	44.630	26.260	31.500	8.996	**124.66**	76.530	17.700	35.800	539.836
		GreenMarl	**32.990**	23.110	**38.612**	**17.709**	**17.231**	**8.184**	139.090	**74.506**	**11.069**	**26.186**	388.670
		StarPlat	61.625	84.430	57.664	24.520	32.290	13.553	160.865	80.111	32.136	94.540	641.734
PR		Galois	-	**0.510**	**0.647**	**0.371**	**0.474**	**0.156**	**0.607**	**0.224**	**0.324**	**0.443**	3.756
		Ligra	25.600	162.660	5.050	3.930	3.623	0.836	2.050	0.822	5.880	0.942	211.393
		GreenMarl	**0.585**	7.211	1.437	0.512	0.585	0.263	1.235	0.525	0.821	0.688	13.862
		StarPlat	1.752	9.002	1.213	0.473	0.509	0.236	1.600	0.667	0.883	0.619	16.954
SSSP		Galois	**0.522**	**0.132**	**0.404**	**0.203**	**0.205**	**0.099**	**19.387**	**6.798**	**0.133**	**0.480**	28.363
		Ligra	10.7	0.148	5.136	1.846	3.89	1.683	283.000	9.043	2.74	10.400	328.586
		GreenMarl	2.182	0.891	1.048	1.16	0.761	0.292	193.548	48.349	0.464	1.361	250.056
		StarPlat	5.831	1.437	1.850	1.759	2.412	0.846	294.303	53.395	1.369	5.237	368.439
TC		Galois	-	**56.432**	33.110	**46.168**	9.811	3.008	0.061	**0.020**	184.260	2.350	335.220
		Ligra	2103.333	188.660	**22.800**	97.360	10.460	1.926	0.147	0.0698	**130.330**	1.706	2556.792
		GreenMarl	11611.029	4257.498	137.559	4564.568	29.426	12.705	0.065	0.021	5647.156	1.435	14650.430
		StarPlat	1414.323	59.925	23.420	111.430	**7.544**	**1.559**	**0.059**	0.024	158.760	**1.176**	1778.220

表3. StarPlat 的 OpenMP 代码与 Galois、Ligra 和 Green-Marl 的性能对比（20 个线程）。所有时间均以秒为单位。运行 BC 时，源点数量为 {20, 80, 150}。

SSSP	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR
	4.127	0.127	1.503	0.633	2.315	0.822	72.654	9.641	1.319	4.477

表4. StarPlat 的单源最短路径 OpenMP 代码在静态调度下的运行时间（秒）

PageRank。 在各种输入上，StarPlat生成的PR代码与Green-Marl的代码相比具有竞争力。有趣的是，Ligra的实现明显较慢，耗时也显著更长。这主要是由于将每个顶点的PR值计算与连续两次迭代PR值之差的计算进行了循环分离。总体而言，Galois（比StarPlat生成的代码快约 2×）优于所有其他基准实现。这是因为它对顶点的PR值进行原位更新，从而实现更快的收敛。StarPlat、Green-Marl和Ligra采用类似的处理方式，使用双缓冲更新PR值。

单源最短路径。 我们发现，GaloisSSSP 比 StarPlat 的代码以及其他对比基准更快。这得益于 Galois 框架中针对应用的优先级调度 [23]。例如，按距离升序处理任务可以减少额外工作的总量。GreenMarl 和 StarPlat 的单源最短路径计算实现几乎相同。两者都采用密集推送配置来处理顶点，因此需要遍历所有顶点以检查它们是否处于活动状态。对于道路网络而言，这通常代价高昂，因为每次迭代中活动前沿较小。此外，由于道路网络的直径较大，这些额外开销会在多次迭代中累积。然而，对于几乎所有图，GreenMarl 的表现都优于 StarPlat。GreenMarl 使用带退避策略的自旋锁实现，以节省不必要的 CPU 周期。StarPlat 也采用无锁

