Layph：通过分层图使增量图处理中的变化传播受限

Song Yu†、Shufeng Gong†⊢、Yanfeng Zhang✉†、Wenyuan Yu§、Qiang Yin‡、Chao Tian¶、Qian Tao§、Yongze Yan†、Ge Yu†、Jingren Zhou§

† 东北大学 § 阿里巴巴集团 ‡ 上海交通大学 ¶ 中国科学院⊢ 教育部医学影像智能计算重点实验室、东北大学{yusong, yanyongz}@stumail.neu.edu.cn, {zhangyf, gongsf, yuge}@mail.neu.edu.cn, {wenyuan.ywy, qian.tao,jingren.zhou}@alibaba-inc.com, {q.yin}@sjtu.edu.cn, {tianchao}@iscas.ac.cn

arXiv:2304.07458v1 [cs.DC] 15 Apr 2023

摘要—真实世界图不断演化，因此需要更新先前的分析结果以适应图变化。通过使用缓存的先前计算状态，增量图计算可以减少不必要的重新计算。然而，一个微小的变化可能传播到整个图，并导致大规模迭代计算。为了解决这一问题，我们提出了 Layph，一种双层图框架。上层是图的骨架，其规模远小于原始图；下层则包含若干互不相交的子图。Layph 将原始图上代价高昂的全局迭代计算限制在较小的图骨架以及少量随输入图变化而更新的子图上。这样，许多顶点和边无需参与迭代计算，从而显著降低计算开销并提升增量图处理的性能。实验结果表明，在响应时间方面，Layph 平均优于当前最先进的增量图系统 9.08× **（最高可达** 36.66×）。

索引词—**增量图处理、分层图、图骨架**

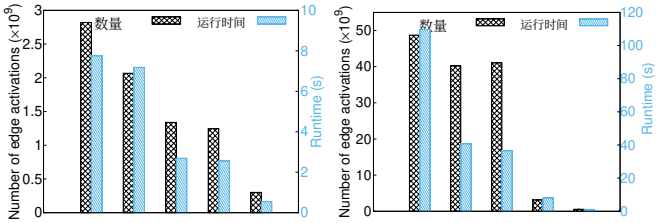
I. 引言

迭代图算法，例如，单源最短路径（SSSP）和 PageRank，已广泛应用于许多领域 [1]–[5]。真实世界图的结构持续变化，顶点和边会被任意插入或删除。这些变化通常很小，例如，在2021 [6]，，英文维基百科上有 640 万篇文章，但平均每天新增的文章仅有 580 篇。传统的经典图处理系统[7]–[13] 必须从头开始重新计算更新后的图。然而，计算结果之间存在大量重叠，

图更新前后的迭代。采用增量图计算来高效应对这些小幅变化是可取的。也就是说，先对原始图应用批处理迭代算法来计算结果，直至 G 收敛；然后使用增量算法，根据输入变化 ΔG 将结果调整为 G。

增量图计算可以通过使用记忆化迭代计算状态（例如，中间顶点状态或消息）来减少不必要的重新计算。增量图计算的优势推动了

✉Yanfeng Zhang 是通讯作者。



(a) 单源最短路径　(b) PageRank

图 1：不同增量图处理系统在单源最短路径和 PageRank 上的边激活次数与运行时间。

许多增量图处理系统的开发，例如 Kick- Starter [14], GraphBolt [15], Ingress [16], DZiG [17], 和 RisGraph [18]。这些系统会记忆化（中间或最终）顶点状态，并将其组织在能够捕获结果依赖关系的数据结构中，例如树（用于关键路径） [14], [18]或多层网络（用于表示每次迭代的依赖关系） [15]，[17]。借助这种结构，顶点/边更新会被传播，以迭代更新顶点的记忆化中间/最终状态。然而，在现有的增量图处理系统中，上游顶点/边更新可能会导致大量下游顶点/边状态更新。也就是说，微小的变化可能传播至整个图，并引发大规模迭代计算。

针对英国图进行 5000 次随机边更新（详情见表I），我们在五种先进的增量图处理系统（KickStarter [14], GraphBolt [15],DZiG [17], RisGraph [18], 和 Ingress [16]）以及一种从头开始在更新后的图上启动计算的重启系统上运行单源最短路径和 PageRank。这些系统的边激活数量和运行时间如图 1 所示。尽管更新量很小（$|\Delta G|/|G| = 5000/(9.4 \times 10^8) < 0.001\%$），但这些更新会在图中广泛传播并进行迭代，导致某些系统中产生大量边激活，其数量几乎接近重新启动迭代计算时的边激活数量。

我们通过图2中的示例，从经验上说明这一观察结果。图 2b 展示了基于图G的更新后的图，其中边（v_3, v_4）被删除，并新增了一条边（v_3, v_2）。如图2c所示，运行单源最短路径时，已有的

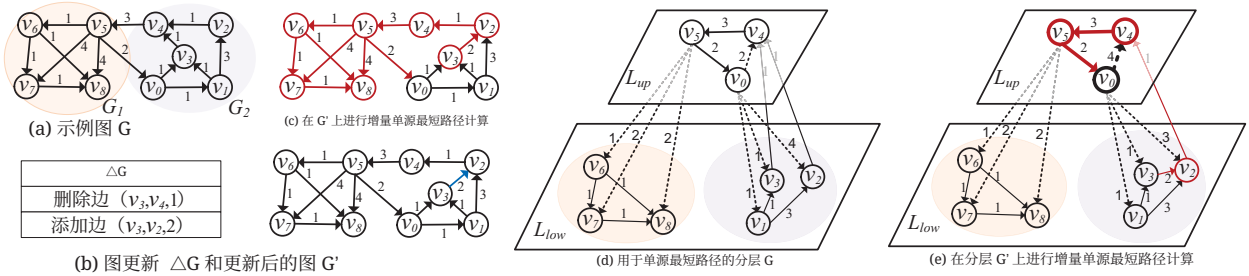


图 2：增量单源最短路径分层图示例，其中 v_0 是源顶点，G_1 和 G_2 是两个稠密子图。虚线表示两个顶点之间的快捷边，通过快捷边可以直接获得从一个顶点到另一个顶点的最短距离。每条边或快捷边上标注的数字表示其权重。在 (c) 和 (e) 中，红色连线或圆圈表示参与迭代计算的激活边、快捷边或顶点。

增量图处理系统 [14], [16], [18]会激活大多数顶点和边。随着迭代的进行，被激活的顶点可能会被更新多次，例如，v_4 及其下游顶点因在不同迭代中收到来自 v_2 的更新消息而被更新了两次。

挑战。根据上述观察和示例，我们可以发现，即使图变化非常小，也可能导致大量迭代计算，即便增量处理系统已经提供了先前缓存的顶点/边状态。主要原因是，在真实世界图中，顶点之间存在数跳之遥的直接或间接连接，因此很难限定受影响区域。真实图的固有属性从根本上限制了增量图计算的有效性。能否重构图结构，以提升增量图计算的性能?

直觉。在增量图计算中，图更新触发的消息会被迭代传播，以更新顶点的状态。当更新消息从入口顶点进入稠密子图时，子图中的大量内部顶点和边都会被激活并参与迭代计算。传入的消息可能需要经过多次迭代才能从出口顶点离开这个稠密子图。一个自然的想法是提取稠密子图的入口和出口顶点，并在它们之间构建快捷边，使消息能够直接穿过稠密子图传播，从而避免激活大量内部顶点和边。如图 2d 所示，我们提取 G_2 的入口顶点 v_0 和出口顶点 v_4，并在它们之间构建一条快捷边。这样，消息就可以通过快捷边直接穿过 G_2 传播。此外，我们还在每个子图的入口顶点与内部顶点之间构建快捷边。入口顶点可以累积传入的消息，最终通过快捷边一次性将这些消息传递给内部顶点。如图 2d 所示，在 v_5 累积所有传入的消息后，v_5 会一次性向 v_6-v_8 发送更新消息。如此一来，只有子图的入口顶点、出口顶点和离群点会参与全局迭代计算。

我们的解决方案。基于上述直觉，我们提出了 Layph，这是一种通过对图进行分层来处理增量图的框架。如图 2d 所示，Layph 将图分为两层：上层（L_{up}）和下层（L_{low}）。L_{up} 是原始图 G 的骨架，由子图的边界顶点和离群点组成，其规模远小于 G。L_{low}由一些互不相交的子图组成。L_{up}上的顶点与 L_{low} 上的顶点通过快捷边（虚线）或边相连。在 ΔG 更新 G 后，我们首先相应地更新分层图。修订消息仅在 L_{low} 上那些由 ΔG 更新的子图内部生成并传播。如图 2e 所示，修订消息从 v_3 生成，并在 G_2 内传播。随后，消息被上传到 L_{up}，例如，在图 2e 中，消息从 v_2 传播到 v_4。全局迭代计算在 L_{up} 上执行。与图 2c 中参与迭代计算的 10 条边相比，图 2e 中 L_{up} 上仅涉及 2 条边/快捷边。因此，在 L_{up} 上进行的全局迭代计算比在图 G' 上快得多。最后，更新被分配给 L_{low} 上的其他子图，例如，G_2。顶点状态通过快捷边直接更新，无需进行迭代计算。可以看到，Layph 仅在上层的小型骨架和少数由 ΔG 更新的子图（位于 L_{low} 上）中执行迭代计算。L_{low} 上的大多数顶点和边都不参与迭代计算。因此，Layph 能够有效加速增量图计算。

总而言之，我们的贡献如下。

- **分层增量图处理框架**。它将增量迭代计算限制在较小的范围内，即 受图更新影响的少数子图以及一个较小的骨架，从而大幅减少迭代过程中边激活的次数。（第三节和第五节）
- **有效骨架提取与快捷边自动推导**。我们设计了一种有效的骨架提取方法，通过复制顶点来缩小骨架规模。基于输入的以顶点为中心的程序，我们提出的框架能够自动推导快捷边的权重。（第四节）
- **高性能运行时引擎**。我们的运行时引擎 Layph 基于 Ingress [16]和阿里巴巴的 libgrape-lite [19]实现。与当前最先进的增量图处理系统相比，Layph 的加速比分别是：相较于 KickStarter [14]，2，为 3.13-15.82× ；相较于 RisGraph [18]，2，为 54-8.49×；相较于

GraphBolt [15], 2, 为 .92-32.93×；相较于 DZiG [17], , 为 .99-36.；相较于 Ingress [16], 为 1.06-7.22× 。（第六节）

二、预备知识

本节介绍迭代图计算和增量图计算所需的预备知识。

A. 迭代图计算

给定输入图 $G=(V,\ E)$，其中 V是一个有限顶点集，$E\subseteq V\times V$是一个边集。加权图中每条边 $(u,\ v)$ 的权重 $\in E$ 为 $w_{u,\ v}$；无权图中则为统一的值 1。一般而言，在累积模型中执行的迭代图算法 $\mathcal{A}$ 包含两类操作：即，消息生成 $\mathcal{F}$和消息聚合 $\mathcal{G}$ [9] [10] [20]。

$$\begin{aligned} m_{u,v}^{i} &= \mathcal{F}\big(m_u^{i-1}, w_{u,v}\big), \\ x_v^{i} &= \mathcal{G}\big(x_v^{i-1}, \{m_{*,v}^{i} | (*,v)\in E\}\big). \end{aligned} \tag{1}$$

