面向批量操作的无锁工作窃取算法

Raja Sai NandhanYadav Kataru爱荷华州立大学计算机科学系nandgate@iastate.edu

Danial Davarnia普渡大学工业工程学院ddavarn@purdue.edu

Ali Jannesari爱荷华州立大学计算机科学系jannesar@iastate.edu

arXiv:2603.05766v1 [cs.DC] 5 Mar 2026

摘要——**工作窃取是一种广泛应用于平衡不规则并行工作负载的技术，大多数现代运行时系统采用无锁工作窃取双端队列来减少竞争并提高可扩展性。然而，现有算法是为通用并行运行时设计的，在专用场景中往往会产生不必要的开销。本文提出一种新的无锁工作窃取队列，专为基于决策图的混合整数规划优化求解器并行化所采用的主从工作框架而设计。该设计支持原生批量操作、容量无界，并假设最多有一个所有者和一个并发窃取者，从而无需进行繁重的同步。**

我们非正式地说明了，在这种受限并发模型下，我们的队列是线性一致且无锁的。基准测试表明，我们的实现实现了恒定延迟的推入性能，即使批大小增加，性能仍保持稳定；相比之下，来自 C++Taskflow 的现有队列的延迟会随着批大小的增加而急剧增长。所有实现的弹出操作性能相当，而我们的窃取操作在不同窃取比例下保持近乎平坦的延迟。我们还研究了一种优化的窃取变体，在实际应用中可将延迟降低至原来的三分之一。最后，基于大图探索的伪工作负载证实，所有实现都呈线性扩展。然而，我们认为，对于节点处理时间不规则的求解器工作负载，我们的算法优势将得到进一步放大。

索引术语——**工作窃取队列、无锁算法、并发、混合整数规划、决策图**

I. 引言

并行计算是一种计算范式，通过同时执行多个任务，比传统的顺序执行更快地完成一组任务。底层机器通常在芯片上包含多个核心，每个核心并行运行不同的任务。为了高效利用 CPU资源，处理器采用不同的负载均衡策略，使各核心尽可能均衡地保持繁忙。当任务在执行过程中生成新的子任务时，这些子任务通常会被调度到创建它们的同一核心上，从而减少任务迁移，并将上下文切换的开销降至最低。

提出了两种负载均衡方案：工作共享和工作窃取 [1]。在工作共享中，每当创建新任务时，核心都会主动卸载部分工作负载

本项目部分经费由美国国家科学基金会资助，资助编号为 CMMI-2547883。

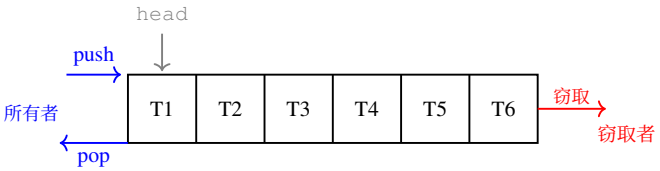

图 1：经典工作窃取双端队列模型：所有者线程在队列顶部推入和弹出任务，而空闲线程则从底部窃取任务。头指针用于跟踪双端队列的顶部。

。当某个核心生成新任务/线程时，调度器会尝试将其中一些任务转移到其他处理器，尤其是那些看起来利用率较低的处理器。其目标是迅速将工作量分散到整个系统中，确保空闲核心尽快获得新任务。然而，频繁迁移任务会带来开销，因为即使其他核心已处于较高负载状态，仍会进行任务分配。这可能导致迁移成本增加、竞争加剧，并降低缓存的利用效率。

相比之下，在工作窃取中，平衡工作负载的责任主要由空闲或利用不足的核心承担。空闲核心不会等待新任务被推入，而是尝试从繁忙核心的队列中窃取任务。这样可以减少任务迁移的开销，因为所有核心都在工作时无需移动任务。因此，工作窃取通常能降低调度开销，并更好地利用缓存。基于这个原因，许多现代并行编程框架（例如 Cilk [2],Intel TBB [3], Java Fork/Join [4]，和 Go 运行时[5]）都采用工作窃取作为主要调度策略。

工作窃取通常涉及为每个核心维护一个私有的双端队列（双端队列）。双端队列是一种支持从两端弹出节点的数据结构。所有者线程从一端（通常是前端）压入和弹出节点，其他线程可以从另一端窃取（如图1所示） [1], [6]。这在支持高效工作分配的同时，最大限度地减少了所有者与窃取者操作之间的竞争。然而，尽管在通用并行运行时中行之有效，现有的工作窃取算法往往做出一些假设，这些假设在具有不同工作负载特征和调度策略的专用场景中可能引入不必要的开销。

一种专门的情形出现在求解组合网-

络问题（CNP）时，这类问题是对某些网络元素施加组合约束的网络优化模型。CNP在从交通运输和供应链管理到物流的各类应用中发挥着核心作用。一个突出的例子是不可分流问题（UFP），这是一种CNP，其中某些节点处的入弧流与出弧流不能被拆分或合并 [7]。具体而言，此类节点上的每一入流都必须与单一出流相匹配。这种结构出现在铁路运输、信息基础设施和云间电信设计等领域 [8]。

在大多数应用中，底层 CNP 的规模很大，由于其中组合约束的复杂性，求解极其困难。随着集群计算和多核计算的快速发展，大规模 CNP 的求解日益依赖于可扩展的并行化求解方法，以便在处理器之间高效分配计算任务。遗憾的是，传统方法（如混合整数规划（MIP）和基于分解的方法）并不适合并行化。为此，近期研究引入了基于决策图（DD）的技术；决策图是对优化问题数据结构 [9], [10]的紧凑图形表示。

A. 决策图概述

形式上，DD 是一个有向无环图 $D=(V,A,\ell)$，包含节点集 V、弧集 A和弧标签函数 ℓ。节点被划分为 $n+2$ 层 $L_1,\dots,L_{n+2}$，其中 n 是决策变量的数量。第一层和最后一层各包含一个节点，即根节点 $r\in L_1$ 和终端节点 $t\in L_{n+2}$。弧被划分为 n 个层级，每条弧$a\in A$ 从 L_i 中的一个节点指向 L_{i+1} 中的一个节点，并带有标签$\ell(a)\in D(x_i)$，表示在i层对变量 x_i 的赋值。因此，任意一条从根节点到终端节点的路径 $p=(a^{(1)},\dots,a^{(n)})$ 都编码了一个针对 $j=1,\dots,n$的可行赋值 $x_j=\ell(a^{(j)})$。

以下面的背包问题为例。

$$\begin{aligned}\max\quad & 8x_1+5x_2+7x_3+6x_4\\ \text{s.t.}\quad & 3x_1+2x_2+4x_3+6x_4\le 7\\ & x_i\in\{0,1\}\end{aligned}\tag{1}$$

图 2 展示了问题 (1) 的一个决策图。每一层对应一个变量 x_i。节点内部的数字表示状态（剩余容量），而节点左侧的数字表示沿路径的累积目标值，该值通过将目标函数系数视为通向相应节点的弧的权重计算得到。最优解通过从 r 到 t的最长路径的长度计算得出。

过去十年间，DD 已应用于离散优化，可用于表示整数规划的可行域。其主要优势在于结构化分解：树的每一层都对应一个决策变量，因此可以存储并重复利用部分解。然而，构建精确DD 将会