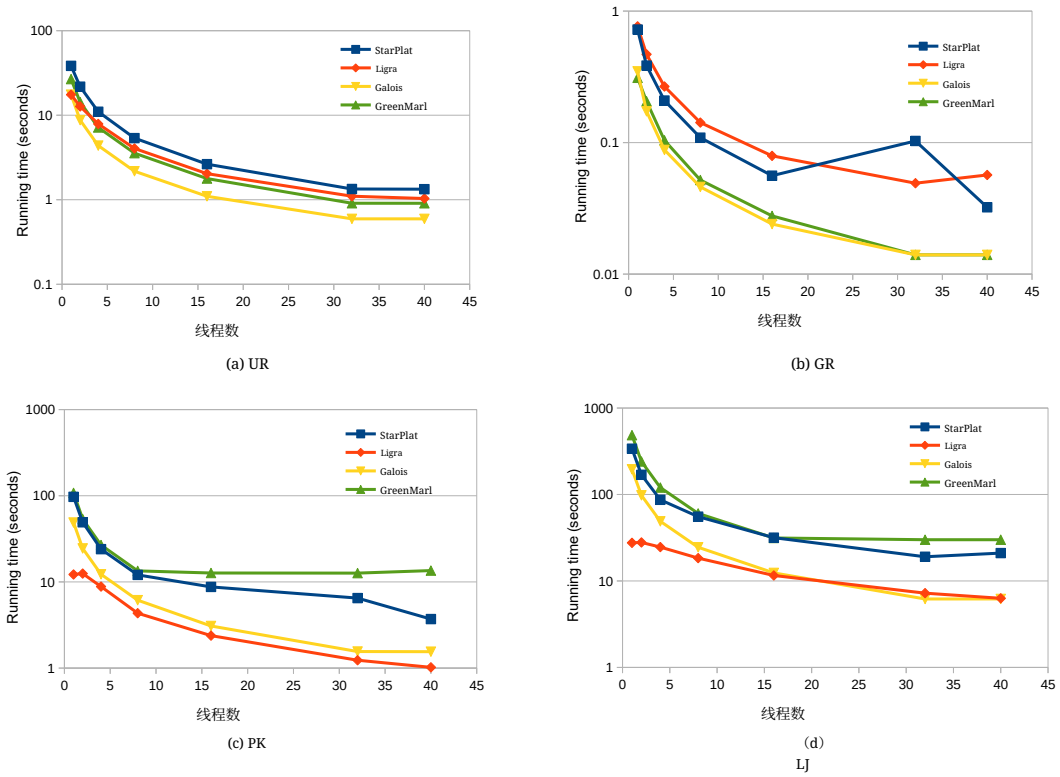

图 17. 不同类型的图在不同线程数下运行时间的对数图（注意 y 轴刻度不同）。

基于原子操作的实现，但不必要的更新更加严重，由于伪共享而导致性能下降。Ligra 的性能与其他基准测试相比并不具备很强的竞争力。Ligra 根据前沿大小在稀疏和稠密边处理之间切换。但对于给定的图套件而言，除道路网络外，这种方向优化并未带来显著的性能提升。

默认情况下，StarPlat 生成的 OpenMP 代码采用动态调度。这种方式在各种算法和图类型中总体上都能取得良好的效果。然而，如表 4所示，单源最短路径（SSSP）代码采用静态调度时，整体性能似乎更好。对于美国（US）和 GR 这类大直径图，这种差异尤为明显，执行时间从一分钟以上缩短至几秒钟。

三角形计数。Galois、StarPlat 和 Green-Marl 在三角形计数中采用节点迭代模式。另一方面，Ligra 采用基于边迭代器的版本，据称这种版本更适用于度数分布偏斜的图，因为基于边的版本具有更好的负载均衡。我们观察到，对于不同的图，表现最佳的框架也各不相同。有趣的是，Green-Marl 生成的代码性能明显较差。

算法		框架	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR	总计
BC	20	Galois	2814	3502	1733	54776	2962	2331	>95小时	>12小时	1006	2611	-
		StarPlat	**442**	**13**	**681**	**244**	**197**	**52**	**917**	**353**	**437**	**75**	3411
	80	Galois	12375	14043	6999	>12小时	11320	9035	>12小时	>12小时	2479	10249	-
		StarPlat	**2102**	**49**	**2705**	**978**	**764**	**194**	**3741**	**1489**	**975**	**299**	13298
	150	Galois	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	-
		StarPlat	**3816**	**94**	**4973**	**1829**	**1498**	**382**	**6782**	**2713**	**1442**	**559**	24088
PR		Galois	398	761	314	376	551	479	257	296	244	268	3943
		StarPlat	**57**	OOM	**23**	**11**	**8**	**2**	**2**	**0.3**	**44**	**3**	-
SSSP		Galois	**33**	11	**30**	63	**27**	24	**1933**	82	**20**	49	2272
		StarPlat	152	**0.4**	104	**35**	47	**15**	2302	**27**	70	**17**	2769
TC		Galois	**71275**	**436**	**327**	**884**	**174**	**19**	7	9	**1238**	**10**	74380
		StarPlat	>24 小时	>72小时	64114	>24 小时	5741	2420	**4**	**0.6**	>24小时	33	-

表 5. StarPlat 的 MPI 代码性能与 Galois 的对比（96 个进程）。所有时间均以秒为单位。BC 使用的源点数量为 {20, 80, 150}。由于采用双缓冲，StarPlat 在 SWin PR 上运行时内存不足。

5.2 MPI

表中给出了 StarPlat 的 MPI 代码与分布式 Galois 框架在四种算法上的执行时间对比5。执行时间不包括读取图和分发图所需的时间。执行时使用 OpenMPI 3.1.6，算法运行在 96 个进程上。

介数中心性。 从表中可以看出，对于全部 10 个图，与 Galois框架相比，StarPlat 的 BC 代码执行时间短得多。当源点数量为 20 时，Galois 代码在图 TW 和 GR 上分别运行 96 小时和 12 小时后仍未完成。同样，当源点数量为 80 时，Galois 代码在运行 12 小时后仍未能给出图 WK、US 和 GR 的结果。由于根据源点数量为 20 和 80 时的执行时间，预计使用 150 个源点处理全部 10 个图所需时间将远超 StarPlat 的 BC 代码，因此没有尝试运行 Galois 代码。