其中 $m_u^{i-1}=\mathcal{G}(\{m_{*,u}^{i-1}|(*,\ u)\ \in E\})$。

消息生成操作 $\mathcal{F}$ 应用于每个顶点$u\ \in\ V$，根据收到的消息 m_u 和边权重 $w_{u,\ v}$，为每条出边 $(u,\ v)$ 准备消息 $m_{u,\ v}$。聚合操作 $\mathcal{G}$ 应用于每个目标顶点 v。它首先聚合所有以 v 为终点的消息，得到一条新消息 m_v；然后聚合旧顶点状态 x_v 和聚合后的消息 m_v，以更新顶点状态 x_v。这两个步骤会反复执行，直至收敛（即顶点状态趋于稳定）。总之，迭代图计算可表示为 $\mathcal{A}=(\mathcal{F},\ \mathcal{G},\ X^0,\ M^0)$，其中 $\mathcal{F}$和 $\mathcal{G}$是用于指定算法逻辑的操作，而 $X^0=\{x_v^0|v\ \in\ V\}$ 和$M^0=\{m_v^0|v\ \in\ V\}$ 分别是顶点状态和根消息的初始值。在输入图 G上的图计算可表示为 $\mathcal{A}(G)$。

假设 $\mathcal{A}$ 可以异步执行，那么它自然可以表示为公式（1），例如单源最短路径。否则，应将同步算法改写为累积模式并异步执行，例如 PageRank。已有一些工作 [9], [10] 将同步算法改写为异步累积模式。

示例1：我们展示两个示例算法。

(a) 单源最短路径。单源最短路径计算有向加权图中从给定源点 s 到所有顶点的最短距离 G。$\mathcal{A}$表示如下

- $\mathcal{F}(m_u,\ w_{u,\ v})=m_u+w_{u,\ v}$; $\mathcal{G}=$取最小值;
- $x_v^0=0$ 若 $v=s$，否则 $x_v^0=+\infty$;
- $m_v^0=0$ 若 $v=s$，否则 $m_v^0=+\infty$。

此处，状态 x_v of v 表示从源点 s 到 v 的最短距离，$w_{u,\ v}$ 表示边（$u,\ v$）的长度。初始时，对于 $v\ =\ s$，我们有 $x_v^0\ =\ m_v^0\ =\ 0$；对于所有 $v\neq s$，我们有$x_v^0=m_v^0=+\infty$。每个顶点 u 都会生成一条消息$m_{u,\ v}$ 并发送给每个邻居 v，该消息表示当前从源点出发的最短距离。每个目标顶点 v 都会聚合来自其入邻居的消息并通过取最小值来更新其状态 x_v。当所有顶点的最短距离值都不再变化时，算法终止。

*(b)*PageRank。PageRank 计算排名分数集合{$\text{PR}_v=d\times \text{sum}^{(u,v)\in E}\text{PR}_u/N_u+(1-d)\ |\ v\in V$}。其中，$d$是一个常数阻尼因子，$N_u$ 表示 u的出邻居数量。不同于利用幂法的原始 PageRank 算法，已证明与原始 PageRank 等价的异步 PageRank 算法 [10] 可表示如下

- $\mathcal{F}(m_u, w_{u,v})=m_u\times d/N_u$;　　$\mathcal{G}=\text{sum}$;
- $x_v^0=0,\ \forall v\in V$;　　$m_v^0=1-d,\ \forall v\in V$.

直观来说，每个顶点 v 使用其状态 x_v 存储 PageRank 分数。初始时，对于所有$v\in V$，我们有 $x_v^0\ =\ 0$ 和 $m_v^0\ =\ 1-d$。每当顶点 u 收到一条消息 m_u时，就会向每个邻居 v发送 $m_u\times d/N_u$。每个邻居 v按求和方式聚合来自其入邻居的消息，并通过累加聚合消息来更新其状态。当所有顶点状态都稳定后，算法终止。 □

公式（1）定义了异步迭代计算的以顶点为中心的格式。在此基础上，我们可以如下定义基于集合的迭代计算

$$\begin{aligned} M^i &= \mathcal{F}(M^{i-1}); \\ X^i &= \mathcal{G}(X^{i-1}\cup M^i). \end{aligned} \tag{2}$$

$X=\{x_v\ |\ v\in V\}$ 是顶点状态集合。 $M^0=\{m_v^0\ |\ v\in V\}$ 是每个顶点的根消息集合，$M^{k\neq 0}=\{m_v^k\}$ 是所有边上生成的消息集合。需要注意的是，在基于集合的格式中，$\mathcal{F}$ 和 $\mathcal{G}$的含义略有变化。$\mathcal{F}$ 是嵌入边信息的消息生成操作，因此只需要一个参数M。$\mathcal{G}$ 是分组聚合器（按顶点 ID 分组）。基于这种集合式计算，经过 n次迭代后的顶点状态集合 X 为

$$\begin{aligned} X^n =& \mathcal{G}\Big(X^0\cup(\mathcal{G}\circ\mathcal{F})(M^0)\cup\ldots\cup(\mathcal{G}\circ\mathcal{F})^n(M^0)\Big) \\ =& \mathcal{G}\Big(X^0\cup\bigcup^{n}(\mathcal{G}\circ\mathcal{F})^k(M^0)\Big), \end{aligned} \tag{3}$$

=其中 $\mathcal{G}\circ\mathcal{F}(\cdot)^{k\ 1}=\mathcal{G}(\mathcal{F}(\cdot))$且$(\mathcal{G}\circ\mathcal{F})^k$ 表示对$(\mathcal{G}\circ\mathcal{F})$应用 k 次。

消息传递视角。从消息传播的角度来看，每个顶点 v 的最终状态 x_v 是通过累积从所有顶点发出并沿不同路径传递的消息 M^0 得到的。在每次迭代中，也就是应用一次 $\mathcal{F}$ 和 $\mathcal{G}$时，一个消息会被处理并拆分为从一个顶点发往其直接邻居的多个消息（在 $\mathcal{F}$的作用下）。来自不同入邻居的消息会被聚合为一条消息（在 $\mathcal{G}$的作用下），并在下一次迭代中再次传播。同时，聚合后的消息会应用于顶点状态（在 $\mathcal{G}$的作用下）。这正是公式（1）所描述的过程。

B. 增量图计算

给定迭代图计算 $\mathcal{A}$ 及其增量对应项 $\mathcal{I}_{\mathcal{A}}$，当

输入图 G 通过 ΔG更新后，增量计算问题便会出现。令 $\mathcal{A}(G)$表示旧图 G 经过批处理图算法 $\mathcal{A}$得到的输出。增量计算的输入包括$\mathcal{A}(G)$和图更新 ΔG。于是有

$$\mathcal{I}_{\mathcal{A}}(\mathcal{A}(G), \Delta G) = \mathcal{A}(G \oplus \Delta G) \tag{4}$$

这意味着，基于旧结果 $\mathcal{A}(G)$ 和图更新 ΔG 执行的增量计算 $\mathcal{I}_{\mathcal{A}}(\mathcal{A}(G)\ \Delta G)$，预期会输出 $\mathcal{A}(G \oplus \Delta G)$，其中 $G \oplus \Delta G$ 表示将更新 ΔG 应用于 G。值得注意的是，增量化方案 $\mathcal{I}_{\mathcal{A}}$ 具有算法特定性，是从其原始算法 $\mathcal{A}$ 推导出来的。

输入批量更新 ΔG 由一组单元更新组成。为简化讨论，我们将序列中的单条边插入或删除视为一次单元更新，这可以模拟某些修改。例如，每次边权重发生变化，都可以视为先删除该边，再添加一条具有新属性的边。增量计算 $\mathcal{I}_{\mathcal{A}}$ 将识别旧输出 $\mathcal{A}(G)$ 中的变化，并针对 ΔG 对先前的计算结果进行修正。

消息传递视角。由方程 (3) 可知，输入变化会影响消息传播，因为 $\mathcal{F}$ 和 $\mathcal{G}$ 都与图结构相关，因此最终会改变顶点状态。由于边的插入、更新或删除，一组消息可能变得无效，而另一组消息可能会缺失。在原始图 G 上运行期间传输的旧消息，如果其传递路径因输入更新 ΔG 而消失，就称为无效消息。在 $G \oplus \Delta G$ 上运行时传输的新消息，如果未出现在 G 上的运行中，就称为缺失消息。在增量计算中，我们应首先找出所有无效和缺失的消息，然后通过生成取消消息（相应地，补偿消息），在 $G \oplus \Delta G$ 的受影响区域内进行修正，以撤销（相应地，重放）无效消息（相应地，缺失消息）的影响 [14]–[17]。本文将取消消息和补偿消息统称为修订消息。

III. 框架概述

本节首先介绍分层图框架的工作流程，然后分析 Layph 的优势。

Layph 工作流程。整体工作流程如图3所示。在增量图处理开始时，给定一个图 G，我们首先将该图划分为两层。上层（

L_{up}) 是图的骨架。 L_{up} 由所有稠密子图的入口/出口顶点、不属于任何稠密子图的顶点，以及它们之间的快捷边或边组成。下层（L_{low}）由所有互不相交的稠密子图构成。每个稠密子图的入口顶点（位于 L_{up} 上）和内部顶点（位于 L_{low} 上）通过 L_{up} 与 L_{low} 之间的快捷边相连。构建分层图的详细信息请参阅第 IV 节。然后，我们在分层图上执行增量图计算，其中

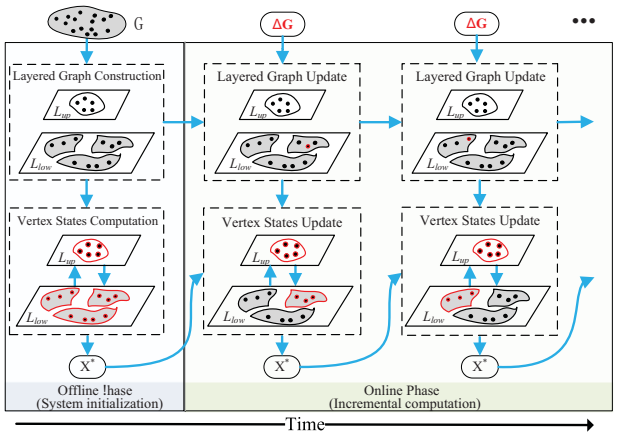


图3：Layph 工作流程。

包括两个步骤：i）分层图更新（第 IV 节）和 ii）顶点状态更新（第 V 节）。

分层图更新。给定原始图 G 的分层图 G，当 G 被 ΔG 更新时，应更新 G。这是因为随着图的变化，快捷边可能会发生变化，包括 L_{up} 上的快捷边以及 L_{up} 与 L_{low} 之间的快捷边。快捷边更新需要进行迭代计算，且仅在由 ΔG 更新的子图上执行。同时，由于这些子图彼此独立，快捷边更新可以很好地并行化。