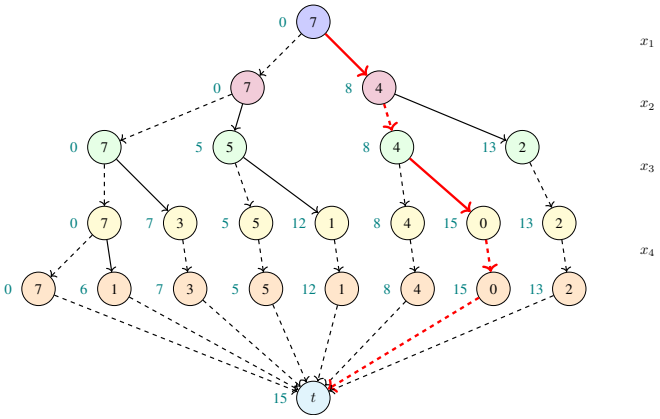

图 2：问题 (1) 的精确决策图。虚线弧和实线弧分别对应决策 0 和 1。最长路径为 15（以红色突出显示），其路径为从根节点r 到终端节点t的红色高亮路径。

导致时间和内存消耗呈指数级增长。因此，在分支定界框架中，通常使用松弛和受限DD，在保持宽度可控的同时提供对偶界和原始界。受限DD（见图 3）通过仅选取大小为最大宽度的部分节点并丢弃其余节点来构建。受限 DD 中的解表示精确DD 原始解集的一个子集。相较之下，松弛DD（见图 4）通过选取大小为最大宽度的部分节点，并将它们合并为一个具有松弛状态值的单一节点来构建，该节点继承所有连接到这些合并节点的弧。这一节点合并过程确保原始解集中的可行解由所得 DD 中路径的一个子集表示，并形成原始解集的超集。

受限决策图为最优解提供原始界（最大化问题中的下界），而松弛决策图提供对偶界（最大化问题中的上界）。通过分支定界框架对受限决策图和松弛决策图进行迭代细化，可将原始–对偶间隙缩小至零。利用这一结构，基于决策图的方法能够实现更高效的计算，并加速分支定界过程。最近已有研究表明，基于决策图的技术可以扩展到混合整数情形，其中优化问题同时涉及离散变量和连续变量，从而显著拓宽能够从该技术中受益的现实问题类别 [11]–[13]。实证研究进一步表明，在包括交通运输、能源系统和统计学习中的稀疏回归在内的广泛应用中，这些方法的性能可以优于现有的优化求解器 [7], [8], [14]–[16]。

然而，即使是这些方法，在面对真实问题实例的规模时也会力不从心。令人鼓舞的是，得益于其特殊的

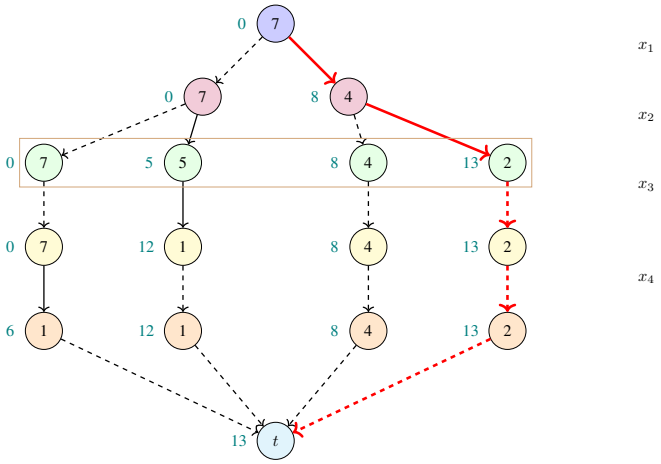

图 3：背包 问题 (1) 的受限决策图。虚线弧和实线弧分别对应决策变量取 0 和 1 的决策。最长 路径为 13（以红色高亮），通过从根节点 r 到终端节点 t 遍历红色高亮路径得到。

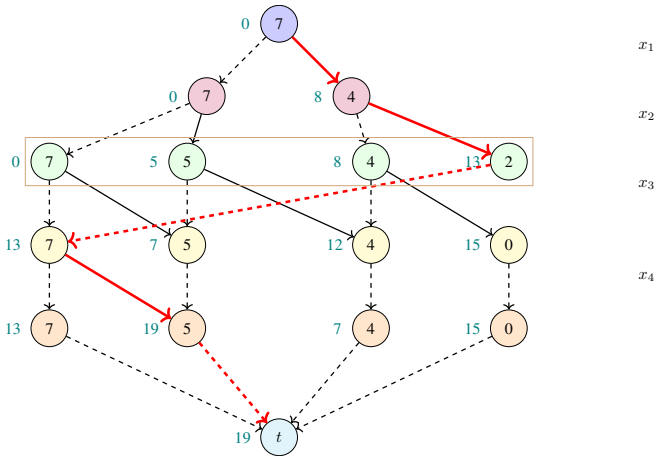

图 4：背包 问题 (1) 的松弛决策图。虚线弧和实线弧分别对应决策变量取 0 和 1 的决策。最长 路径为 19（以红色高亮），通过从根节点 r 到终端节点 t 遍历红色高亮路径得到。

基于图的结构 [17]，决策图天然适合并行化。这一特性可用于提高求解速度，使大规模问题变得可处理。在分支定界过程中，并行化尤其有效：此时，决策图节点会被递归探索，以计算界；通过结合从混合整数情形下适当分解中得到的割平面，还可以进一步强化这些界 [14]。由于搜索需要处理大量节点，且每个节点都可能产生更多节点，分支定界树会迅速增长，直到原始界与对偶界收敛（从而得到最优解）。为应对可扩展性方面的挑战并缩短求解时间，我们的研究聚焦于并行化基于决策图的求解器所需的一个关键组件，该求解器适用于多核环境，目标是在多个处理器之间高效分配节点探索。

基于决策图的求解器采用主从架构，其中每个工作线程从其私有队列中取出一个节点并对其进行探索 [18]。在此过程中，工作线程可能会生成额外的节点并将其推入队列。当某个工作线程的队列变空时，主线程会介入，从繁忙的工作线程窃取节点，并重新分配给空闲的工作线程。此外，主线程维护有关工作线程状态的簿记信息，并动态调整分配策略以保持均衡利用。

B. 主要贡献

我们的主要贡献如下。

- 我们提出了一种新型无锁工作窃取队列，专为主从架构设计。该设计支持原生批量操作，无需调整大小即可实现无界队列增长，并采用简化的并发模型，仅包含一个窃取者。
- 我们通过为每项操作确定线性化点，并证明该算法在保障无锁进展的同时维持安全性，给出了正确性的概念性证明概要。
- 我们展示了大量基准测试和一项伪工作负载研究，证明所提出的队列在批量推送和批量窃取方面均能实现恒定延迟性能。此外，我们还展示了一项优化，可在常见情形下显著降低窃取延迟。

本文其余部分安排如下。第二节概述了塑造我们设计的需求。第三节介绍所提出的算法，第三-B 节则给出正确性论证。第四节报告实验评估结果。第五节回顾了工作窃取队列的相关研究。最后，第六节总结全文，并讨论其局限性及未来研究方向。