PageRank。对于除 SW 之外的所有图，StarPlat 生成的 PR 代码性能都优于 Galois PR 代码。由于双缓冲，StarPlat 在 SW 上运行 PR 时会耗尽内存。我们目前正在研究解决方法。

单源最短路径。 通过比较 StarPlat 与 Galois 的单源最短路径代码的执行时间，我们可以观察到，对于 TW、OK、LJ、US 和 RM 这 5 个图，Galois 的性能更好，而对于其余 5 个图，StarPlat 的性能更好。StarPlat 所需的迭代次数少于 Galois。这是因为在单次外层循环迭代中，进程的本地节点内可以进行多层距离更新。

三角形计数。 StarPlat 生成的三角形计数代码效率较低，因为它会多次遍历每个节点的邻居，并检查这些节点之间的邻接关系。因此，StarPlat 的三角形计数代码运行时间远长于 Galois。对于 US 和 GR 这两个道路图，StarPlat 的三角形计数耗时少于 Galois 代码。对于 TW、SW、WK 和 RM 等一些图，StarPlat 的代码即使运行 24 小时后仍未能得出结果。

5.3 CUDA

表6 展示了三种框架 LonestarGPU、Gunrock 和 StarPlat 在我们的十个图上运行四种算法的绝对运行时间。运行时间包括 CPU-GPU 数据传输。

介数中心性。 在十个图中的八个上，StarPlat生成的代码性能优于基于库的Gunrock代码。在两个道路网络上，Gunrock的表现明显优于StarPlat。我们利用CUDA的grid同步

算法		框架	TW	SW	OK	WK	LJ	PK	US	GR	RM	UR	总计
BC	1	LonestarGPU	-	-	-	-	-	-	-	-	-	-	-
	1	Gunrock	2.122	4.237	0.525	0.535	0.548	0.317	**2.811**	**1.750**	1.238	0.944	15.027
	1	StarPlat	**0.002**	**0.004**	**0.149**	**0.153**	**0.078**	**0.029**	17.656	6.359	**0.225**	**0.079**	24.734
	20	StarPlat	6.992	2.279	2.762	3.014	1.298	0.534	369.701	126.485	2.949	1.593	517.607
	80		28.179	9.332	11.331	12.050	4.886	1.907	1444.656	518.968	6.509	6.372	2044.189
	150		55.548	MLE	21.241	27.271	9.609	3.794	2636.453	978.758	9.912	11.957	3754.543
PR		LonestarGPU	-	**0.240**	0.363	**0.104**	**0.225**	**0.240**	**0.832**	**0.294**	**0.240**	**0.240**	2.778
		Gunrock	15.230	36.910	2.430	2.460	2.952	1.085	13.345	6.499	9.170	5.487	95.568
		StarPlat	**4.081**	7.112	**0.256**	1.780	1.300	0.257	3.420	0.679	0.891	0.257	20.033
SSSP		LonestarGPU	0.045	0.077	0.217	0.058	0.084	0.037	**0.162**	**0.091**	0.129	0.183	1.083
		Gunrock	2.272	4.057	0.616	0.556	0.562	0.311	1.283	1.140	1.034	0.915	12.746
		StarPlat	**0.001**	**0.002**	**0.078**	**0.044**	**0.027**	**0.012**	1.667	0.695	**0.120**	**0.028**	2.674
TC		LonestarGPU	-	31.990	2.998	2.771	**0.110**	**0.039**	11.874	5.695	**1.270**	0.499	57.246
		Gunrock	**67.718**	7.369	**0.843**	**0.997**	0.850	0.404	1.490	0.712	3.200	1.040	84.623
		StarPlat	10540.002	**1.410**	46.700	4.009	3.006	0.655	**0.001**	**0.001**	824.620	**0.034**	11420.430

表6. StarPlat 的 CUDA 代码与 LonestarGPU 和 Gunrock 的性能对比。所有时间单位均为秒。LonestarGPU 未实现介数中心性，并且无法加载最大的图 TW。

为我们带来的优势，而Gunrock则严重依赖这种批量同步处理。Gunrock的迪杰斯特拉算法在道路网络上表现出色。另一方面，在社交图和随机图上，我们的实现表现更好。我们还展示了多个不同规模（20、80和150）的源点下的性能。除了soc-sinaweibo之外，StarPlat生成的代码与其他框架持平或表现更佳。最后，与Gunrock和LonstarGPU不同，StarPlat支持从一组源点计算介数中心性。

PageRank。 我们观察到，这三个框架的相对性能表现一致，其中，手工编写的 LonestarGPU 代码优于另外两个框架，而 StarPlat 的性能优于 Gunrock。StarPlat 采用双缓冲方法，读取当前的 PR值，并生成下一次迭代所需的 PR值。这种分离方式减少了更新过程中的同步需求，但需要在各次迭代之间设置屏障。LonestarGPU 则采用 PR值的原位更新方式，因此收敛速度更快。