顶点状态更新。图发生变化时，我们首先根据记忆的信息 [14]–[16], 推导出修订消息，然后在分层图上传播修订消息，以修正顶点状态。Layph 上的增量计算按以下方式执行。

- **修订消息上传**。为了将顶点推导出的修订消息应用到 L_{up} 上的顶点，需要将 L_{low} 上的修订消息上传至 L_{up}。与快捷更新类似，消息上传也可以并行执行，且只需在受 ΔG 影响的子图上执行。

- **在** L_{up} 上进行迭代计算。收到来自 L_{low} 的修订消息后，通过迭代计算传播修订消息，并修正 L_{up} 上顶点的状态。

- **修订消息分配**。在 L_{up} 上完成迭代计算后，每个子图的入口顶点（位于 L_{up} 上）会汇总所有修订消息。汇总后的修订消息通过快捷边从入口顶点分配给内部顶点（位于 L_{low} 上），以修正 L_{low} 上顶点的状态。

分析。从上述 Layph 工作流程可以看出，迭代计算仅在 L_{up} 及 L_{low}上的少数子图中执行。未被 ΔG 更新的子图中的顶点和边不会参与迭代计算，从而显著降低了计算开销。

IV. 分层图构建与更新

Layph 在分层图上执行。本节介绍如何构建分层图并对其进行增量更新。

A. 分层图构建

本节首先介绍如何提取上层的顶点，然后提供一种自动化的快捷边计算方法。

1）上层顶点提取：如我们在第一节介绍了 Layph 背后的直觉，并在第三节介绍了工作流程，我们应将稠密子图的入口顶点和出口顶点，以及不属于任何子图的顶点提取到上层，以构建图的骨架。这要求我们从原始图中找出所有稠密子图。在介绍稠密子图发现方法之前，我们先给出入口/出口/内部顶点和稠密子图的形式化定义。

定义 1（入口/出口/内部顶点）。给定图 $G(V,E)$的一个子图$G_i(V_i,E_i)$，其中 $V_i \subseteq V$和 $E_i \subseteq E$。 G_i 的入口顶点定义为 $V_i^I = \{v \mid (u,v) \in E, u \in V \setminus V_i, v \in V_i\}$， G_i 的出口顶点定义为$V_i^O = \{v \mid (v,w) \in E, v \in V_i, w \in V \setminus V_i\}$，而 G_i 的内部顶点定义为$\hat{V}_i = V_i - V_i^I - V_i^O$。

定义2（稠密子图）。给定输入图G（V,E）， G 的子图 G_i（V_i,E_i）是一个稠密子图，其中入口顶点数与出口顶点数的乘积小于 G_i中的边数，即$|V_i^I| \times |V_i^O| < |E_i|$。

我们对稠密子图的定义基于以下观察。对于子图 G_i 的每个入口顶点 $v \in V_i^I$，都需要通过快捷边将其连接到所有出口顶点。因此， G_i 中的快捷边数量等于入口顶点数与出口顶点数的乘积，即， $|V_i^I| \times |V_i^O|$。如果 G_i中的边很少，例如， $|V_i^I| \times |V_i^O| > |E_i|$，那么通过快捷边将消息从入口顶点传播到出口顶点，会比通过 G_i中的边传播更慢，因为更多的快捷边会导致更多的消息生成操作和聚合操作。

根据定义2，稠密子图要求内部边尽可能多，边界（入口/出口）顶点尽可能少。我们发现，稠密子图的要求与社区的要求相似。社区要求内部边尽可能多，外部边尽可能少。这启发我们采用社区发现算法来发现稠密子图。因此，本文使用社区发现算法（例如 Louvain [21]）来查找稠密子图。不过，社区发现算法可能会找到极大的子图，这会降低系统性能，因为极大的图可能导致负载不均衡。因此，我们在发现子图时添加阈值 K 以限制每个子图的大小，即 每个子图中的顶点数都小于 K。经验上，K 设为顶点总数的 0.002-0.2% 左右。我们还采用工作窃取技术来处理负载不均衡：空闲的处理线程会主动寻找可执行的工作。社区不一定是稠密子图。我们根据定义2，从社区发现算法发现的稠密子图候选集中选出稠密子图，即 $|V_i^I| \times |V_i^O| < |E_i|$。

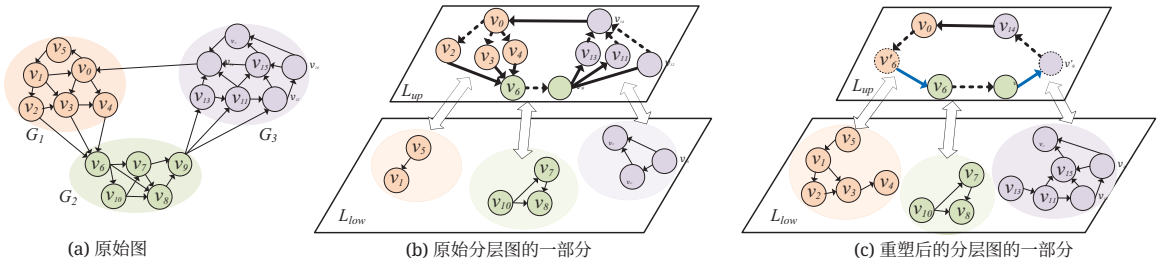
发现稠密子图后，将其中的内部顶点和边放入下层，而其他顶点和边即子图的入口/出口顶点，以及不属于任何稠密子图的顶点及其边，则提取到上层。

问题分析。虽然我们可以使用上述方法发现稠密子图，但该方法有一个关键局限：由于入口/出口顶点数量庞大，仍需建立大量快捷边。如图 4 所示，我们发现大多数边界顶点（入口/出口顶点）的度都很高，并且很可能与其他子图中的许多顶点相连，因此目标/源子图中会有许多入口/出口顶点。例如，顶点 v_9 的出度很高，且有 3 条出边连接到子图 G_3，因此子图 G_3 中会有 3 个入口顶点；顶点 v_6 的入度很高，且有 3 条入边源自子图 G_1，因此子图 G_1 中会有 3 个出口顶点。大量的入口/出口顶点会导致上层骨架规模庞大，如图 4b 所示，这会影响迭代计算的性能，并增加快捷边计算/更新的计算开销。

解决方案：顶点复制。图 4 展示了子图中存在一些共享同一源/目标顶点的入口/出口顶点。这启发我们提出一种顶点复制方法，以减少入口/出口顶点和快捷边的数量。其思路如图 4c 所示。发现稠密子图后，如果子图中共享同一源/目标顶点 v 的入口/出口顶点数量超过阈值，则会将源/目标顶点v （宿主顶点）复制到子图 G_i 中，作为一个代理顶点 v'。一个高度顶点可以在多个不同的稠密子图中拥有多个代理顶点。入口顶点和出口顶点都可以在其他稠密子图中拥有代理顶点。例如，在图 4c 中，入口顶点 v_6 在子图 G_1中有一个代理顶点 v_6'，作为新的出口顶点。原本，子图 G_1 中有 3 个出口顶点连接到入口顶点v_6，现在则只有一个出口顶点 v_6'。出口顶点v_9 在 G_3 中有一个代理顶点 v_9'，作为新的入口顶点。原本， G_3 中应该有 3 个入口顶点，它们都源自顶点 v_9，但现在只有一个入口顶点 v_9'。

通过在子图之间复制出口顶点或入口顶点，稠密子图中的一些边界顶点变成了内部顶点，并从 L_{up} 移动到 L_{low}。 L_{up} 上的图骨架大小大幅缩减。

2）快捷边计算：在上层中，每个子图只有入口顶点和出口顶点。为了能够正确、快速地将消息从入口顶点传播到出口顶点，需要用快捷边连接它们。在 L_{up}上的迭代计算过程中，入口顶点通过快捷边直接向出口顶点发送消息，而不会将消息向下传播到内部顶点。为了更新 L_{low}上的顶点状态，入口顶点会缓存



(a) 原始图 (b) 原始分层图的一部分 (c) 重塑后的分层图的一部分

图 4：上层重塑的示意图。虚线圆圈表示代理顶点。粗黑连线表示原始图中的加权/无权边。虚线连线表示子图中从入口顶点到出口顶点的快捷边。蓝色连线表示顶点与其复制的代理顶点之间的连接。在 (b) 和 (c) 中，为简化起见，我们用双向空心箭头表示 L_{up} 与 L_{low} 之间的快捷边和边集合。

这些本应传播到内部顶点的消息，然后在迭代计算终止后再将其向下传播到内部顶点。然而，将这些消息传播到所有内部顶点可能需要进行迭代计算。为了高效地将消息从入口顶点传播到内部顶点，我们也用快捷边连接它们。

根据上述讨论，分层图中有两种快捷边：1）从稠密子图的入口顶点到出口顶点的快捷边；2）从稠密子图的入口顶点到内部顶点的快捷边。从本质上说，这两种快捷边都连接稠密子图的入口顶点与其他顶点。因此，可以用同一种方法同时计算它们。在介绍快捷边计算方法之前，我们先给出快捷边的正式定义。

定义3（快捷边）。给定子图 G_i（V_i, E_i）以及到达入口顶点 V_i^I的输入消息向量 $M = \{m_u \mid u \in V_i^I\}$，快捷边 S_i 是从入口顶点 V_i^I到所有顶点 V_i的直接连接，即 $S_i = \{\vec{w}_{u,v} \mid u \in V_i^I, v \in V_i\}$，其中 $\vec{w}_{u,\ v}$ 是从顶点 u 到顶点 v的快捷边权重，使得

$$\mathcal{G}_{V_i}\big(\mathcal{F}_{S_i}(M)\big) = \mathcal{G}_{V_i}\Big(\bigcup_{k=1}^{\infty}(\mathcal{G}_{V_i} \circ \mathcal{F}_{E_i})^k(M)\Big), \tag{5}$$

其中，$\mathcal{F}_{S_i}$ 和 $\mathcal{F}_{E_i}$ 分别表示通过快捷边 S_i 和原始边 E_i 进行的消息传播，而$\mathcal{G}_{V_i}$ 表示在顶点集 V_i上的消息聚合。

在 G_i 中，从入口顶点 u 到顶点 v的快捷边权重 $\vec{w}_{u,\ v}$ 可通过以下方程计算

$$\vec{w}_{u,v} = \mathcal{G}_v\Big(\bigcup_{k=1}^{\infty}(\mathcal{G}_{V_i} \circ \mathcal{F}_{E_i})^k(m_u)\Big), \tag{6}$$

其中，$\mathcal{G}_v$ 是顶点 v上的分组聚合，m_u 是单位消息。这意味着，我们首先为入口顶点 u初始化一条单位消息m_u。然后，在子图 G_i 上执行迭代计算，将消息从 u 传播到 v，直到 G_i 中的所有顶点都不再接收消息，或收到的消息可以忽略。最后，v 接收的消息的聚合值可视为从 u 到 v的快捷边权重，即$\vec{w}_{u,\ v}$。单位消息m_u 应为 $\mathcal{F}$ 操作的单位元，以便进行初始化。如示例 2 所示，‘+’的单位元是 0。在 SSSP 中，消息的最小值由 u 发出并被 v 接收的消息对应的最短路径，是从 u到 v 的最短路径，即从 u 到 v的快捷边权重。为减轻用户负担，Layph 可以在无需用户干预的情况下，通过调用用户定义的 $\mathcal{F}$ 和 $\mathcal{G}$ 函数，自动完成快捷边计算（见 II-A ）。