II. 动机

A. 需求

在为主从工作框架中的 DD-BD 求解器 [8] 并行版本设计专用的工作窃取算法时，我们发现现有算法未能充分满足若干需求。这些需求源于求解器特有的工作负载特征，以及主进程所执行的调度策略。以下概述了主要需求：

首先，队列应支持批量操作。在我们的场景中，节点生成通常是批量进行的，一次往往会生成一百多个节点。同样，当主线程尝试通过窃取来获取节点时，最好一次操作就获取队列中一定比例的可用节点，而不是逐个获取。支持批量推送和批量窃取操作可以缩短工作线程的等待时间，因为一次函数调用就能传输多个任务，同时也能减少系统中的窃取和推入尝试总次数。虽然可以通过包装重复的单节点

操作来模拟批量操作，但这种方法会引入不必要的开销。因此，原生支持批量操作是一项基本要求。

其次，队列必须是无界的，以容纳会生成大量节点的工作负载。开始时，队列可能会迅速增长，随后随着节点被逐一探索而逐渐缩小。在这种情况下，固定大小的数组实现并不适用，因为数组容量无法在编译时预先确定。同样，动态调整大小的数组也不理想，因为扩展或收缩时需要逐个复制元素，代价高昂。因此，队列必须能够自然地无限增长，但仍受内存限制。

第三，在任何给定时刻，更新队列的并发线程最多只有两个：所有者和单一窃取者。并发窃取者的数量限制为一个，因为主线程充当全局负载均衡器，独自负责根据调度策略窃取节点并将其分配给空闲工作线程。这一约束简化了求解器的同步要求和停止准则，并有助于实现更高效的设计 [19]。

第四，与现代工作窃取设计一致，我们力求避免使用锁。鉴于更新队列的并发线程最多只有两个，我们不需要基于锁的同步所提供的严格可串行化操作保证。此外，避免使用互斥锁可以消除相关的同步开销，而这种开销在细粒度并行环境中尤其高昂。相反，可以通过原子操作等非阻塞同步原语来确保进度。

这些要求共同构成了我们新算法设计的基础，我们将在下一节中介绍该算法。

B. 主从模型

采用单一指定窃取者的主从架构，这一选择自然源于我们当前正在并行化的 DD-BD 求解器的特性。在节点探索过程中，工作线程在生成子节点的同时还会生成割平面，这些割平面是收紧可行域并能够对不可行解和次优解进行积极剪枝的约束。当这些割平面应用于来自搜索空间同一区域或密切相关区域的节点时最为有效，因为此类节点具有相似的结构属性[20]–[22]。

为利用这种局部性，求解器维护搜索过程的集中式视图。主线程负责全局记录，跟踪每个工作线程当前正在探索哪些子空间、监控负载统计，并协调自适应负载均衡决策。这项职责属于调度策略，而非队列抽象本身，因为它依赖于求解器特有的运行时信息，例如子空间亲和性、剪枝效果和工作负载演变。工作窃取队列提供了高效转移任务的机制，而主线程则决定何时、从哪个工作线程以及向哪个工作线程重新分配工作量。

出现负载不均衡时，主线程会有选择地从繁忙的工作线程窃取节点批次，并将其重新分配给空闲的工作线程。重要的是，主线程能够识别出拓扑上与某个工作线程当前子空间最接近的节点子集，并据此进行分配，从而保持局部性，并最大限度地发挥先前生成的割平面的效果。允许多个并发窃取者会破坏这一策略，因为这会导致工作线程队列发生竞争，并使任务分配随机化，进而削弱对求解器性能至关重要的局部性优势。

集中式主线程也便于求解器的未来扩展。具体而言，我们计划使用 MPI 支持分布式执行，其中由单一协调器进行完整的簿记，可简化跨机器的全局调度决策和数据移动。这些扩展属于正在进行的工作，但它们进一步促使我们选择由单一主线程负责窃取和重新分配的设计。

主线程的主要职责是协调负载均衡，并掌握搜索的全局情况。尽管单窃取者设计看起来可能有所限制，但这体现了在调度灵活性与全局协调之间经过权衡后的选择。通过集中执行窃取操作，主线程可以根据整体搜索状态做出明智的工作量重新分配决策。重要的是，主线程不会永久占用一个专用核心；当所有工作线程都在忙碌时，它会参与节点探索，并仅在工作线程空闲时发起窃取。

主线程还会有意等到工作线程的本地队列几乎耗尽后，才重新分配工作量。单个节点可能级联生成数百个新节点，尤其是当这些节点对应有希望的解或最优解时。过早重新分配可能会导致刚分配出去的工作量在节点扩展后很快又需要被窃取回来，从而增加不必要的窃取操作，并破坏局部性。这种设计确保只有在重新分配很可能稳定且有益时才会进行。

主线程的这种行为确保它尽可能参与计算工作，并且只在出现全局负载不平衡时才介入。

III. 算法

双端队列采用单向链表实现，并维护两个原子变量：大小，表示队列当前的元素数量；以及 头指针，指向队列当前的头部。每次 API 操作都会更新这两个值，以确保并发访问下的正确性。每个节点都会填充额外的内存，以防止伪共享。尽管这种数据结构被称为双端队列，但所有遍历都从头节点开始，并在遇到标记链表末尾的空指针时结束。双端队列的类如清单 1 所示。

A. API 操作

推入：（清单 2）：给定一批节点组成的链表，此函数会以原子方式将新节点推入队列头部。它将传入链表的“尾部”连接到

```
1 class lf_node { 2 public: 3 uint8_t padding [CACHE_LI
NE_SIZE]; 4 lf_node *next; 5 }; 6 7 typedef tuple<lf_node*, lf_
node*, size_t>llist; 8 9 class lf_queue { 10 private: 11
std::atomic<size_t>size = 0; 12 std::atomic<lf_node *>head =
nullptr; 13 public: 14 llist steal(double proportion); 15 lf_node
*pop(); 16 void push(const llist&nodes); 17 };
```

清单 1：我们的队列类及其 API 操作的简化伪代码

```
1 void推入(const llist &节点) {2 lf_节点 *start = 节点.start,
*end = 节点.end;3 大小_tn =节点.n;4 end->next =头
指针.load(RELAXED);5 头指针.store(start, 释放序);6 大小.
fetch_add(n, 获取序_释放序);7 }
```

清单 2：所有者调用的推送操作，用于将给定的一批节点插入队列头部

当前队列头部，然后以原子方式更新队列的头指针，使其指向新链表的起始节点。最后，根据添加的节点数量，以原子方式增加队列大小。第三-B 节对内存顺序语义进行了深入分析。

弹出：（清单 3），由所有者调用，用于提取并移除队列的顶端元素（头指针），并以原子方式更新头指针和大小。当队列为空时，此操作返回空指针。

窃取：（清单 4）：窃取者调用此函数，从队列尾部拆离并获取一个连续的节点子链表。给定一个比例参数，函数会确定一个切分点（如图 5 所示），使操作完成后队列中大约保留指定比例的节点，并将剩余后缀移除并作为链表段返回。操作首先读取队列的当前大小并计算

```
1 lf_节点 *弹出() {2 lf_节点 *rv =头指针.
load(RELAXED);3 如果(!rv)返回NULL;4
头指针.store(rv->next,RELAXED);5 大小.
fetch_sub(1, 获取序_释放序);6 rv->next =
NULL;7 返回rv;8 }
```