单源最短路径。 Gunrock 使用迪杰斯特拉算法，借助两级优先队列计算最短路径。我们在 StarPlat 中编写了贝尔曼-福特算法的一个变体。因此，这种比较可能并不十分恰当。但我们仅从应用角度比较二者——以最短执行时间计算从源点出发的最短路径。在这十个图上，LonestarGPU 和 StarPlat 的性能都优于 Gunrock。在 LonestarGPU 和 StarPlat 之间，没有明显的优胜者。事实上，它们的执行时间非常相近。

三角形计数。与其他三种算法不同，TC 不是一种基于传播的算法。此外，其特点是内核中包含一个双重嵌套循环（见图20）。还需要进行一次迭代来检查边（第7行）；如果压缩稀疏行表示中的邻居已排序，则可采用线性方式或二分查找来实现。由于最内层循环存在这种差异，不同实现之间的耗时差别可能十分明显，我们在三个框架中都观察到了这一现象。它们在这十个图上的性能表现各异，没有一种框架能成为全面领先者。

6 相关工作

我们在接下来的三个小节中根据目标后端对相关工作进行分类，并在第 6.4 节讨论通用框架。

6.1 OpenMP

Ligra [29] 是一种轻量级图处理框架，专为共享内存并行/多核机器设计，能够简化图遍历算法的编写。该框架提供了两个非常简单的例程，分别用于边映射和顶点映射。这些例程可应用于任意顶点子集，因此该框架很适合许多针对顶点子集运行的图遍历算法。从抽象层面看，顶点子集以一组整数标签的形式维护，用于标识其中包含的顶点；而顶点映射例程则会将用户提供的函数应用于每个整数。

Galois 是一种用于图分析的轻量级基础设施， [22] 它支持一种比受限模型更通用、表达能力更强的编程模型。该基础设施支持基于应用特定优先级对细粒度任务进行自主调度，并提供一个可扩展数据结构库。现有的领域特定语言可以在 Galois 系统之上实现，以实现高吞吐量。可以设想为 StarPlat 构建一个 Galois 后端。

Green-Marl [14] 是一种用于共享内存图处理的领域特定语言。依赖 BFS/DFS 遍历的程序可以使用该语言中相应的构造简洁地编写。不过，对于其他图算法，用户必须显式定义对顶点或边的迭代。因此，算法描述可以借助该语言提供的构造直观地编写，同时展现固有的数据级并行性。StarPlat 借鉴了该语言的一些理念，并将其性能与 Green-Marl、Ligra 和 Galois 进行了定量比较。

GraphIt [33] 是一种用于图分析的领域特定语言，通过让程序员能够轻松找到适合其特定算法和输入图的最佳优化组合，实现高性能。它本质上将计算与调度分离。算法使用算法语言进行指定，性能优化则使用单独的调度语言进行指定。

6.2 MPI

D-Galois 是一个分布式图处理系统，它通过将 Gluon（一种通信优化技术）与 Galois 共享内存图处理系统 [8] 对接而形成。我们在实验中将其与 Gluon 进行比较。

Pregel 和 Giraph 图处理框架受 BSP 编程模型 [21, 27] 的启发。Pregel 将图处理计算分解为一系列迭代，在这些迭代中，顶点接收上一轮迭代发送的数据，改变其状态，并向其出邻居发送消息，这些消息将在下一轮迭代中被接收。GPS 为 Pregel 框架 [28] 提供了额外功能，例如支持全局计算、动态图重新分区，以及对大型邻接表进行重新分区。

编译器会自动将使用 Green-Marl 领域特定语言编写的一类程序（称为 Pregel 规范形式）转换为 Pregel 代码 [15]。它还会应用一些转换，例如边翻转、拆解嵌套循环、转换随机访问，以及转换广度优先搜索和深度优先搜索遍历，从而将非 Pregel 规范形式的模式转换为 Pregel 规范形式。另一个名为 DisGCo 的编译器可以将所有 Green-Marl 程序转换为等效的 MPI RMA 程序 [26]。在将 Green-Marl 程序转换为 MPI RMA 代码时，它处理了语法差异、内存视图差异、串行代码与并行代码交错，以及图表示等挑战。

DH-Falcon DSL 及其编译器 [7] 为异构分布式环境中的形态图算法和非顶点中心图算法提供了规范与实现。Gemini [35] 是一种分布式图分析方案，它采用拉取式与推送式计算方法，实现了兼顾密集与稀疏模式处理的混合方法。此外，该方案使用优化的图表示和分区方法，实现节点内及节点间的负载均衡。

GraphLab 提供了一种容错的分布式图处理抽象，可针对机器学习算法执行动态图的异步处理 [20]。PowerGraph [10]则提出了另一种并行分布式图处理抽象，用于处理幂律图，并采用缓存机制和分布式图放置机制 [19]。GraphChi 支持在基于磁盘的系统上使用滑动方法异步处理动态图。