示例 2：考虑在图2a所示的图上运行单源最短路径。在计算子图G_2内部的快捷边时，将单位消息 $m_{v_0} = 0$（作为“+”的单位元，因为 $\mathcal{F} = m_u + w_{u,\ v}$ 包含“+”）输入入口顶点 v_0。我们迭代执行 $\mathcal{F} = m_u + w_{u,\ v}$ 以传播消息，并使用 $\mathcal{G} = \min$ 聚合每个顶点接收到的消息。最后，如图2d所示，$\{v_1, v_2, v_3, v_4\}$上接收到的消息的聚合值分别为 $\{1, 4, 1, 2\}$，即 快捷边的权重为$\vec{w}_{v_0,\ v_1} = 1$，$\vec{w}_{v_0,\ v_2} = 4$，$\vec{w}_{v_0,\ v_3} = 1$，$\vec{w}_{v_0,\ v_4} = 2$。□

最后，我们给出分层图的形式化定义。

分层图。给定一个输入图 $G(V,\ E)$，以及一组稠密子图 $\{G_1(V_1,\ E_1)..... G_N(V_N,\ E_N)\}$，分层图由上层 $L_{up} = (L_V^{up},\ L_E^{up})$、下层 $L_{low} = (L_V^{low},\ L_E^{low})$以及$L_{up}$ 与 L_{low}之间的边构成，其中 L_V^{up}（分别是 L_V^{low}）为上层（分别为下层）的顶点集，而 L_E^{up}（分别是 L_E^{low}）为上层（分别为下层）的边集。

- 上层（L_{up}）。– 顶点集 $L_{up}^V = \bigcup_{i=1}^{N}\{V_i^I, V_i^O\} \cup \{V - \cup_{i=1}^{N} V_i\}$ 由所有稠密子图的入口顶点和出口顶点，以及不属于任何稠密子图的顶点组成。– 边集 $L_{up}^E = \bigcup_{i=1}^{N}\{\vec{w}_{u,v} \mid u, v \in S_i, u \in V_i^I, v \in V_i^O\} \cup$ 边集 $\{E - \bigcup_{i=1}^{N} E_i\}$

由每个稠密子图中从入口顶点到出口顶点的快捷边，以及不属于任何稠密子图的边组成。

- 低层（L_{low}）。– 顶点集 $L_{low}^V = \bigcup_{i=1}^{N}\{\hat{V}_i\}$ 由所有稠密子图的内部顶点组成。– 边集 $L_{low}^E = \bigcup_{i=1}^{N}\{E_i - \{u\ v \in E_i \mid u \in V_i\ v \in (,)\hat{},$由 $V_i^I \cup V_i^O\}\}$ 各子图内的边组成，但不包括从内部顶点指向入口/出口顶点的边。
- 连接 L_{up} L_{low} $L_{up\ low}^E = \bigcup_{i=1}^{N}\{\{\vec{w}_{u,v} \in$ 与。$S_i \mid u \in V_i^I, v \in V_i\} \cup \{u\ v \in E_i \mid u \in V_i\ v \in \hat{}(,)\hat{},\ V_i^I \cup V_i^O\}\}$ 由从入口顶点到内部顶点的快捷边，以及从内部顶点到

每个稠密子图内入口/出口顶点的边组成。

上层的规模（相对于 $|L_{up}^{V}|$ 和 $|L_{up}^{E}|$）预计远小于原始图的规模（相对于 $|V|$ 和 $|E|$）。例如，在图2中，上层包含 3 个顶点和 3 条边/快捷边，小于包含 9 个顶点和 14 条边的原始图。

分析。 由于引入了快捷边，Layph 将需要更多空间。额外的空间开销包括从入口顶点到每个子图中所有顶点的快捷边，即 也就是， $O(\sum_{i=1}^{N}(|V_i^I| \times |V_i|))$，其中 $|V_i^I|$ 是子图 G_i 的入口顶点数量，而 $|V_i|$ 是 G_i中的所有顶点数量。实际上，如图11a（见第VI-G节）所示，额外的空间开销始终小于原始图的空间开销。

B. 分层图更新

当 G 通过 ΔG更新时，分层图也需要更新。由于稠密子图的生成或消失，顶点可能会在两个层之间移动，例如，新生成子图的内部顶点会从L_{up} 移动到 L_{low}。 为了避免反复发现子图所带来的高昂开销，我们采用增量式社区检测方法（如 C-Blondel [22]或 DynaMo [23]）对稠密子图进行增量更新。在实践中， ΔG 的规模与 G相比非常小。较小的 ΔG不会对现有稠密子图产生显著影响。因此，只有在累积了足够多的 ΔG 时，我们才会更新稠密子图。然而，即使非常小的 ΔG 仍可能改变分层图中若干快捷边的权重。

快捷边更新。快捷边更新有三种类型。i）删除。如果某个入口顶点来自外部的所有入边都被删除，即，与外部的连接被切断，该入口顶点就会变成内部顶点，应移除从该顶点出发的快捷边。ii）添加。如果向某个内部顶点添加一条来自外部的入边，该内部顶点就会变成入口顶点。应计算它到子图中其他顶点的快捷边。iii）权重更新。如果子图内部存在边的添加或删除操作，则应更新快捷边的权重。

根据定义3，在每个稠密子图内部构建快捷边。此外，从公式（6）可以看出，每条快捷边在 G_i 上的权重仅取决于 G_i中的边和顶点，并且不同子图上的快捷边彼此独立。因此，我们只需更新受ΔG影响的子图上的快捷边，而且每个子图的快捷边都可以并行更新。删除或添加快捷边时，可以直接在子图内通过移除或计算快捷边来完成。对于权重更新，为避免冗余计算，我们采用增量方法进行更新。

根据公式（6），快捷边的权重通过迭代计算得出；从 u 到 v 的快捷边权重等于 v 通过从 u 到 v 的所有路径接收到的所有消息的聚合。在稠密子图内部添加或删除边之后，部分消息由 v 接收到的消息可能会变得无效或缺失。因此，快捷边的更新可以采用现有的增量计算方法 [14]–[16]。计算旧快捷边时，可以根据备忘信息推导出补偿消息和取消消息。这些消息将用于重新执行和撤销缺失及无效消息对顶点 v 的影响；由于稠密子图内部添加和删除边， v 接收到的消息中存在一些缺失和无效消息。

示例 3：考虑在更新后的图上运行单源最短路径，如图 2b 所示。由于 ΔG 只改变了 G_2，因此无需更新与 G_1 相关的快捷边。对于G_2， L_{up} 上的顶点无需改变，因为只有内部边发生变化，而且快捷边可以增量更新。因此，我们可以将旧快捷边的权重作为新快捷边的初始权重，即将 $\{\vec{w}_{v_0,v_1}, \vec{w}_{v_0,v_2}, \vec{w}_{v_0,v_3}\}$ 的初始值设为 $\{1,4,1,2\}$。由于边 $v_3 \to v_4$ 被删除，且 v_4 的状态取决于 v_3，因此需要生成取消消息m_{v_3,v_4}=$\perp$ （$\perp$ 表示该顶点需要重置为默认状态，即单源最短路径中的 ∞ ），并且m_{v_3,v_4} 将v_4 的状态设为 ∞ [14], [16], [18]。与此同时， v_4 将从其邻居 v_2收到消息$m_{v_2,v_4} = 5$ 。此外，由于边$v_3 \to v_2$ 被添加，因此需要生成补偿消息m_{v_3,v_2}=3。随后，所有这些修订消息都将在 G_2内部传播。最后，如图 2e 所示，在 $\{v_1, v_2, v_3, v_4\}$ 上收到的消息的聚合值分别为 $\{1,3,1,4\}$，即快捷边的新权重为$\vec{w}_{v_0,v_1}$=1,$\vec{w}_{v_0,v_2}$=3, $\vec{w}_{v_0,v_3}$=1,$\vec{w}_{v_0,v_4}$=4。□

V. 基于分层图的增量处理

本节将介绍 Layph 如何在分层图上执行增量图处理。

修订消息的推导。如方程 (3) 所示，最终顶点状态由接收到的消息决定，这些消息来自所有顶点，并沿不同路径传递。图更新后，由于消息传播路径发生变化，顶点接收到的消息也可能随之改变。增量图处理框架可以自动 [14], [16]或手动 [15], [17]获取修订消息，即 补偿消息和取消消息，并传播这些消息，以修正缺失和无效消息对顶点状态的影响 [15], [16]。对于修订消息，我们可以采用先前工作 [16]中提出的方法推导得出。

推导出修订消息后，我们借助 Layph 高效地传播这些消息。正如第三节所述，在 Layph 上传播修订消息分为三个步骤：1）消息上传；2）迭代计算；3）消息分配。

A. 消息上传

上层 L_{up} 仅包含部分顶点，而 L_{low} 上每个子图中的内部顶点 L_{up}不参与迭代计算。为了确保所有

L_{up} 上的顶点能够收敛并体现内部顶点的影响， L_{up} 上的迭代计算不仅应收集由 L_{up} 上的顶点推导出的修订消息，还应收集由内部顶点推导出的修订消息。因此，需要将 L_{low} 上稠密子图的内部顶点推导出的修订消息上传至 L_{up}。由于每个稠密子图的入口/出口顶点都位于 L_{up}上，因此可以通过将修订消息传播到入口/出口顶点来完成消息上传。

并非每个稠密子图中的所有内部顶点都与入口/出口顶点相连，因此，我们执行局部迭代计算，将内部修订消息传播到子图的入口/出口顶点。当入口/出口顶点收到的消息可以忽略时，迭代计算即告终止。上传消息后，入口顶点 V_i^I 和出口顶点 V_i^O 上累积的消息可视为其初始修订消息 即， $\mathbb{M}^0_{V_i^I \cup V_i^O} = \mathcal{G}_{V_i^I \cup V_i^O}\big(\bigcup_{k=1}^{\infty}(\mathcal{G}_{V_i} \circ \mathcal{F}_{E_i})^k(\mathbb{M}^0_{V_i})\big)$。再结合 L_{up}^V 上不属于任何稠密子图的顶点的初始消息 即 $\mathbb{M}^0_{V-\cup_{i=1}^N V_i}$， L_{up} 上顶点的初始消息可表示如下

$$\mathbb{M}^0_{L_{up}^V} = \Big(\bigcup_{i=1}^{N} \mathcal{G}_{V_i^I \cup V_i^O}\big(\bigcup_{k=1}^{\infty}(\mathcal{G}_{V_i} \circ \mathcal{F}_{E_i})^k(\mathbb{M}^0_{V_i})\big)\Big) \cup \mathbb{M}^0_{V-\cup_{i=1}^N V_i}, \tag{7}$$

其中 $\mathbb{M}^0_{V_i}$ 表示初始修订消息。

注释。无需在 L_{low} 上对所有子图执行消息上传，因为修订消息只会在受 ΔG [14]–[17] 影响的子图上生成。一般来说，由于 ΔG 的规模很小，受影响的子图数量也很少。对于所有受 ΔG 影响的子图，可以高效地并行执行消息上传，因为每个子图彼此独立。