清单 3：所有者调用弹出操作以移除并返回队列头部。

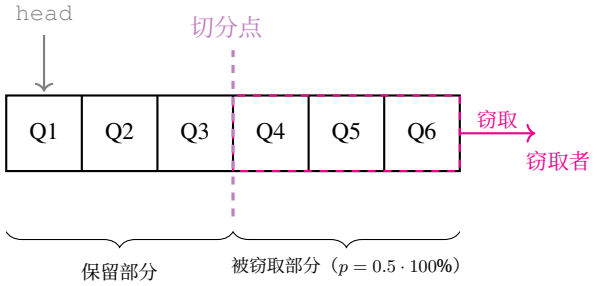

图 5：批量窃取：窃取者通过一次操作从尾部（右侧）移除一定比例 p（虚线区域）。队列在切分点 处断开（虚线），此后后缀成为被窃取的子链表。图中示例为$p = 50$ %。

从头指针开始需要跳过的节点数，以到达切分点。函数遍历链表直至切分点，执行一致性检查以确保队列没有显著变空，记录next指针，并将切断节点的next指针设为空值，从而断开链表。随后遍历拆离的链表段，以确定其尾部并统计被窃取元素的数量，然后将大小 计数器原子递减相应数量。如果队列太小，或一致性检查发现队列正在快速变空，则中止窃取尝试，不返回任何节点。

B. 分析

推入函数仅由所有者线程调用。它首先对头指针执行宽松 原子加载。由于头指针始终只由所有者修改，宽松加载就足够了：它保证能观察到同一线程在先前的推入或弹出操作中写入的最新值[23] [1]。随后对头指针执行的存储采用释放序语义。这确保程序顺序中所有先前的内存指令都在更新头指针之前变得可见，从而防止更早的指令被重排到该存储之后。

大小 计数器的更新以获取-释放序 语义原子执行 [2]。这种选择确保了双向排序：先前的指令不能移到更新之后，后续的指令也不能移到更新之前。因此，任何其他线程若以获取序语义加载大小并观察到此更新，就保证能够看到头指针对应的新值。实际上，获取-释放序 操作充当内存屏障，确保对头指针 和大小的修改保持一致。

窃取操作的正确性源于队列的结构不变量以及受限的并发模型。最多有两个线程会与队列交互

[1]没有其他线程会更新头指针，因此不存在观察到过期值或不一致值的风险。宽松 加载只是复用最近一次写入。[2]一条获取序-释放序 读-改-写指令可同时防止向前和向后的重排序：先前的指令不能越过该更新，后续指令也不能移到它之前。

```
1 llist steal(double proportion) { 2 // 此操作完成后要保留的节点百分比。 3 proportion = 1- proportion; 4 size_t sz = size.load(获取序); 5 if (sz < _queue_limit_) return {NULL, NULL, 0}; 6 7 size_t n_skip =sz * proportion; 8 size_t rem =sz - n_skip; // 要从尾部弹出的节点数。 9 size_t k = n_skip; 10 11 lf_node *start =head.load(获取序); 12 while (n_skip && start) { 13 start =start->next; 14 n_skip--; 15 } 16 17 if (n_skip ||!start) 18 return {NULL, NULL, 0}; // 节点数量不足。 19 20 size_t ssz =size.load( 获取序 ); 21 if (ssz <=(sz - (k>>1))) 22 // 节点弹出得太快，中止操作。 23 return {NULL, NULL, 0}; 24 25 lf_node *begin =start->next; 26 start->next =nullptr; 27 size.fetch_add(0, 释放序); 28 29 // 遍历到链表尾部并更新大小。 30 lf_node *end = begin; 31 size_t count = 0; 32 while (end) { 33 count++; 34 if (!end->next) break; 35 end =end->next; 36 } 37 size.fetch_sub(count); 38 return {begin, end,count}; 39 }
```

图 4：窃取者调用弹出操作，从队列尾部窃取指定比例的节点。若成功，则返回一批以链表形式组织的节点。

并发执行：所有者和单一窃取者。队列是一个以头指针为根节点、以空值标记末端的单向链表。所有者通过在头指针处前置节点来执行推入操作，并通过推进头指针来弹出节点，但绝不会改写内部的next 指针。窃取者从头指针的快照（以获取序语义读取）开始遍历链表，找到切割位置，然后在切分点处断开链表（第 26 行）。这次单独的写入定义了线性化点：此后，前缀仍可从头指针访问，而后缀则返回给窃取者。由于所有者不会重新连接内部链接，且只会在前缀处前置节点或缩短前缀，因此不会有后缀节点被重复或丢失。

代码包含多项一致性检查，以便在遍历期间队列变化过快时中止操作。第 20–21 行（清单 4）会重新加载当前大小，并验证至少还剩下最初观察到的节点的一半，以检测队列是否正被快速清空。这些检查可防止窃取者切入已被所有者几乎清空的队列。使用释放序语义执行的存储操作（第 27 行）可确保对next 设为空值 的更新（第 26 行）得到正确发布，使所有者后续执行的推入和弹出操作能够一致地观察到这次切断。结合对大小的原子递减，这可保证操作返回后，不再有被窃取的节点可从头指针访问。因此，每个节点要么保留在前缀中，要么作为被窃取的后缀返回，同时相应地更新大小。该操作是无锁的，因为窃取者要么通过一次切断成功并更新大小，要么在条件不满足时立即中止，从而确保始终至少有一个线程能够取得进展。

我们算法中的操作是线性一致的。每项操作都存在一个明确定义的线性化点，位于其调用与响应之间。推入在线性化点处写入新头指针时完成，弹出在线性化点处将头指针更新为其后继节点时完成，而窃取在线性化点处写入、通过设置start->next=空值来交付链表时完成。这些线性化点确保保留实时顺序：在窃取开始前完成的推入和弹出操作，可由窃取者通过获取式加载看到；而在后续操作之前完成的窃取，则会因释放语义而得到一致观察。从外部来看，操作序列与一个有效的双端队列顺序执行无法区分：所有者在一端插入和移除任务，而窃取者从另一端移除任务。因此，该实现无需使用锁即可满足线性化。

IV. 评估

我们通过将每项 API 操作与来自C++ Taskflow 库的另外两种工作窃取算法（无界队列和有界队列）进行基准测试，评估了我们的算法（LF 队列）的性能。该基准测试分别测量每项 API 操作的延迟：推入、弹出和窃取。每次基准测试迭代后都会重置队列状态，以确保各次迭代之间的一致性。硬件和软件配置如表 I 所示。

组件	**规格**
处理器型号	Intel Xeon Platinum 8358
基准时钟频率	2.60 GHz
插槽	2
每插槽核心数	32
核心总数	64
硬件线程	128
每个核心的一级缓存	48 KB
编译器	GCC 15
语言标准	C++20
编译器标志	-O3 -march=native