6.3 CUDA

Gunrock [32] 是一个图处理库，采用以数据为中心的抽象，对边前沿和顶点前沿执行操作。Gunrock 的所有操作均为批量同步操作，通过在前沿内的值上进行计算，或使用以下三个函数创建一个新前沿来影响前沿：筛选、计算和推进。Gunrock 图处理库通过合并访问和尽可能减少线程分歧，构建高效的前沿操作实现。

LonestarGPU [5] 是一组图分析 CUDA 程序。它采用多种与计算、内存和同步相关的技术，以提升底层图算法的性能。我们对 StarPlat、Gunrock 和 LonestarGPU 进行了定量比较。

Medusa [34] 是一个简化 GPU 计算任务的软件框架。与 Gunrock 类似，它提供可供构建的应用程序接口，以构造各种图算法。Medusa 利用了 BSP 模型，并提出了一种新的 EVM（边-消息-顶点）模型。在该模型中，局部计算在顶点上执行，计算则通过沿边传递消息来推进。

CuSha [17] 是一个图处理框架，采用两种图表示：G-Shards 和 Concatenated Windows（CW）。G-Shards 借鉴了一种近期为非 GPU 系统提出的思路，将图划分为称为分片的有序边集。为提高处理稀疏图时的 GPU 利用率，CW 是一种创新格式，可改进分片的使用方式。CuSha 通过在流式多处理器上并行处理多个分片，充分发挥 GPU 的计算能力。CuSha 面向大型图并行处理的架构允许用户创建顶点中心计算并将其接入，从而简化编程。与最先进的虚拟线程束中心方法相比，CuSha 在加速方面显著胜出。

MapGraph [9] 是一个并行图编程框架，提供高层抽象，帮助编写高效的图程序。它采用 SOA（数组结构）确保合并内存访问，并使用基于 GAS（Gather-Apply-Scatter）抽象的动态调度策略。动态调度可提升内存性能，并根据顶点度将工作负载分配给线程。

6.4 通用框架

Kakkos 3 [31] 是一个以 C++ 编写的编程模型，旨在为所有主流高性能计算平台（CUDA、HIP、SYCL、HPX、OpenMP 和作为后端的 C++ 线程）开发具有性能可移植性的应用程序。它为代码的并行执行和数据管理提供了抽象。不过，Kokkoskernels 主要用于基于矩阵的操作或算法，直到最近才开始支持着色算法，许多其他复杂算法目前尚未得到支持。在理念上，Kokkos 与 StarPlat 有相似之处，但前者是通用的，并非领域特定语言。

Graphit [33] 在最新版本中新增了对 GPU 和 CPU 代码生成的支持。它通过一种新颖的 GPU 调度语言和一个编译器，扩展了 GPU图处理框架的优化空间；该编译器支持多种优化，例如结合负载均衡、边遍历方向、活跃顶点集创建、活跃顶点集处理顺序和内核融合。

Julia [3] 是一种为高性能而设计的高级动态编程语言。它非常适用于数值分析和计算科学。它支持并发、并行和分布式计算（支持或不支持 MPI 均可）。Circle [1] 是一种新型 C++ 20 编译器。它通过添加许多新颖的语言特性扩展了 C++ 。Circle 是一种异构编译器，也支持 GPU 计算和着色器编程。借助 Circle，用户可以编写可移植到不同设备的 GPGPU 代码，并使用 Vulkan 计算（通过指向物理存储缓冲区的真实指针）；这是只有 Circle 编译器才实现的强大功能。Carbon [6] 是 C++ 的一种实验性后继语言，目前仍在开发中。其主要目标之一是支持现代操作系统平台、硬件架构和环境，并实现快速且可扩展的开发（类似于 Python 和 Rust）。Odin [4] 又是一种通用编程语言，具有独特的类型系统，专为高性能现代系统和面向数据编程而打造。它仍在开发中，支持数组结构（SoA）数据类型和数组编程。Chapel [12, 13] 是一种现代并行编程语言，旨在实现可移植性和可扩展性。Chapel 的首要目标是支持通用并行编程，并大幅提高大规模并行编程的效率。

上述语言的近期兴起，凸显了人们对一种简单语言的需求：无论用户从事哪个领域的工作，这种语言都能在不同的并行编程平台和架构之间移植，同时不影响用户的工作效率。

秉持类似理念，英特尔 [16] 和 Nvidia [24] 正致力于开发异构程序，将简单的 C++程序转换为可在各种架构（多核、GPU、FPGA，甚至 Arm 服务器）上运行的并行程序。不过，只有使用 STL 算法的程序才会被并行化。它们的编译器会根据目标平台，将标准 C++的 STL 算法替换为相应的并行版本。Intel OneAPI（包括 DPC++ 和 TBB）使用 C++ STL、并行 STL（PSTL）、Boost Compute 和 SYCL 进行并行化。Nvidia 的 nvc++ 使用不同的链接器，并通过命令行参数指定目标处理器。nvc++支持 ISO C++17，并通过 C++17 并行算法、OpenACC 和 OpenMP 实现面向 GPU 和多核 CPU 编程。