示例 4： 以图2d中的 v_0 为源顶点，在分层图上运行单源最短路径，直到收敛。当图按图2b所示发生变化时，分层图会按图 2e 所示更新。此时，原始图中所有顶点的收敛状态都作为更新后的图中各顶点的初始状态，即 $\{x_{v_0}, ..., x_{v_8}\}$ 为 $\{0, 1, 4, 1, 2, 5, 6, 7, 7\}$。由于 G_1 未受到 ΔG 的直接影响，因此无需在 G_1 上推导修订消息。在 G_2 上，将生成一条取消消息$m_{v_3,\ v_4}$=⊥以及两条补偿消息$m_{v_2,\ v_4}$=5 和 $m_{v_3,\ v_2}$=3 。对于取消消息$m_{v_3,\ v_4}$=⊥，它会使 v_4 重置为默认状态（即 ∞），而所有依赖于 v_4 的顶点也会根据依赖树 [14], [16], [18] 重置为默认状态。随后，其余所有修订消息都将在 G_2 内传播，最终所有消息也会聚合到 L_{up} 上的出口顶点 v_4 ，即 m_{v_4}=4。此时， L_{up} 获得了 L_{low} 的所有修订消息，而 G_2 的 v_2 和 v_4 获得了新状态 x_{v_2}=3 和 x_{v_4}=4。□

B. 上层的迭代计算

消息上传后， L_{low} 上子图内部顶点推导出的修订消息已传播至 L_{up}。然而，根据方程 (7)，这些上传的消息仅缓存在稠密子图的入口顶点和出口顶点中。必须在 L_{up} 上执行迭代计算，以传播修订消息，使 L_{up} 上的其他顶点能够接收所有修订消息并修订其状态。

迭代计算仅在 L_{up}上执行，即 只有L_{up}^V 和 L_{up}^E 会参与迭代计算，而稠密子图的入口顶点和出口顶点也会参与迭代计算，因为它们位于 L_{up}^V上。当入口顶点收到消息时，它们不会向内部顶点发送消息，而是通过快捷边将消息传播到出口顶点。迭代计算完成后， L_{up} 上顶点的状态可表示如下

$$X^*_{L_{up}^V} = \mathcal{G}_{L_{up}^V}\Big(X^0_{L_{up}^V} \cup \bigcup_{k=1}^{\infty}(\mathcal{G}_{L_{up}^V} \circ \mathcal{F}_{L_{up}^E})^k(\mathbb{M}^0_{L_{up}^V})\Big). \tag{8}$$

根据下面的定理 1，我们可以看出，在 L_{up}上完成迭代计算后，顶点会收敛到与在原始图上执行迭代计算相同的状态。

定理 1： 对于方程 *(7)* 中定义的初始消息 $\mathbb{M}^0_{L_{up}^V}$ 和初始状态 $X^0_{L_{up}^V}$，在上层进行迭代计算后，上层顶点的收敛状态 $X^*_{L_{up}^V}$ $L_{up}(L_{up}^V, L_{up}^E)$ 与在更新后的图 $G \oplus \Delta G$ 上进行迭代计算后得到的状态相同。□

证明概要： 通过替换为 $\mathbb{M}^0_{L_{up}^V}$ 方程 (7)，每个已更新子图的内部顶点发出的初始消息会经由边界顶点传播出去。通过迭代应用 $\mathcal{F}_{L_{up}^E}$ 和 $\mathcal{G}_{L_{up}^V}$，这些初始消息无论来自内部顶点还是 L_{up} 的顶点，都会沿着 L_{up} 传播，最终聚合到 L_{up}上的顶点。 □

例5：图 2e 展示了在 L_{up}上的迭代计算。根据例4，我们得到 L_{up}上所有顶点的状态$\{x_{v_0}$=$0, x_{v_4}$=$4, x_{v_5}$=$\infty\}$ 和修订消息 $\{m_{v_4}$=$4\}$ 。然后基于这些初始状态，在 L_{up} 上执行迭代计算。首先，只有 v_4是活跃顶点，因为它有修订消息 $\{m_{v_4}$=$4\}$。v_4 是出口顶点，消息 m_{v_4}、$_{v_5}$=m_{v_4}+w_{v_4}、$_{v_5}$=7通过出边 (v_4, v_5)生成。 v_5 是入口顶点，它将消息 m_{v_4}、$_{v_5}$ 聚合到 m_{v_5} ，将自身的顶点状态从 x_{v_5}=∞ 更新为 x_{v_5}=7 ，并存储 m_{v_5}以进行消息分配（第 V-C 节）。 v_5 随后生成消息 m_{v_5}、$_{v_0}$=m_{v_5}+w_{v_5}、$_{v_0}$=9 并将其发送给 v_0。接收到 m_{v_5}、$_{v_0}$后，v_0 无法更新消息 m_{v_0} 。因此， L_{up} 上的所有顶点都达到收敛状态，即 $\{v_0^*$=$0, v_4^*$=$4, v_5^*$=$7\}$。□

C. 修正消息分配

由于迭代计算仅在 L_{up} 上执行，因此修订消息不会触及 L_{low} 上各子图的内部顶点，也就是说，内部顶点不会收到来自外部的修订消息。必须再执行一步，将外部消息应用于内部顶点。

尽管内部顶点不会收到来自外部的修订消息，但根据定理 1，每个稠密子图的入口顶点都已收到来自其他稠密子图中的顶点以及 L_{up} 的所有修订消息。为了使内部顶点能够收到外部修订消息，在迭代计算过程中，入口顶点会先缓存收到的消息，再通过快捷边将其传播到出口顶点。经过多次迭代后，入口顶点可能会缓存大量消息，因此我们只存储聚合后的消息。缓存的消息可表示如下

$$\mathbb{M}_{V^I} = \bigcup_{i=1}^{N} \mathcal{G}_{V_i^I}\Big(\bigcup_{k=1}^{\infty}(\mathcal{G}_{L_{up}^V} \circ \mathcal{F}_{L_{up}^E})^k(\mathbb{M}^0_{L_{up}^V})\Big). \quad (9)$$

最后，我们通过入口顶点与内部顶点之间的快捷边，将缓存于入口顶点中的消息发送给内部顶点。 L_{low} 上的顶点状态可表示如下

$$X^*_{L_{low}^V} = \mathcal{G}_{L_{low}^V}\Big(X_{L_{low}^V} \cup \bigcup_{i=1}^{N}(\mathcal{G}_{\hat{V}_i} \circ \mathcal{F}_{\hat{S}_i})(\mathbb{M}_{V_i^I})\Big), \quad (10)$$

其中，$\hat{} S_i = \{\vec{w}_{u,v} \in S_i \mid u \in V_i^I, v \in \hat{} V_i\}$ 是两个层之间的一组快捷边，$X_{L_{low}^V}$ 是在 L_{low} 上经过局部迭代计算后得到的顶点状态，用于向 L_{up}的顶点上传修订消息，即

$$X_{L_{low}^V} = \mathcal{G}_{L_{low}^V}\big(X^0_{L_{low}^V} \cup \bigcup_{k=1}^{\infty}(\mathcal{G}_{V_i} \circ \mathcal{F}_{E_i})^k(\mathbb{M}^0_{V_i})\big). \quad (11)$$

我们通过以下定理来保证正确性。

定理2： 在上层进行迭代计算后，通过快捷边将入口顶点累积的消息分配给内部顶点，所得的内部顶点状态与在更新后的图 $G \oplus \Delta G$上进行迭代计算后得到的状态相同。□

证明概要： 根据方程（11），在L_{up}的顶点上进行局部迭代计算以上传修订消息后，内部顶点之间的影响已相互传递。累积的外部消息$\mathbb{M}_{V_i^I}$包含子图外所有其他顶点的影响，这些影响累积在入口顶点 V_i^I处。将这些外部消息分配给内部顶点，即 $\mathcal{G}_{\hat{V}_i}\big(\mathcal{F}_{\hat{} S_i}(\mathbb{M}_{V_i^I})\big)$，即可将外部影响施加到内部顶点上。对这些外部消息与内部顶点状态$X_{L_{low}^V}$ 进行聚合，其结果等于在整个图上进行迭代计算所得的结果。□

例6： 接续例5，对于已激活的入口顶点 v_5，它通过快捷边向内部顶点分配修订消息。具体而言，$m_{v_5,v_6}=m_{v_5}+\vec{w}_{v_5,v_6}=8$、$m_{v_5,v_7}=m_{v_5}+\vec{w}_{v_5,v_7}=9$ 和 $m_{v_5,v_8}=m_{v_5}+\vec{w}_{v_5,v_8}=9$。最后，$\{v_6,v_7,v_8\}$ 通过消息聚合操作获得收敛状态 $\{x^*_{v_6}=8, x^*_{v_7}=9, x^*_{v_8}=9\}$。□

表I：实验中使用的数据集

图	顶点	边	Size
UK-2005（英国） [24]	39,459,925	936,364,282	16GB
IT-2004（意大利） [25]	41,291,594	1,150,725,436	19GB
SK-2005（斯洛伐克） [26]	50,636,154	1,949,412,601	33GB
新浪微博（世界银行） [27]	58,655,850	261,323,450	3.8GB

六、实验

我们基于 Ingress [16],实现了 Layph。Ingress 是一个自动化增量化框架，配备了不同的记忆策略，以支持以顶点为中心的图计算。本节中，我们评估 Layph，并将其与现有最先进的增量图处理系统进行比较。

A. 实验设置

这些实验使用阿里云ecs.r6.13xlarge实例（52 个 vCPU、384 GB 内存、64 位 Ubuntu 18.04，编译器为 GCC 7.5）。

图工作负载。实验中使用了四种典型的图分析算法，包括单源最短路径（SSSP）、广度优先搜索（BFS）、PageRank（PR）和惩罚命中概率（PHP） [28]。SSSP 和 BFS 可以写成公式（1）所示的形式。我们还使用文献 [9], [10]中的方法，将 PHP 和 PageRank 改写为公式（1）所示的形式。当所有顶点状态不再变化时，前两种算法即视为收敛；当连续两次迭代之间的顶点状态差异小于 $1e^{-6}$时，后两种算法即视为收敛。

数据集与更新。实验中使用了四个真实图（见表I），包括三个网络图 UK-2005（UK） [24],、IT-2004（IT） [25], 和 SK-2005（SK） [26],，以及一个社交网络新浪微博（WB） [27]。我们通过向 G 随机添加新边并从 G中删除已有边，构造了 ΔG 。除非另有说明，添加边和删除边的数量默认均为 5,000。默认情况下， ΔG 指边变化；此外，我们还随机生成一个包含 1,000 个变化顶点（包括 500 个新增顶点和 500 个删除顶点）的 ΔG ，以评估处理顶点更新的性能。