表 I：我们实验中使用的硬件和软件配置。

图6展示了推送操作的延迟随批大小（1、128、512、1024）的变化情况。对于较小的批量推入（1—128个节点），三种实现的性能相近。然而，

随着批大小增加，性能出现分化。在 TF BD 队列和 TF UB 队列中，延迟随着推入节点数量的增加而急剧上升；由于反复执行逐节点操作并产生调整大小开销，推入1024个节点时延迟接近5000纳秒。相比之下，LF 队列在所有批大小下都表现出近乎恒定的延迟，即使推入1024个节点时也保持在500纳秒以下。这表明，LF 队列能够有效摊销批量推送的成本，并提供不受批大小影响的可预测性能；在我们的工作负载中，一次会生成数百个节点，因此这一特性尤其有利。

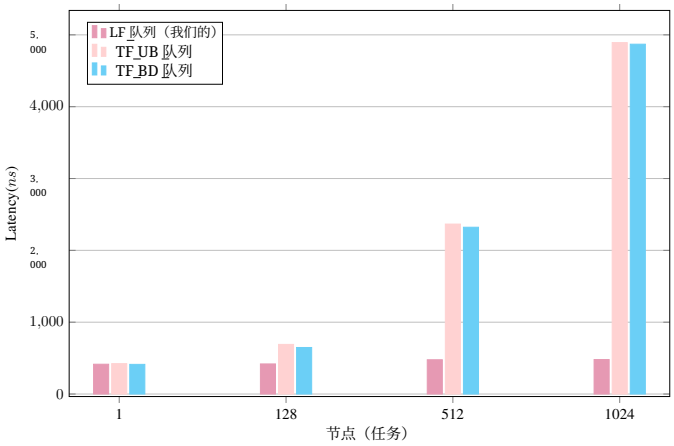

图6：推送操作的平均延迟随批大小的变化。所提出的队列（LF 队列）无论批次中推入的任务数量如何，延迟几乎保持恒定。相比之下，Taskflow有界队列（TF BD 队列）和无界队列（TF UB 队列）的延迟随批大小的增长而上升。

窃取操作的结果如图7所示。此处，横轴表示从队列中窃取的节点比例。例如，数值 20 表示窃取队列尾部的20%。在小规模窃取（10%）时，Taskflow 的有界队列和无界队列优于 LF 队列（$\approx 16\mu s$ vs$\approx 39\mu s$）。然而，随着被窃取节点比例的增加，Taskflow 的延迟几乎呈线性增长，在窃取队列的 60% 时超过 112μs。相比之下，LF 队列保持稳定性能，无论被窃取比例如何，延迟始终约为 40μs。这证明了 LF 队列批量窃取机制的效率，该机制将遍历开销分摊到多个节点上，并避免了重复同步。

可以对窃取 操作进行一项小优化。在原始版本中，窃取者会从头指针遍历整个队列直至尾部，即使已经在切分点处断开链表也不例外。由于所有者可能会并发推入或弹出元素，从而改变队列大小，因此必须进行这次第二次遍历，才能精确计算被窃取的节点数。相比之下，如果能够保证所有者在窃取期间没有更新队列，优化变体就会跳过这次第二次遍历。在这种情况下，可以根据初始

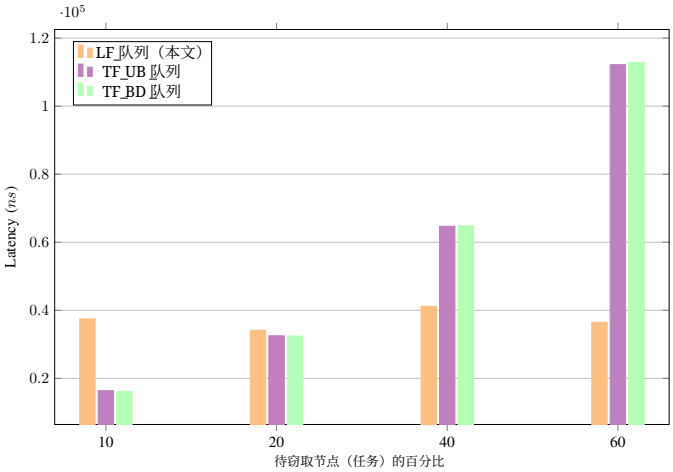

图 7：窃取操作的平均延迟随从初始大小为 10,000 的队列中被窃取任务比例的变化关系。10% 表示通过单次调用`steal`从队列尾部窃取 10% 的节点。所提出的算法（LF 队列）在较小比例（<30%）下的延迟高于 Taskflow（TF UB 队列和 TF BD 队列），而在较大比例下则优于另外两种实现。

大小和切割位置确定被窃取的节点数，并且在链表断开后立即返回。

图 8 展示了两种实现下窃取操作的延迟。非优化版本无论被窃取部分的比例如何，都需要约 37-38 μs，这反映了遍历整条链表的开销。相比之下，优化版本的延迟随比例增大而显著降低：从 10% 时的 37.7 μs 降至 60% 时的 12.5μs。这一改进源于优化版本避免了线性尾部遍历，从而将窃取开销分摊到队列中更大比例的部分上。

尽管这种情况在我们的目标工作负载中经常出现——工作线程通常会与主线程并行处理各自的节点并保持忙碌——但我们并未将其纳入基线算法。我们的目标是测量最坏情况下窃取 的延迟，此时遍历不可避免。尽管如此，这些结果表明，在实际应用中，这种优化可以在常见场景下显著降低窃取延迟。

对于弹出操作，三个队列的表现相近，平均延迟为 $\approx 213-216n$ 秒，这表明性能没有显著差异（图略）。

除微基准测试外，我们还在更贴近实际的工作负载下评估了我们的队列。理想情况下，这将涉及我们的并行求解器；然而，该求解器仍在开发中，并且由于随机化和内部子问题求解器，其工作负载难以模拟。作为近似，我们基于对大型有向无环图（有向无环图）的并行探索实现了伪工作负载。在此设置中，每个工作线程维护一个私有的工作量窃取队列，并从初始节点开始。探索过程中发现的出向节点被推入该工作线程的

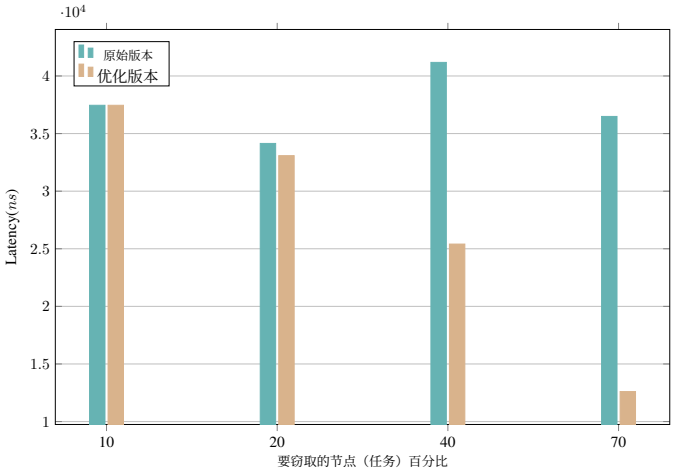

图 8：算法中窃取操作原始（未优化）版本与优化版本的平均延迟对比。优化版本在窃取过程中，若所有者未更新队列，则可避免遍历至队列末尾。随着被窃取任务比例增加，这项优化显著降低了延迟（降低 3 倍）。