我们观察到，Gunrock 的 Essentials-cpp 和 Essentials [25]遵循相似的思路，采用现代 C++并行性构造和 lambda，并以其批量同步异步、数据中心的抽象模型构建。目前，它们实现了广度优先搜索和单源最短路径。

7 结论

我们介绍了 StarPlat，这是一种用于实现图分析算法的领域特定语言。该语言模型提供了各种构造，使用户能够以较高层次表达图问题，而无需从头开始实现常见模式的复杂细节。这也有助于针对这些模式提供在内存和时间方面均经过优化的解决方案。目前，我们支持在多核、分布式和众核后端上进行开发。我们讨论了如何将 StarPlat 的构造转换为各种后端。通过使用一系列大型图和四种不同的算法，我们进行了实验验证：生成代码的性能可与现有框架和库中经过手动调优及自动生成的实现相媲美。

尽管目前已有多种以编程语言和库形式提供的软件解决方案，用于开发图算法，但下一步的重要工作是构建能够灵活支持多平台开发的解决方案。StarPlat 正朝着这一方向努力。StarPlat 的未来发展方向是提高解决方案的可移植性，以支持 OpenACC 和 OpenCL 后端，并实现针对异构架构的自动代码生成。

致谢

我们衷心感谢对马德拉斯印度理工学院 HPCE 计算资源的使用。本研究得到印度国家超级计算任务计划资助，项目编号为 CS1920/1123/MEIT/008606。

参考文献

[1] Sean Baxter。2021。Circle C++编译器。https://www.circle-lang.org/[2] Maciej Besta、Michał Podstawski、Linus Groner、Edgar Solomonik 和 Torsten Hoefler。2017。推还是拉：减少图计算中的通信与同步。载于第26届高性能并行与分布式计算国际研讨会论文集（美国华盛顿特区）（HPDC '17）。计算机协会，美国纽约，93–104。https://doi.org/10.1145/3078597.3078616[3] Jeff Bezanson、Alan Edelman、Stefan Karpinski 和 Viral B. Shah。2017。Julia：数值计算的新方法。《SIAM评论》59卷第1期（2017），65–98。https://doi.org/10.1137/141000671[4] Ginger Bill。2016。Odin语言。https://odin-lang.org/[5] Martin Burtscher、Rupesh Nasre 和 Keshav Pingali。2012。GPU上非规则程序的定量研究。载于《2012年IEEE工作负载特征国际研讨会论文集》，IISWC 2012，美国加利福尼亚州拉霍亚，2012年11月4日至6日。IEEE计算机学会，美国纽约，141–151。https://doi.org/10.1109/IISWC.2012.6402918[6] Chandler Carruth。2022。Carbon语言。https://github.com/carbon-language/carbon-lang/[7] Unnikrishnan Cheramangalath、Rupesh Nasre 和 Y. N. Srikant。2017。DH-Falcon：面向分布式异构系统大规模图处理的语言。载于《2017年IEEE集群计算国际会议》，CLUSTER 2017，美国夏威夷州檀香山，2017年9月5日至8日。IEEE计算机学会，美国纽约，439–450。https://doi.org/10.1109/CLUSTER.2017.72[8] Roshan Dathathri、Gurbinder Gill、Loc Hoang、Hoang-Vu Dang、Alex Brooks、Nikoli Dryden、Marc Snir 和 Keshav Pingali。2018。Gluon：一种面向分布式异构图分析、优化通信的底层框架。载于第39届ACM SIGPLAN编程语言设计与实现会议论文集（美国宾夕法尼亚州费城）（PLDI 2018）。计算机协会，美国纽约，752–768。https://doi.org/10.1145/3192366.3192404[9] Zhisong Fu、Michael Personick 和 Bryan Thompson。2014。MapGraph：用于在GPU上快速开发高性能图分析的高级应用程序接口。载于图数据管理经验与系统研讨会论文集（美国犹他州斯诺伯德）（GRADES'14）。计算机协会，美国纽约，1–6。