竞争系统。我们将 Layph 与五种最先进的增量图处理系统进行比较：GraphBolt [15],Kick- Starter [14], DZiG [17], Ingress [16], 和 RisGraph [18]。事实上，由于要求每个顶点只能有一个依赖项，KickStarter 和 RisGraph 不支持 PageRank 和 PHP。GraphBolt 和 DZiG 没有提供单源最短路径和广度优先搜索的实现。因此，我们仅在 GraphBolt 和 DZiG（分别在 KickStarter 和 RisGraph）上运行 PageRank 和 PHP（分别运行单源最短路径和广度优先搜索）。所有这些系统均使用 16 个工作线程运行。

B. 总体性能

我们首先比较 Layph 与竞争系统在不同数据集上执行各项工作负载时的响应时间。图 5 展示了归一化结果，其中以 Layph 的响应时间为基准，即，Layph 的运行时间为 1。具体而言，图 5e 展示了响应

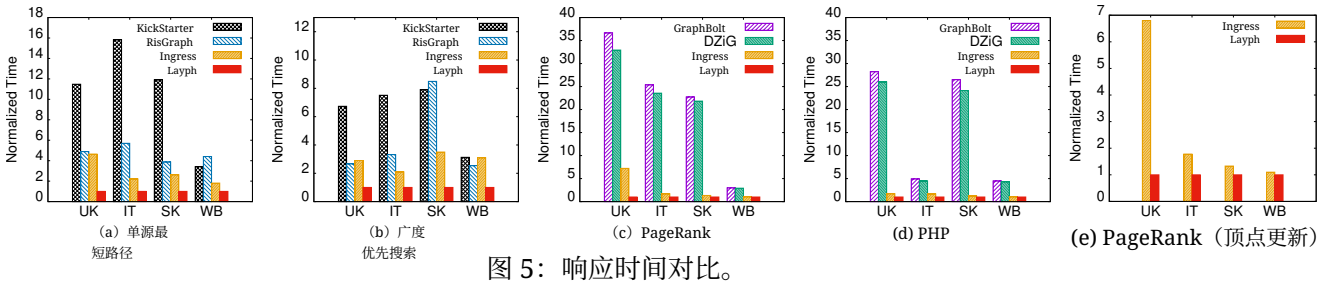


图 5：响应时间对比。

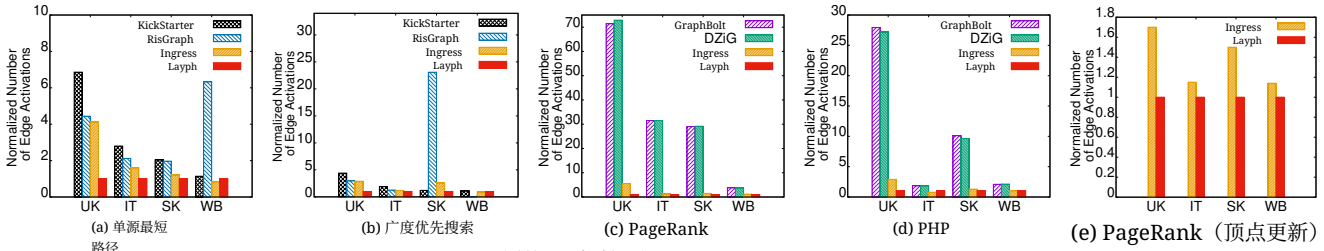


图 6：边激活次数对比。

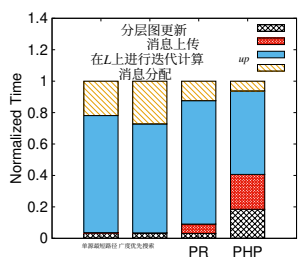
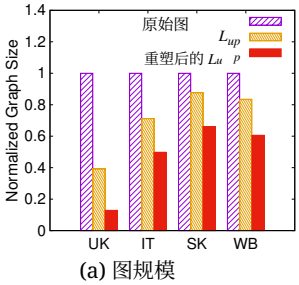


图 7：运行时间分解。



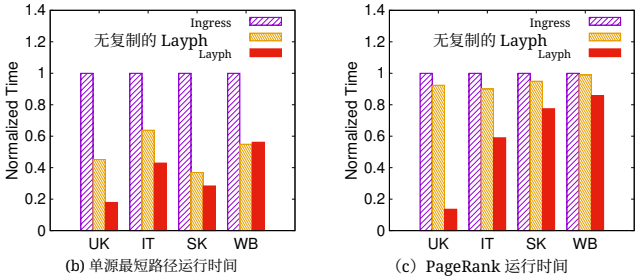
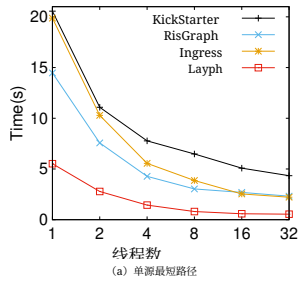


图 8：顶点复制的影响。



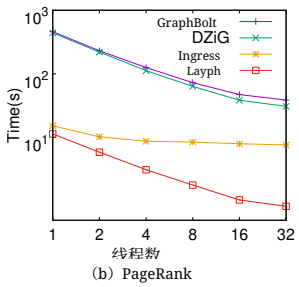
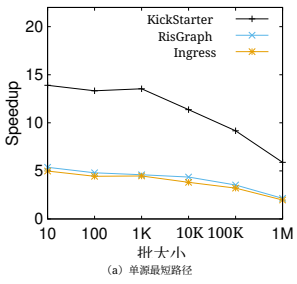


图 9：将线程数从 1 扩展到 32。



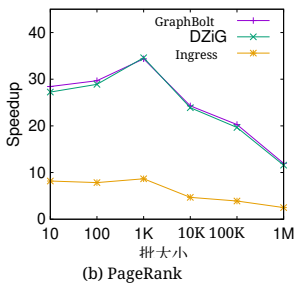


图 10：改变批大小时相对于竞争系统的加速比。

时间：处理顶点更新所需的时间，其余图表展示边更新所需的时间。可以看出，Layph 在处理顶点变化方面的改进，与其在处理边变化方面的改进一致。更新顶点时，其他系统均出现运行时错误，因此我们只比较 Ingress 和 Layph。结果表明，Layph 在大多数情况下始终优于其他系统。具体而言，Layph 相较于 KickStarter 的加速比为 3.13–15.82×（平均为 8.49× ），相较于 RisGraph 为 2.54–8.49×（平均为 4.49× ），相较于 GraphBolt 为 2.99–36.66×（平均为 18.99×），相较于 DZiG 为 2.92–32.93×（平均为 17.53× ），相较于 Ingress 为 1.06–7.22×（平均为 2.54× ）。为解释上述结果，我们还在图 6 中报告了边激活总数。一次边激活代表一次 $\mathcal{F}$ 操作。在大多数图工作负载中，$\mathcal{F}$ 的开销远高于 $\mathcal{G}$ 操作，因为$\mathcal{F}$ 的数量和 $\mathcal{F}$ 的单位开销通常都大于 $\mathcal{G}$。

从图 5 和图 6 可以看出，边激活次数的归一化值与各系统的归一化响应时间呈现相似趋势。

在单源最短路径和广度优先搜索方面，RisGraph 比 Kick- Starter 更快，因为它在增量更新期间支持更多并行性，并允许局部化数据访问。Ingress 和 RisGraph 的性能相当，因为 Ingress 中的记忆化路径引擎采用了类似的思路。Layph 利用分层图，性能优于其他竞争系统。请注意，在 WB 上执行广度优先搜索时，RigGraph 比 Layph 慢，但边激活次数更少。这是因为 RisGraph 能够识别安全更新和不安全更新，从而减少边激活。恰好 WB 上的大多数更新对于广度优先搜索而言都是安全的。然而，识别更新是否安全的额外开销相对较大，因为 WB 非常小。而在单源最短路径方面，与 Ingress 相比，Layph 的响应时间也更短，但边激活次数更多。

这是因为 WB 中存在一些大型稠密子图，需要进行更多快捷更新，从而增加边激活次数。由于 Layph 对快捷更新的并行支持良好，因此这对响应时间的影响很小。

对于 PageRank 和 PHP，DZiG 比 Graph- Bolt 更快，因为 DZiG 具有稀疏性检测机制，并能据此调整增量计算方案。此外，Ingress 比 DZiG 和 GraphBolt 更快。这可以归因于其无记忆化引擎，该引擎比其他引擎效率更高。Layph 构建于 Ingress 之上，并能利用分层图进一步限制迭代计算范围，从而减少激活边的数量，如图 6 所示。我们发现，Layph 在 WB 上的提升较小。原因是 WB 中的子图远大于其他图中的子图，这增加了开销并削弱了收益。我们将在第VI-F节进一步解释这一原因。

C. 运行时间分解

在增量计算过程中，Layph 包含四个阶段：分层图更新、修订消息上传、在上层进行迭代计算，以及消息分配。为研究各阶段耗时，我们在英国运行了四种算法，并记录每个阶段的运行时间。图 7 展示了不同阶段的运行时间占比。可以看出，迭代计算占用了大部分运行时间。消息分配是开销第二大的阶段。除 PHP 外，分层图更新和修订消息上传都非常快。这是因为 PHP 的迭代计算非常快，例如仅需 418 ms，因此这两个阶段所占时间相对较长。结果表明，我们系统中的额外开销，即 分层图的维护，开销很小。基于上述实验分析，在增量图处理中采用分层图是值得的。

D. 变化的线程数

我们将执行线程数从 1 增加到 32，以观察运行时间的缩短情况。我们在英国数据集上运行单源最短路径，并将 Layph 与 KickStarter、RisGraph 和 Ingress 进行比较。结果如图9a所示。可以看到，随着线程数增加，所有系统的运行时间都如预期般持续下降。当线程数超过 8 时，运行时间的下降趋势更加平缓。这是因为所有这些系统都使用原子操作来保证正确性，因此线程增多会导致更多写写冲突，从而损害并行性。与使用 1 个线程时的运行时间相比，Layph 使用 32 个线程时可实现 10.1× 的加速比，高于 KickStarter（4.7× 的加速比）、RisGraph（6.2× 的加速比）和 Ingress（9.0× 的加速比）。我们还在英国数据集上运行 PageRank，并将 Layph 与 GraphBolt、DZiG 和 Ingress 进行比较。结果如图9b所示，其中Y轴采用以10为底的对数刻度。可以看到，GraphBolt、DZiG 和 Layph 的扩展性能均优于 Ingress。原因在于，PageRank 中写写冲突问题比单源最短路径中更严重。在 GraphBolt 和 DZiG 中，每次迭代都需要记录顶点状态，这可以缓解冲突问题，但要付出大量空间开销作为代价。在 Layph 中，快捷更新过程和局部分配过程都包含许多相互独立的局部计算，因此 Layph 更适合并行处理。因此，Layph 能更充分地发挥并行性的优势。