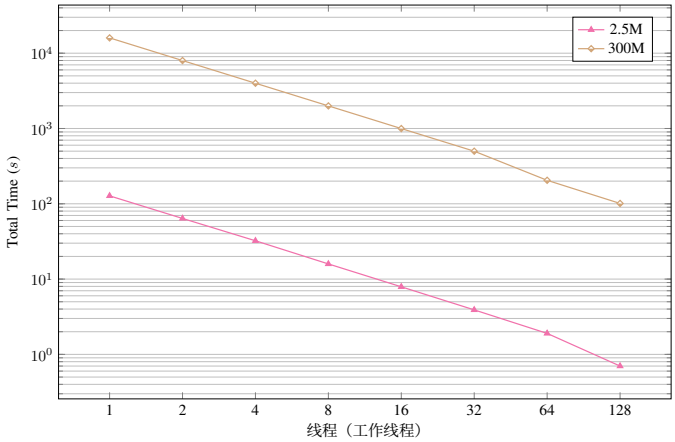

图 9：使用我们的队列进行有向无环图探索的执行时间，输入图规模分别为 2.5M 和 300M 个节点，线程数从 1 增加到 128。横轴和纵轴均采用对数刻度，以突出近似对数线性的可扩展性：线程数翻倍时，执行时间大致减半。

队列，前提是它们尚未被其他工作线程认领。当某个工作线程的队列变空时，它会尝试从另一个工作线程的队列中窃取一半节点，并按工作线程 ID 的顺序选择受害者。为了模拟单一并发窃取者，每个队列维护一个原子标志，窃取者可以在从此队列窃取之前和之后切换该标志。

我们构建了两个输入图，规模分别为 250 万个节点和 3 亿个节点。每项实验使用不同数量的线程执行，线程数从 1 增至 128，并测量总执行时间。图 9 所示的结果表明，我们的算法在两个输入图上均表现出稳定的可扩展性：线程数翻倍时，执行时间约减半，显示出接近对数线性加速的性能。这一趋势在所有输入规模下都一致，说明即使工作负载大幅增加，队列仍能保持高效。为直观展示这一特性，我们以对数刻度呈现结果。对应不同图规模的两组数据呈现出相同的斜率，凸显了算法的可扩展性不受输入图规模影响。

图 10 展示了上述实验中 TF UB 队列的性能。TF BD 队列的性能也非常相似，因此未在图中列出，以保持图表清晰。这些结果表明，在此伪工作负载中，单次队列操作的开销并非整体性能的主导因素。相反，可扩展性主要取决于有向无环图结构所固有的并行性以及探索任务的分布。

不过，需要注意的是，在这个有向无环图探索基准测试中，处理每个节点的成本是统一的。相比之下，我们的求解器中的节点处理时间差异很大，

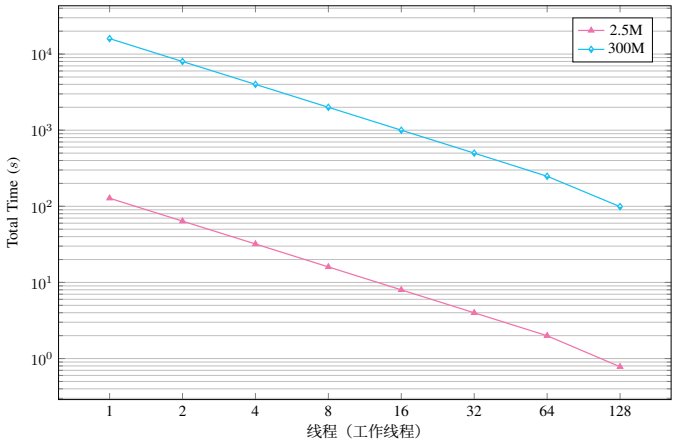

图 10：使用 TF UB 队列进行有向无环图探索的执行时间，输入图规模分别为 2.5M 和 300M 个节点，线程数从 1 增加到 128。横轴和纵轴均采用对数刻度。TF BD 队列呈现相似的斜率，因此未予展示。

并且取决于多种因素，例如解的可行性、正在进行的剪枝和细化，以及通过内部求解器求解随机子问题（对偶线性规划）。这种不规则工作负载更有可能放大队列开销的影响，尤其是在竞争激烈的情况下。因此，我们预计，将算法集成到求解器后，其批量操作和简化的并发模型在降低调度开销方面的优势将更加明显。

V. 相关工作

工作窃取逐渐成为一种有理论依据的策略，用于调度动态多线程计算，并提供严格的时间和空间保证

。Blumofe 和 Leiserson 的经典分析 [1] 表明，随机工作窃取队列能够在 P 个处理器上，以期望时间 $T_1/P + O(T_\infty)$ 执行工作量为 T_1 、跨度为 T_∞ 的计算；对于完全严格的计算，其空间使用量不超过顺序空间的一个常数倍。这一结果确立了工作窃取在嵌套并行中的高效性和稳健性，并为 Cilk 等系统奠定了理论基础。

为使工作窃取切实可行，需要高效的并发双端队列，以便所有者在一端推入/弹出，而窃取者从另一端窃取，并将竞争降至最低。Arora、Blumofe 和 Plaxton 的早期工作 [24]在多线程环境中提出了非阻塞调度器和双端队列接口，确立了无锁数据结构对可扩展调度的重要性。Hendler、Lev 和 Shavit [25] 沿此方向进一步提出了基于链式段的动态工作窃取队列，从而实现无界增长，但也引入了额外的复杂性。Chase 和 Lev [26] 随后提出了广为接受的动态循环工作窃取队列，这是一种按需增长的简单无锁环形缓冲区，并成为运行时系统的标准蓝图。

工作窃取支撑着若干主流运行时。Lea 的 Java Fork/Join 框架 [4], 后来成为 JDK 的一部分，将工作窃取引入了 Java 生态系统。Intel 的 Thread Building Blocks（TBB，现为 oneTBB） [3]在 C++ 中将工作窃取作为其核心任务调度策略。其他系统，例如 Go 运行时 [5] 调度器，也高度依赖工作窃取。这些框架验证了工作窃取在工业环境中的通用性与性能，但它们大体遵循 Chase-Lev 设计，并假定存在多个窃取者以及不规则工作负载的通用场景。

虽然 Chase-Lev 双端队列在顺序一致性下是正确的，但在 ARM 和 POWER 等弱内存架构上可能会失效。Lˆe、Pop、Cohen 和 Zappa Nardelli[27] 通过形式化证明 Chase-Lev 风格的双端队列在弱内存模型下的正确性，弥补了这一缺口。他们找出了在不同架构上保持线性化和无锁性所需的最小获取序与释放序栅栏集合，同时避免不必要的同步开销。基于这些设计衍生出的实用实现包括 C++ Taskflow 库 [28], , 该库提供有界和无界的无锁队列。有界版本采用环形缓冲区，适用于可预测工作负载；无界版本则支持动态扩展，适用于不规则工作负载。

除 Chase-Lev 之外，研究人员还提出了其他双端队列设计和调度策略。例如，Lace[29] 引入了一种分裂双端队列，其私有区域与共享区域之间的边界可以动态调整，从而降低竞争并尽量减少栅栏的使用。其他研究则探索了如何将工作窃取应用于具有依赖关系和异构工作负载的基于 DAG 的任务图。例如，Taskflow [28],采用工作窃取来调度任务构成的有向无环图，同时考虑任务依赖关系，体现了调度方式从扁平任务池向基于图的调度转变这一更广泛的趋势。