https://doi.org/10.1145/2621934.2621936[10] Joseph E. Gonzalez、Yucheng Low、Haijie Gu、Danny Bickson 和 Carlos Guestrin。2012。PowerGraph：自然图上的分布式图并行计算。载于第10届USENIX操作系统设计与实现研讨会（OSDI 12）。USENIX协会，美国加利福尼亚州好莱坞，17–30。https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez[11] Kshitij Gupta、Jeff A. Stuart 和 John D. Owens。2012。针对GPGPU工作负载的持久线程式GPU编程研究。载于《2012年创新并行计算》（InPar）。IEEE，美国纽约，1–14。https://doi.org/10.1109/InPar.2012.6339596[12] Guillaume Helbecque、Jan Gmys、Tiago Carneiro、Nouredine Melab 和 Pascal Bouvry。2022。面向性能的Chapel高生产力语言与传统编程环境比较研究。载于PMAM@PPoPP 2022：第十三届多核与众核编程模型及应用国际研讨会论文集，线上会议／韩国首尔，2022年4月2日至6日。ACM，美国纽约，21–29。https://doi.org/10.1145/3528425.3529104[13] Guillaume Helbecque、Jan Gmys、Tiago Carneiro、Nouredine Melab 和 Pascal Bouvry。2022。面向性能的Chapel高生产力语言与传统编程环境比较研究。载于PMAM@PPoPP 2022：第十三届多核与众核编程模型及应用国际研讨会论文集，线上会议／韩国首尔，2022年4月2日至6日。ACM，美国纽约，21–29。https://doi.org/10.1145/3528425.3529104[14] Sungpack Hong、Hassan Chafi、Eric Sedlar 和 Kunle Olukotun。2012。Green-Marl：一种易用且高效的图分析领域特定语言。载于第17届编程语言与操作系统体系结构支持国际会议论文集，ASPLOS 2012，英国伦敦，2012年3月3日至7日，Tim Harris 和 Michael L. Scott（编）。ACM，英国伦敦，349–362。https://doi.org/10.1145/2150976.2151013[15] Sungpack Hong、Semih Salihoglu、Jennifer Widom 和 Kunle Olukotun。2014。利用领域特定语言简化可扩展图处理。载于年度IEEE/ACM代码生成与优化国际研讨会论文集（美国佛罗里达州奥兰多）（CGO '14）。计算机协会，美国纽约，208–218。https://doi.org/10.1145/2581122.2544162[16] 英特尔。2021。oneAPI数据并行C++ (DPC++)。https://www.oneapi.io[17] Farzad Khorasani、Keval Vora、Rajiv Gupta 和 Laxmi N. Bhuyan。2014。CuSha：GPU上的以顶点为中心的图处理。载于第23届高性能并行与分布式计算国际研讨会论文集（加拿大不列颠哥伦比亚省温哥华）（HPDC '14）。计算机协会，美国纽约，239–252。https://doi.org/10.1145/2600212.2600227[18] Milind Kulkarni、Martin Burtscher、Rajeshkar Inkulu、Keshav Pingali 和 Calin Casçaval。2009。非规则应用中有多少并行性？载于第14届ACM SIGPLAN并行编程原理与实践研讨会论文集。《SIGPLAN通知》44卷第4期，3–14。https://doi.org/10.1145/1594835.1504181[19] Aapo Kyrola、Guy Blelloch 和 Carlos Guestrin。2012。GraphChi：仅用一台PC进行大规模图计算。载于第10届USENIX操作系统设计与实现研讨会（OSDI 12）。USENIX协会，美国加利福尼亚州好莱坞，31–46。https://www.usenix.org/conference/osdi12/