E. 变化的更新量

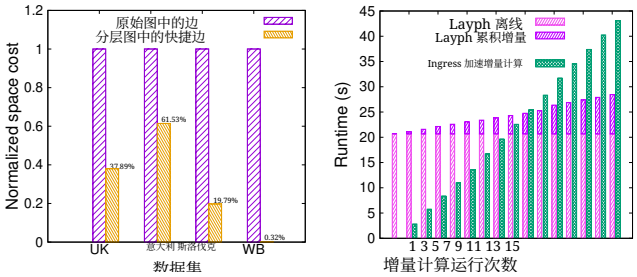
为研究不同更新量下的性能，我们在英国将更新集的大小（即批大小）从 10 调整到 1000 万，并比较 Layph 与竞争系统在运行单源最短路径和 PageRank 时的表现。图10展示了 Layph 相较于竞争系统的加速比。批大小越小，加速比越显著，因为 Layph 利用分层图有效缩小了全局迭代的范围。在 PageRank 中，如果批大小过小，例如10，这些更新的影响可能仅作用于子图内，因此迭代计算仅限于受影响的子图。然而，批大小增大时，加速比的提升就不那么显著。这是因为在我们的系统中，更多更新可能影响更多子图，从而增加快捷边更新开销，并削弱分层图的优势。但是，大批量更新会延长响应时间并失去实时性，因此，对于延迟敏感型应用或在线应用，较小批量的更新更为可取。

F. 顶点复制的影响

为验证第IV-A1节提出的顶点复制的有效性，我们测量了原始图、原始上层和重塑后的上层的规模，如图8a所示。可以看到，采用分层图后，原始图的规模大幅缩小（减少了 12%-60%）；通过顶点复制，原始上层的规模进一步缩小（减少了 34%-87%）。我们还在带有 Ingress 的原始图、使用 Layph 的原始上层（不进行顶点复制）以及使用 Layph 的重塑后上层上运行单源最短路径和 PageRank。运行时间结果分别如图8b和图8c所示。可以看到，大多数运行时间结果与其图规模或上层规模成正比。值得注意的是，在 WB 上使用 Layph 运行单源最短路径时，进行顶点复制后的运行时间比不进行顶点复制更长。通过深入分析图属性，我们发现 WB 中子图的规模很大。进行顶点复制后，一次边更新可能会在多个与该更新边相关的子图上引发多次局部重新计算。因此，如果许多大型子图受到影响，分层图更新中快捷计算的成本就会很高，甚至可能超过其带来的收益。相反，如果受影响子图的规模较小，则不会影响性能，因为快捷计算会非常快。

G. 额外开销分析

为评估额外空间和离线操作对 Layph 的影响，我们首先在图11a中报告 Layph 的额外空间开销。可以看到，分层图带来的额外空间开销分别为原始图的37.89%、61.53%、19.79%、



(a) 空间开销 (b) 离线预处理时间

图 11：额外空间开销和离线预处理时间。

和0.32%，这是可以接受的。随后，在 UK 上执行 SSSP 时，我们在图11b中报告离线预处理时间（Layph 离线）、Layph 的累计增量计算时间（Layph 累计增量）以及 Ingress 的累计增量计算时间（Ingress 累计增量）。结果表明，在进行9次增量计算后，Layph 的运行时间（包括离线时间和累计增量计算时间）开始低于 Ingress。这是因为离线操作只需执行一次，却能在每次增量计算中带来显著的性能提升。

七、相关工作

增量图处理系统。近年来，面向演化图的增量处理受到了广泛关注 [14]–[18], [29]–[41]。Tornado [29] 为定点图计算提供了基于循环的增量化支持。KickStarter [14] 维护依赖树，以记忆收敛状态的关键路径，并执行必要的调整以适应变化。RisGraph [18]在图更新后推导出安全近似结果，并通过迭代计算修正这些结果。GraphBolt [15]通过记忆各次迭代之间的中间状态来跟踪依赖关系，并逐次迭代调整依赖关系，以实现增量计算。i^{2}MapReduce [38], [39] 通过记忆中间 MapReduce 输出，扩展 Hadoop MapReduce 以支持增量迭代图计算。类似地，许多其他工作（例如 DZiG [17] 和 HBSP 模型 [41], ）也会记忆并重用先前的计算，以尽量减少无用的重新执行。Ingress [16] 能够根据算法属性自动选择最佳的记忆化方案。上述系统会将图更新的影响传播到整个图中，从而导致大量顶点和边被激活，最终产生大量计算。

增量图处理硬件加速器。近期提出了许多基于新型硬件来加速动态图处理的解决方案 [37],[42]–[45]。GraSU [42] 提供了首个基于 FPGA、面向动态图的高吞吐量图更新库。它通过利用空间相似性来加速图更新。Jet-Stream [43]将面向图工作负载的基于事件的加速器 [20] 扩展为支持流式更新。它在累积型和单调图算法上均表现良好。 [44]提出了输入感知的软件和硬件解决方案，以提升增量图更新和图处理的性能。TDGraph [45]提出了高效的拓扑驱动增量执行方法，用于加速器设计，以实现更规律的状态传播和更好的数据局部性。

增量图算法。此外，还有许多针对特定算法提出的增量方法，例如，正则路径查询 [46], 强连通分量 [47],子图同构 [48]，k- 核 [49]，图划分 [50], [51] 和三角形计数 [52]。与这些针对特定算法的方法不同，我们的 Layph 框架扩展了 Ingress [16]，，后者能够通过通用方法从批处理对应算法中自动推导出增量算法。它支持一系列具有不同计算模式的增量图算法，即基于遍历的算法（例如单源最短路径和广度优先搜索）以及基于迭代的算法（例如 PageRank 和 PHP）。

基于分区的方法。已有一些基于分区的方法被提出以改进图处理，例如 Blogel[53],Giraph++ [54]、Grace [55], GRAPE [13]。这些方法采用以块为中心（或以子图为中心）的框架来处理图，并尝试减少线程或处理器之间的通信开销（减少子图之间的信息流动）。然而，这些系统是为静态图处理设计的。与这些现有方法不同，Layph 的创新之处在于提出了一种分层图结构，以改进动态图的增量图处理，旨在减少海量消息传播所导致的计算开销。

VIII. 结论

我们提出了 Layph，这是一种通过分层图来加速增量图处理的框架。它将原始图上的全局迭代计算限制为下层上少量相互独立的小规模局部迭代计算（用于更新快捷边并上传消息），以及上层图骨架上的一次全局计算。由于我们的分层图有效限制了迭代计算所涉及的顶点和边的数量，因此 Layph 非常适用于演化图的增量计算。具体而言，只有下层受 ΔG 影响的稠密子图和上层的图骨架会执行迭代计算。Layph 基于我们此前的工作 Ingress 实现，以支持消息驱动的增量计算。实验结果表明，Layph 能显著提升动态图的增量处理性能。

致谢

本研究得到国家自然科学基金（62072082、U2241212、U1811261、62202088、62202301）、国家社会科学基金（21&ZD124）、中央高校基本科研业务费（N2216012、N2216015）、辽宁省重点研发计划（2020JH2/10100037）以及阿里巴巴创新研究（AIR）计划研究基金的资助。