局部性与非统一内存访问感知是另一个重要方向。虽然工作窃取在负载均衡方面非常有效，但它并不考虑内存层次结构，在多插槽系统上，不加区分地窃取可能导致代价高昂的远程内存访问。Acar、Blelloch 和 Blumofe[30]早期的分析表明，工作窃取在期望情况下可以提供良好的缓存性能，但后续系统明确引入了局部性感知策略。Chen 和 Guo [31]提出了局部性感知工作窃取，并结合运行时性能分析，为多核系统自适应地进行受害者选择。这些成果表明，在均衡负载的同时保持局部性既可行又有益，而且任务在各核心上的放置与迁移会显著影响性能。

为摊销批量任务操作的调度开销，研究人员探索了批量窃取技术，使窃取者能够一次尝试窃取多个任务。一种广为人知的策略是窃取一半策略，即窃取者获取受害者队列中约一半的任务，而非仅窃取一个。Hendler 和 Shavit 提出的这一方法 [32], 可减少窃取频率，Go 和 Rust 的 Tokio 等现代运行时也采用该方法来提高吞吐量。批量窃取可以显著降低每个任务的同步开销，但也有过度窃取的风险：窃取超出所需的工作量，可能导致受害者线程利用不足，并因移动过多任务而增加开销。为解决这一问题，研究人员提出了自适应分块大小调整策略。Adnan 和 Sato [33] 开发了一种动态长度的工作窃取算法，可在单窃取者场景下根据受害者的负载调整被窃取的任务数量。该策略利用运行时信息（例如受害者任务栈的深度）为每次窃取尝试确定合适的块大小。与始终窃取固定数量任务的方法相比，这种自适应方法既能避免窃取不足，也能避免过度窃取。

沿相似思路的一项近期进展是基于块的工作窃取（BWoS） [34],，它推广了将双端队列划分为多个分块或块的思想。在 BWoS 中，每个每核任务队列由若干大小固定、具有独立元数据的块组成。所有者线程与窃取者仅在整个块的层面进行同步，而非针对单个任务。通过从中间窃取，BWoS 避免窃取者总是争用最近产生的任务；相反，较旧的块可以在极少协调的情况下被整体取走。工作窃取的领域专用扩展也随之出现，例如用于并行垃圾回收以及异构 CPU-GPU 系统。诸如 SLAW [35] 等自适应调度器会在工作优先与帮助优先策略之间切换，以根据程序状态同时优化开销与栈使用量。这些工作凸显了工作窃取作为一种通用范式的灵活性，可针对多样的工作负载和硬件架构进行调优。

我们的求解器采用集中式主从架构和结构化任务模式，因而有别于通用并行运行时。大多数现有的无锁工作窃取算法（例如 Chase-Lev）都假设采用单任务操作和多个并发窃取者，这与我们的求解器需求不符。具体而言，我们的工作负载涉及批量任务生成与传输，而现有算法只能通过反复执行单节点操作来模拟

这一过程，效率低下。我们还采用单窃取者并发模型，从而简化同步并降低原子操作开销。此外，我们需要队列能够无界增长且无需调整大小，而基于环形缓冲区或动态调整大小的双端队列无法高效支持这一需求。

尽管 BWoS 和 Lace 等设计在高度并发的通用场景中表现出色，但它们引入了一些复杂机制（例如块元数据、分割边界管理），而这些机制在我们的双线程并发模型中并非必需。此外，它们机会式且随机化的窃取方式与我们领域感知、由主线程控制的任务再分配策略并不相符。基于这些约束，我们开发了一种专用队列，支持原生批量操作、简化同步并可无限增长，以满足基于决策图的求解器的需求。

在评估中，我们将算法与 Taskflow 的两种实现进行了比较，因为这些实现易于获取，且能顺利集成到我们的测试框架中。尽管 Lace 或 BWoS 等其他无锁工作窃取算法提供了开源实现，但我们发现很难将其适配到实验环境中，尤其是因为它们的运行时假设和数据结构与我们的环境不同。我们展示了，我们针对求解器特定工作负载设计的算法，通过采用批量操作、无界队列增长以及简化的单所有者/单窃取者并发模型，实现了具有竞争力甚至更优的性能。

VI. 局限性与结论

在本研究中，我们提出了一种新的无锁工作窃取算法，专为满足基于图的优化求解器的需求而设计。与现有的通用算法不同，我们的队列支持批量操作，并假设至多有一个并发窃取者。这些设计选择使我们能够避免昂贵的同步，简化正确性推理，并实现高效的推入、弹出和窃取操作。

评估结果表明，所提出的算法优于 C++ Taskflow 中最先进的实现。最值得注意的是，该算法实现了常数延迟的推入性能，与 Taskflow 的队列形成鲜明对比：后者的延迟会随批大小增加而急剧上升。这使得在生成大量任务的场景中，性能表现可预测。所有实现的弹出操作性能相近。对于窃取操作，即使窃取操作所占比例增加，我们的队列仍能保持稳定的延迟，而 Taskflow 队列的性能则显著下降。我们还研究了一种优化的窃取变体：当所有者空闲时，该变体无需完整遍历即可完成操作，从而将延迟最多降低 3 倍。尽管为体现最坏情况性能，该优化未纳入基准实现，但它在实际工作负载中尤为重要。最后，在一个基于 DAG 探索的伪工作负载中，三种队列均呈线性扩展，且完成时间几乎相同，这反映了节点处理的成本均匀。

我们的研究也存在局限性。首先，虽然有向无环图工作负载能为可扩展性提供有益的见解，但它并不能完全捕捉目标求解器不规则且由启发式驱动的行为。在实际求解过程中，求解器的节点处理时间可能差异很大；我们预计这种差异会进一步放大批量操作和简化并发所带来的益处。不过，将该算法全面集成到求解器中仍有待未来工作。其次，我们的算法假设只有一个并发窃取者，并不直接支持多个并发窃取者。

尽管存在这些局限，我们的结果表明，针对特定领域的特性定制工作窃取算法，能够带来切实的收益。未来的工作将重点关注将该队列集成到求解器中，以验证其在真实工作负载下的性能；根据需要扩展设计以支持多个并发窃取者；并探索更多优化，例如提前返回窃取路径。

最后，我们强调，我们的算法并非旨在适用于所有场景时都优于通用工作窃取队列。特别是，我们并未声称其能提升小规模、细粒度窃取操作的性能，也未声称其适用于允许每个队列有许多并发窃取者的系统。我们的设计有意针对粗粒度任务生成、批量窃取和集中式调度的工作负载；在这些场景中，简化同步并摊销遍历开销，比降低单次窃取延迟更有益。