技术会议论文/演讲/kyrola[20] Yucheng Low、Joseph Gonzalez、Aapo Kyrola、Danny Bickson、Carlos Guestrin 和 Joseph M. Hellerstein。2012。分布式 GraphLab：一种用于云端机器学习的框架。arXiv:1204.6078 [cs.DB][21] Grzegorz Malewicz、Matthew H. Austern、Aart J.C Bik、James C. Dehnert、Ilan Horn、Naty Leiser 和 Grzegorz Czajkowski。2010。Pregel：一种用于大规模图处理的系统。载于2010年ACM SIGMOD 数据管理国际会议论文集（美国印第安纳州印第安纳波利斯）（SIGMOD '10）。计算机协会，美国纽约，135–146。https://doi.org/10.1145/1807167.1807184[22] Donald Nguyen、Andrew Lenharth 和 Keshav Pingali。2013。用于图分析的轻量级基础设施。载于ACM SIGOPS第24届操作系统原理研讨会，SOSP '13，美国宾夕法尼亚州法明顿，2013年11月3日至6日，Michael Kaminsky 和 Mike Dahlin（编）。ACM，英国伦敦，456–471。https://doi.org/10.1145/2517349.2522739[23] Donald Nguyen 和 Keshav Pingali。2011。为非规则算法合成并发调度器。载于第16届编程语言与操作系统体系结构支持国际会议论文集，ASPLOS 2011，美国加利福尼亚州纽波特海滩，2011年3月5日至11日，Rajiv Gupta 和 Todd C. Mowry（编）。ACM，美国纽约，333–344。https://doi.org/10.1145/1950365.1950404[24] Nvidia。2022。NVIDIA HPC SDK 22.7版。https://docs.nvidia.com/hpc-sdk/index.html[25] Muhammad Osama、Serban D. Porumbescu 和 John D. Owens。2022。并行图分析基础。载于图、体系结构、编程与学习研讨会论文集（GrAPL 2022）。IEEE，美国纽约，314–317。https://doi.org/10.1109/IPDPSW55747.2022.00061[26] Anchu Rajendran 和 V. Krishna Nandivada。2020。DisGCo：用于分布式图分析的编译器。《ACM体系结构与代码优化汇刊》17卷第4期，第28篇文章（2020年9月），26页。https://doi.org/10.1145/3414469[27] Sherif Sakr、Faisal Moeen Orakzai、Ibrahim Abdelaziz 和 Zuhair Khayyat。2016。使用Apache Giraph进行大规模图处理。Springer，美国纽约。https://doi.org/10.1007/978-3-319-47431-1[28] Semih Salihoglu 和 Jennifer Widom。2013。GPS：一种图处理系统。载于第25届科学与统计数据库管理国际会议论文集（美国马里兰州巴尔的摩）（SSDBM）。计算机协会，美国纽约，第22篇文章，共12页。https://doi.org/10.1145/2484838.2484843[29] Julian Shun 和 Guy E. Blelloch。2013。Ligra：一种用于共享内存的轻量级图处理框架。载于ACM SIGPLAN 并行编程原理与实践研讨会，PPoPP '13，中国深圳，2013年2月23日至27日，Alex Nicolau、Xiaowei Shen、Saman P. Amarasinghe 和 Richard W. Vuduc（编）。ACM，中国深圳，135–146。https://doi.org/10.1145/2442516.2442530[30] Christian R. Trott、Damien Lebrun-Grandié、Daniel Arndt、Jan Ciesko、Vinh Q. Dang、Nathan D. Ellingwood、Rahulkumar Gayatri、Evan Harvey、Daisy S. Hollman、Dan Ibanez、Nevin Liber、Jonathan R. Madsen、Jeff Miles、David Poliakoff、Amy Powell、Sivasankaran Rajamanickam、Mikael Simberg、Dan Sunderland、Bruno Turcksin 和 Jeremiah J. Wilke。2022。Kokkos 3：面向百亿亿次时代的编程模型扩展。《IEEE并行与分布式系统汇刊》33卷第4期（2022），805–817。https://doi.org/10.1109/TPDS.2021.3097283[31] Christian R. Trott、Damien Lebrun-Grandié、Daniel Arndt、Jan Ciesko、Vinh Q. Dang、Nathan D. Ellingwood、Rahulkumar Gayatri、Evan Harvey、Daisy S. Hollman、Dan Ibanez、Nevin Liber、Jonathan R. Madsen、Jeff Miles、David Poliakoff、Amy Powell、Sivasankaran Rajamanickam、Mikael Simberg、Dan Sunderland、Bruno Turcksin 和 Jeremiah J. Wilke。2022。Kokkos 3：面向百亿亿次时代的编程模型扩展。《IEEE并行与分布式系统汇刊》33卷第4期（2022），805–817。https://doi.org/10.1109/TPDS.2021.3097283[32] Yangzihao Wang、Andrew A. Davidson、Yuechao Pan、Yuduo Wu、Andy Riffel 和 John D. Owens。2016。Gunrock：一种GPU高性能图处理库。载于第21届ACM SIGPLAN并行编程原理与实践研讨会论文集，PPoPP 2016，西班牙巴塞罗那，2016年3月12日至16日，Rafael Asenjo 和 Tim Harris（编）。ACM，西班牙巴塞罗那，11:1–11:12。https://doi.org/10.1145/2851141.2851145[33] Yunming Zhang、Mengjiao Yang、Riyadh Baghdadi、Shoaib Kamil、Julian Shun 和 Saman P. Amarasinghe。2018。GraphIt：一种高性能图领域专用语言。《ACM程序设计语言汇刊》2卷，OOPSLA（2018），121:1–121:30。https://doi.org/10.1145/3276491[34] Jianlong Zhong 和 Bingsheng He。2014。Medusa：简化GPU上的图处理。《IEEE 并行与分布式系统汇刊》25卷第6期（2014），1543–1552。https://doi.org/10.1109/TPDS.2013.111[35] Xiaowei Zhu、Wenguang Chen、Weimin Zheng 和 Xiaosong Ma。2016。Gemini：一种以计算为中心的分布式图处理系统。载于第12届USENIX操作系统设计与实现研讨会（OSDI 16）。USENIX协会，美国佐治亚州萨凡纳，301–316。https: //www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

StarPlat 中的图算法

下面各图分别展示了介数中心性、PageRank 和三角形计数的 StarPlat 规范18、19和20 。此外，图21 展示了基于拉取的单源最短路径。

```
1 函数 计算介数中心性（图 g，节点属性 < 浮点型 > 介数中心性，节点集合 <g> 源节点集合）
{2g
型
```

5 节点属性 < 浮点型 > delta；6 g. 附加节点属性（delta = 0）；7 g. 附加节点属性

19 差值 += 值 - v.pageRank;20 v.pageRank_下一个 = 值; 21 }

2223

贝塔

g 19. PR 计算 tation in StarPlat

1 函数 计算传递闭包（图 g）{2 inttriangle_count = 0;3 4 对所有（v 在 g.

{6

n_

的

1 函数 computePullSSSP(图 g, 节点 src){2 节点属性 <整数 >dist; 3 节点属性 <boo

e);

un

_t

12

权重