参考文献

[1] J. Tang、T. Wang、J. Wang 和 D. Wei，“基于网络结构特征对博客圈进行高效的社交网络近似分析”，载于第三届社交网络挖掘与分析研讨会论文集，*SNAKDD 2009*，法国巴黎，*2009*年*6*月*28*日。ACM，2009年，第7页。[2] L. Page、S. Brin、R. Motwani 和 T. Winograd，“PageRank 引文排名：为网络建立秩序。”斯坦福大学信息实验室，技术报告，1999年。[3] J. Cho、S. Roy 和 R. Adams，“页面质量：寻找无偏的网页排名”，载于*ACM SIGMOD*国际数据管理会议论文集，美国马里兰州巴尔的摩，*2005*年*6*月*14*日至*16*日。ACM，2005年，第551–562页。[4] Y. Ahn、S. Park、S. Lee 和 S. Lee，“基于异构图的推荐模拟器”，载于第七届*ACM*推荐系统会议，*RecSys '13*，中国香港，*2013*年*10*月*12*日至*16*日。ACM，2013年，第471–472页。[5] B. Berger、R. Singh 和 J. Xu，“生物系统分析的图算法”，载于第十九届*ACM-SIAM*离散算法研讨会论文集，*SODA 2008*，美国加利福尼亚州旧金山，*2008*年*1*月*20*日至*22*日。SIAM，2008年，第142–151页。[6] “维基百科的规模”，2020年，*https://en.wikipedia.org/wiki/Wikipedia:Size of Wikipedia*。[7] G. Malewicz、M. H. Austern、A. J. C. Bik、J. C. Dehnert、I. Horn、N. Leiser 和 G. Czajkowski，“Pregel：一种大规模图处理系统”，载于*ACM SIGMOD*国际数据管理会议论文集，*SIGMOD 2010*，美国印第安纳州印第安纳波利斯，*2010*年*6*月*6*日至*10*日。ACM，2010年，第135–146页。[8] J. E. Gonzalez、Y. Low、H. Gu、D. Bickson 和 C. Guestrin，“Powergraph：在自然图上进行分布式图并行计算”，载于第十届*USENIX*操作系统设计与实现研讨会，*OSDI 2012*，美国加利福尼亚州好莱坞，*2012*年*10*月*8*日至*10*日。USENIX协会，2012年，第17–30页。[9] Q. Wang、Y. Zhang、H. Wang、L. Geng、R. Lee、X. Zhang 和 G. Yu，“自动化递归聚合数据处理的增量与异步求值”，载于*2020*年数据管理国际会议论文集，*SIGMOD 2020*会议，线上会议 *[*美国俄勒冈州波特兰*], 2020*年*6*月*14*日至*19*日。ACM，2020年，第2439–2454页。[10] Y. Zhang、Q. Gao、L. Gao 和 C. Wang，“Maiter：一种用于基于增量的累积迭代计算的异步图处理框架”，*IEEE*并行与分布式系统汇刊，第25卷，第8期，第2091–2100页，2014年。[11] Z. Yanfeng、G. Qixin、G. Lixin 和 W. Cuirong，“Priter：一种用于优先级迭代计算的分布式框架”，载于*ACM*云计算研讨会（与*SOSP 2011*同期举行），*SOCC '11*，葡萄牙卡斯凯什，*2011*年*10*月*26*日至*28*日。ACM，2011年，第13页。[12] X. Zhu、W. Chen、W. Zheng 和 X. Ma，“Gemini：一种以计算为中心的分布式图处理系统”，载于第十二届*USENIX*操作系统设计与实现研讨会，*OSDI 2016*，美国佐治亚州萨凡纳，*2016*年*11*月*2*日至*4*日。USENIX协会，2016年，第301–316页。[13] W. Fan、J. Xu、Y. Wu、W. Yu、J. Jiang、Z. Zheng、B. Zhang、Y. Cao 和 C. Tian，“并行化顺序图计算”，载于*2017*年*ACM*数据管理国际会议论文集，*2017*年 *SIGMOD* 会议，美国伊利诺伊州芝加哥，*2017*年*5*月*14*日至*19*日。ACM，2017年，第495–510页。[14] K. Vora、R. Gupta 和 G. Xu，“Kickstarter：通过截断近似快速、准确地计算流式图”，载于第二十二届编程语言与操作系统体系结构支持国际会议论文集，*ASPLOS 2017*，中国西安，*2017*年*4*月*8*日至*12*日。ACM，2017年，第237–251页。[15] M. Mariappan 和 K. Vora，“Graphbolt：由依赖关系驱动的流式图同步处理”，载于第十四届*EuroSys*会议论文集，*2019*年，*28*，*2019 25*2019，25 1 25 16德国德累斯顿，*3*月 -。ACM，第：–：页。[16] S. Gong、C. Tian、Q. Yin、W. Yu、Y. Zhang、L. Geng、S. Yu、G. Yu 和 J. Zhou，“通过灵活记忆化实现增量图处理自动化”，*VLDB*基金会会刊，第14卷，第9期，第1613–1625页，2021年。[17] M. Mariappan、J. Che 和 K. Vora，“Dzig：流式图的稀疏性感知增量处理”，载于*EuroSys '21*：第十六届欧洲计算机系统会议，线上活动，英国，*2021*年*4*月*26*日至*28*日。ACM，2021年，第83–98页。[18] G. Feng、Z. Ma、D. Li、S. Chen、X. Zhu、W. Han 和 W. Chen，“Risgraph：一种用于演化图的实时流处理系统，以支持每次更新的分析延迟低于毫秒级，吞吐量达每秒数百万次操作”，载于SIGMOD '21：国际数据管理会议，线上活动，中国，2021年6月20日至25日。ACM，2021年，第513–527页。[19] “libgrape-lite”，2020年，https://github.com/alibaba/libgrape-lite。[20] S. Rahman、N. B. Abu-Ghazaleh和R. Gupta，“Graphpulse：用于异步图处理的事件驱动硬件加速器”，载于第53届IEEE/ACM国际微体系结构研讨会（MICRO 2020），希腊雅典，2020年10月17日至21日。IEEE，2020年，第908–921页。[21] V. D. Blondel、J.-L. Guillaume、R. Lambiotte和E. Lefebvre，“大型网络中的社区快速展开”，《统计力学杂志：理论与实验》，第2008卷，第10期，第P10008页，2008年。[22] D. Zhuang、J. M. Chang和M. Li，“DynaMo：通过增量式最大化模块度进行动态社区检测”，《IEEE知识与数据工程汇刊》，第33卷，第5期，第1934–1945页，2021年。[23] M. Seifikar、S. Farzi和M. Barati，“C-Blondel：一种高效的基于Louvain的动态社区检测算法”，《IEEE计算社会系统汇刊》，第7卷，第2期，第308–318页，2020年。[24] “UK-2005”，2005年，https://www.cise.ufl.edu/research/sparse/matrices/LAW / uk-2005.html。[25] “IT-2004”，https://law.di.unimi.it/webdata/it-2004/，2004年。[26] “SK-2005”，https://law.di.unimi.it/webdata/sk-2005/，2005年。[27] R. A. Rossi和N. K. Ahmed，“网络数据存储库：交互式图分析与可视化”，载于第二十九届AAAI人工智能会议论文集，2015年1月25日至30日，美国得克萨斯州奥斯汀。AAAI出版社，2015年，第4292–4293页。[28] Z. Guan、J. Wu、Q. Zhang、A. K. Singh和X. Yan，“评估并排序图中的结构相关性”，载于ACM SIGMOD国际数据管理会议（SIGMOD 2011），希腊雅典，2011年6月12日至16日。ACM，2011年，第937–948页。[29] X. Shi、B. Cui、Y. Shao和Y. Tong，“Tornado：一种针对演变数据进行实时迭代分析的系统”，载于2016年国际数据管理会议论文集（SIGMOD 2016），美国加利福尼亚州旧金山，2016年6月26日至7月1日。ACM，2016年，第417–430页。[30] D. Sengupta、N. Sundaram、X. Zhu、T. L. Willke、J. S. Young、M. Wolf和K. Schwan，“Graphin：一种在线高性能增量图处理框架”，载于Euro-Par 2016：并行处理——第22届并行与分布式计算国际会议，法国格勒诺布尔，2016年8月24日至26日，论文集，丛书：计算机科学讲义，第9833卷。施普林格，2016年，第319–333页。[31] S. Ko、T. Lee、K. Hong、W. Lee、I. Seo、J. Seo和W. Han，“iturbograph：扩展并自动化增量图分析”，载于SIGMOD '21：国际数据管理会议，线上活动，中国，2021年6月20日至25日。ACM，2021年，第977–990页。[32] X. Jiang、C. Xu、X. Yin、Z. Zhao和R. Gupta，“Tripoline：通过图三角不等式实现广义增量图处理”，载于EuroSys '21：第十六届欧洲计算机系统会议，线上活动，英国，2021年4月26日至28日。ACM，2021年，第17–32页。[33] T. A. K. Zakian、L. A. R. Capelli和Z. Hu，“以顶点为中心的程序的增量化”，载于2019年IEEE国际并行与分布式处理研讨会（IPDPS 2019），巴西里约热内卢，2019年5月20日至24日。IEEE，2019年，第1019–1029页。[34] D. G. Murray、F. McSherry、R. Isaacs、M. Isard、P. Barham和M. Abadi，“Naiad：一个及时数据流系统”，载于ACM SIGOPS第二十四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。ACM，2013年，第439–455页。[35] F. McSherry、D. G. Murray、R. Isaacs和M. Isard，“微分数据流”，载于第六届创新数据系统研究双年会议（CIDR 2013），美国加利福尼亚州阿西洛马，2013年1月6日至9日，在线论文集。www.cidrdb.org，2013年。[36] P. Vaziri和K. Vora，“控制有状态流式图处理的内存占用”，载于2021年USENIX年度技术会议（USENIX ATC 2021），2021年7月14日至16日，I. Calciu和G. Kuenning编。USENIX协会，2021年，第269–283页。[37] D. Chen、C. Gui、Y. Zhang、H. Jin、L. Zheng、Y. Huang和X. Liao，“Graphfly：通过依赖流实现高效的异步流式图处理”，载于2022年SC22：高性能计算、网络、存储与分析国际会议，2022年，第632–645页。[38] Y. Zhang、S. Chen、Q. Wang和G. Yu，“i2mapreduce：用于演变大数据挖掘的增量MapReduce”，载于第32届IEEE国际数据工程会议（ICDE 2016），芬兰赫尔辛基，2016年5月16日至20日。IEEE计算机学会，2016年，第1482–1483页。

[39] Y. Zhang和S. Chen，“i2mapreduce：增量迭代MapReduce”，载于第二届云智能国际研讨会（与*VLDB 2013*同期举行 *26, 2013 13,* 2013, 3 1 3 4），*Cloud-I ' Riva del Garda*，意大利特伦托，*8*月。ACM，第：–：页。[40] Z. Cai、D. Logothetis和G. Siganos，“促进实时图挖掘”，载于第四届云数据管理国际研讨会论文集，*CloudDB 2012*，美国夏威夷州毛伊岛，*2012*年*10*月*29*日。ACM，2012年，第1–8页。[41] C. Wickramaarachchi、C. Chelmis和V. K. Prasanna，“促进大规模动态图上的快速增量计算”，载于*2015*年*IEEE*国际并行与分布式处理研讨会工作坊，*IPDPS 2015*，印度海得拉巴，*2015*年*5*月*25*日至*29*日。IEEE计算机学会，2015年，第1166–1171页。[42] Q. Wang、L. Zheng、Y. Huang、P. Yao、C. Gui、X. Liao、H. Jin、W. Jiang和F. Mao，“GraSU：一种用于基于FPGA的动态图处理的快速图更新库”，载于*FPGA '21*：*2021*年*ACM/SIGDA*现场可编程门阵列国际研讨会，线上活动，美国，*2021*年*2*月*28*日至*3*月*2*日。ACM，2021年，第149–159页。[43] S. Rahman、M. Afarin、N. B. Abu-Ghazaleh和R. Gupta，“JetStream：利用事件驱动硬件加速器分析流数据上的图”，载于*MICRO '21*：第*54*届*IEEE/ACM*微体系结构国际年会，线上活动，希腊，*2021*年*10*月*18*日至*22*日。ACM，2021年，第1091–1105页。[44] A. Basak、Z. Qu、J. Lin、A. R. Alameldeen、Z. Chishti、Y. Ding和Y. Xie，“利用输入知识提升流式图处理性能”，载于*MICRO '21*：第*54*届*IEEE/ACM*微体系结构国际年会，线上活动，希腊，*2021*年*10*月*18*日至*22*日2021, 1036 1050-。ACM，第–页。[45] J. Zhao、Y. Yang、Y. Zhang、X. Liao、L. Gu、L. He、B. He、H. Jin、H. Liu、X. Jiang和H. Yu，“TDGraph：一种用于高性能流式图处理的拓扑驱动加速器”，载于*ISCA '22*：第*49*届计算机体系结构国际年会，美国纽约州纽约，*2022*年*6*月*18*日至*22*日。ACM，2022年，第116–129页。[46] W. Fan、C. Hu和C. Tian，“增量图计算：可行与可撤销”，载于*2017*年*ACM*国际会议论文集——数据管理国际会议，*2017*年 *SIGMOD* 会议，美国伊利诺伊州芝加哥，*2017*年*5*月*14*日至*19*日。ACM，2017年，第155–169页。[47] J. Holm、K. de Lichtenberg 和 M. Thorup，“用于连通性、最小生成树、2-边连通性和双连通性的多对数时间确定性全动态算法”，《美国计算机协会学报》，第48卷，第4期，第723–760页，2001年。[48] K. Kim、I. Seo、W. Han、J. Lee、S. Hong、H. Chafi、H. Shin 和 G. Jeong，“Turboflux：一种用于流式图数据的快速连续子图匹配系统”，载于*2018*年数据管理国际会议论文集，*2018*年 *SIGMOD* 会议，美国得克萨斯州休斯敦，*2018*年*6*月*10*日至*15*日。ACM，2018年，第411–426页。[49] R. Li、J. X. Yu 和 R. Mao，“大型动态图中的高效核维护”，*IEEE*知识与数据工程汇刊，第26卷，第10期，第2453–2465页，2014年。[50] W. Fan、M. Liu、C. Tian、R. Xu 和 J. Zhou，“图划分算法的增量化”，*VLDB*学报，第13卷，第8期，第1261–1274页，2020年。[51] W. Fan、C. Hu、M. Liu、P. Lu、Q. Yin 和 J. Zhou，“并行图计算的动态扩缩”，*VLDB*学报，第12卷，第8期，第877–890页，2019年。[52] A. McGregor、S. Vorotnikova 和 H. T. Vu，“数据流中三角形计数的更优算法”，载于第*35*届 *ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集，*2016*年 *PODS*，美国加利福尼亚州旧金山，*2016*年*6*月*26*日至*7*月*1*日。ACM，2016年，第401–411页。[53] D. Yan、J. Cheng、Y. Lu 和 W. Ng，“Blogel：一种用于真实世界图分布式计算的以块为中心的框架”，*VLDB*学报，第7卷，第14期，第1981–1992页，2014年。[54] Y. Tian、A. Balmin、S. A. Corsten、S. Tatikonda 和 J. McPherson，“从‘像顶点一样思考’到‘像图一样思考’”，*VLDB*学报，第7卷，第3期，第193–204页，2013年。[55] W. Xie、G. Wang、D. Bindel、A. J. Demers 和 J. Gehrke，“ 利用块更新进行快速迭代图计算 ”，*VLDB* 学报，第 6 卷，第 14期，第2014–2025页，2013年。