参考文献

[1] R. D. Blumofe 和 C. E. Leiserson，“通过工作窃取调度多线程计算”，《*ACM*期刊》，第46卷，第720–748页，1999年9月。[2] R. D. Blumofe、C. F. Joerg、B. C. Kuszmaul、C. E. Leiserson、K. H. Randall 和 Y. Zhou，“Cilk：一种高效的多线程运行时系统”，《并行与分布式计算期刊》，第37卷，69，1996 1，55第–页。[3] C. Pheatt，“Intel®线程构建模块”，《计算机科学学院期刊》，第23卷，第298页，2008年4月。[4] D. Lea，“一种 Java 分叉/汇合框架”，载于《*ACM 2000 Java Grande* 会议论文集》，JAVA ’00，（美国纽约州纽约市），第36–43页，计算机协会，2000年。[5] Golang，“Go 编程语言。” https://github.com/golang/go，2025年。[6] 维基百科贡献者，“双端队列——维基百科，自由的百科全书。” https://en.wikipedia.org/w/index.php?title=双端队列&oldid=1233074336，2024年。 [在线；访问于24-August-2025]。[7] D. Davarnia、J.-P. Richard、E. Icyuz 和 B. Taslimi，“具有不可分割节点流的网络模型及其在单元列车调度中的应用”，《运筹学》，第67卷，第1053–1068页，2019年。[8] H. Salemi 和 D. Davarnia，“利用决策图求解不可分割网络流问题”，《交通科学》，第57卷，第4期，第937–953页，2023年。[9] D. Bergman、A. A. Cire、W.-J. van Hoeve 和 J. Hooker，《用于优化的决策图》。施普林格国际出版，2016年。[10] W.-J. van Hoeve，“决策图优化导论”，《*INFORMS*运筹学教程》，第1–28页，2024年。[11] D. Davarnia，“基于决策图的连续非线性规划强松弛”，《运筹学快报》，第49卷，第2期，第239–245页，2021年。[12] D. Davarnia 和 W.-J. Van Hoeve，“通过决策图对整数非线性规划进行外逼近”，《数学规划》，第187卷，第111–150页，2020年。[13] D. Davarnia 和 M. Kiaghadi，“混合整数非线性规划全局优化的图形框架”，2024年。[14] H. Salemi 和 D. Davarnia，“可由决策图表示的混合整数规划的结构及其在机组组合中的应用”，《运筹学》，第71卷，第6期，第1943–1959页，2022年。

[15] D. Davarnia 和 M. Kiaghadi，“采用非凸正则化函数的统计模型参数估计的图形化全局优化框架”，载于《第28届人工智能与统计国际会议论文集》（Y. Li、S. Mandt、S. Agrawal 和 E. Khan 编），机器学习研究论文集第258卷，第3484–3492页，PMLR，2025年5月3日至5日。[16] E. Khademnia 和 D. Davarnia，“网络多胞体上的双线性项凸化”，《运筹学数学》，第50卷，第2期，第1019–1041页，2024年。[17] D. Bergman、A. A. Cire、A. Sabharwal、H. Samulowitz、V. Saraswat 和 W.-J. van Hoeve，“利用决策图进行并行组合优化”，载于《约束规划中的人工智能与运筹学技术集成》（H. Simonis 编），（Cham），第351–367页，施普林格国际出版，2014年。[18] D. Bergman、A. A. Cire、W.-J. van Hoeve 和 J. Hooker，《基于决策图的分支定界法》，第95–122页。Cham：施普林格国际出版，2016年。[19] B. Bloom、D. Grove、B. Herta、A. Sabharwal、H. Samulowitz 和 V. Saraswat，“Satx10：一种可扩展的即插即用并行SAT框架”，载于《可满足性测试理论与应用——SAT 2012》（A. Cimatti 和 R. Sebastiani 编），（柏林、海德堡），第 463–468 页，Springer Berlin Heidelberg，2012年。[20] L. Perron，“约束规划中的搜索过程与并行性”，载于《约束规划原理与实践——CP’99》（J. Jaffar 编），（柏林、海德堡），第346–360页，Springer Berlin Heidelberg，1999年。[21] T. Moisan、J. Gaudreault 和 C.-G. Quimper，“基于差异度的并行搜索”，第30–46页，2013年9月。[22]《组合优化中的局部搜索》。普林斯顿大学出版社，2003年。[23] A. Williams，C++ 《Concurrency in Action》。曼宁出版社，第2版，2019年。[24] N. S. Arora、R. D. Blumofe 和 C. G. Plaxton，“多道程序多处理器的线程调度”，载于《第十届ACM并行算法与体系结构年度研讨会论文集》，SPAA ’98，（美国纽约州纽约市），第119–129页，计算机协会，1998年。[25] D. Hendler、Y. Lev 和 N. Shavit，“动态内存ABP工作窃取”，载于《分布式计算》（R. Guerraoui 编），（柏林、海德堡），第188–200页，Springer Berlin Heidelberg，2004年。[26] D. Chase 和 Y. Lev，“动态循环工作窃取双端队列”，载于《第十七届ACM算法与体系结构并行性年度研讨会论文集》，SPAA ’05，（美国纽约州纽约市），第21–28页，计算机协会，2005年。[27] N. M. Lˆe、A. Pop、A. Cohen 和 F. Zappa Nardelli，“弱内存模型下正确且高效的工作窃取”，SIGPLAN 通讯，第48卷，第69–80页，2013年2月。[28] T.-W. Huang、D.-L. Lin、C.-X. Lin 和 Y. Lin，“Taskflow：一种轻量级并行异构任务图计算系统”，《IEEE并行与分布式系统汇刊》，第33卷，1320，2020 1303第–页。[29] T. van Dijk 和 J. C. van de Pol，“Lace：用于工作窃取的非阻塞分裂双端队列”，载于《Euro-Par 2014：并行处理研讨会》，（Cham），第206–217页，施普林格国际出版，2014年。[30] U. A. Acar、G. E. Blelloch 和 R. D. Blumofe，“工作窃取的数据局部性”，载于《第十二届ACM并行算法与体系结构年度研讨会论文集》，SPAA ’00，（美国纽约州纽约市），第1–12页，计算机协会，2000年。[31] Q. Chen 和 M. Guo，“基于在线性能分析和自动调优的多插槽多核架构局部性感知工作窃取”，ACM体系结构与代码优化汇刊，第12卷，2015年7月。[32] D. Hendler 和 N. Shavit，“非阻塞窃取一半工作队列”，载于《第二十一届分布式计算原理年度研讨会论文集》，PODC ’02，（美国纽约州纽约市），第280–289页，计算机协会，2002年。[33] ADNAN 和 M. SATO，“用于灵活负载均衡的动态多重工作窃取策略”，日本电子信息通信学会信息与系统汇刊，第E95.D卷，第6期，第1565–1576页，2012年。[34] J. Wang、B. Trach、M. Fu、D. Behrens、J. Schwender、Y. Liu、J. Lei、V. Vafeiadis、H. H¨artig 和 H. Chen，“BWoS：经形式化验证、用于并行处理的分块式工作窃取”，载于第17届USENIX操作系统设计与实现研讨会（OSDI 23），（波士顿，马萨诸塞州），第833–850页，USENIX协会，2023年7月。[35] Y. Guo、J. Zhao、V. Cave 和 V. Sarkar，《Slaw：面向多核系统的可扩展、局部性感知型自适应工作窃取调度器》，载于第*15*届*ACM SIGPLAN*并行编程原理与实践研讨会论文集，PPoPP ’10（美国纽约州纽约市），第341–342页，计算机协会，2010年。