arXiv:2304.13194v2 [cs.DC] 5 Jan 2024

Jet：图形处理单元上的多级图划分

MICHAEL S. GILBERT*、KAMESH MADDURI†、ERIK G. BOMAN‡、和
SIVASANKARAN RAJAMANICKAM§

摘要。 多层启发式算法是实现高质量串行和并行图划分的主流策略。划分细化是多级图划分的关键步骤。在本研究中，我们提出了 Jet，这是一种专为图形处理单元（GPU）设计的划分细化并行算法。我们将 Jet 与 GPU 感知粗化相结合，开发出一种 k路图分区器，即 Jet 分区器。与当前最先进的共享内存划分器相比，这种新的分区器在大量测试图上实现了更高的质量。

1. 引言。 并行图划分 [11] 是大规模图分析 [35,39]和高性能科学计算 [7,36]的关键推动因素。图划分的任务是在图中创建大小近似相等的不相交顶点集合，同时最小化割大小，即连接不同集合中顶点的边数。大多数图划分软件工具和算法都采用多层启发式算法。多层启发式算法在粗化 阶段构建一系列规模逐步缩小的图，在最小的图上求解问题（此处为图划分），然后将解解除粗化 以适应顶层图。解除粗化步骤还会利用序列中每个图的信息，通过称为细化的过程改进 解。图划分的细化算法通过移动顶点 来提高解的质量。图划分细化问题在多核系统 [3, 23, 33]的共享内存算法背景下已得到充分研究。我们的工作研究图形处理单元（GPU）上的划分细化问题，重点是达到或超越快速多核分区器所获得的划分质量。

我们的细化算法名为 Jet，将细化算法的两个主要任务解耦：改善割大小并维持解的平衡。这使我们的算法能够移动更大的顶点集合，而无需过度依赖细粒度同步。这对于实现高度并行性至关重要。此外，在割大小归约阶段，我们采用了一种新颖的启发式方法来选择要移动的顶点集合。该启发式方法使我们的细化算法能够在每次遍历中移动更大的顶点集合，并摆脱局部最小值。在这一启发式方法中，我们利用当前划分状态的信息为每个顶点分配优先级值，然后根据这些优先级近似计算下一划分状态的期望值。每个顶点邻域内下一划分状态的期望值决定了该顶点是否应当移动。在我们进行实验的大多数测试图上，与其他并行细化方案相比，这一启发式方法产生了质量更高的划分。

我们基于此前关于粗化的研究 [19]，并结合我们提出的新型细化算法 Jet，开发了一款面向图形处理器的分区器。GPU 加速使该分区器的划分时间始终快于其他分区器。在有限元方法、社交网络和半导体仿真等不同领域的图上，我们的分区器也始终能实现更小的割大小。

*宾夕法尼亚州立大学，大学公园市，美国（msg5334@psu.edu）。
†宾夕法尼亚州立大学，美国大学公园市（madduri@psu.edu）。
‡桑迪亚国家实验室，美国阿尔伯克基（egboman@sandia.gov）。§桑迪亚国家实验室，美国阿尔伯克基（srajama@sandia.gov）。

以下是主要的算法贡献和性能亮点：

- 我们提出了 Jet，这是一种新颖的高质量、GPU 并行的 k路细化算法。我们的实验表明，就图划分质量而言，Jet 的表现优于 MultitryLocalSearch 算法。
- 我们提出了一种 k路图分区器，即 Jet 分区器。该分区器利用 GPU 加速，在大多数测试案例中实现了比竞争方法 $2\times$更快的划分时间。我们还修改了 GPU 实现，使其适用于多核执行。
- 在由 60 多个图组成的多样化测试集上，与最先进的共享内存划分器相比，我们展现出了更优的质量。

2. 背景与先前工作。

2.1. 问题定义。 考虑一个具有 G 个顶点（或节点）和m 条边的图。我们假设该图是无向的，且不包含自环或重边。顶点可以带有关联的正整数权重。边权重为正整数，表示两个顶点之间连接的强度。顶点-权重对表示为 V，带权边三元组表示为 E。对于正整数 k， G的 k路划分是由 V的两两不相交子集组成的集合（或分区 $\{p_1, p_2, \ldots, p_k\} =P$），满足 $\cup_{i=1}^{k} p_i = V$。一个分区的权重/大小 p_i 是其所含顶点权重之和。给定一个划分，割集是边的集合 $\langle u,\ v,\ w_{uv}\rangle \in E$其中 u 和 v 属于不同的分区。割集中边权重之和称为该划分的代价（对于无权图，也称为割大小或边割）。形式为非负实常数 λ 的平衡约束会限制各分区的权重：$\text{weight}(p_i) \leq (1+\lambda)\frac{\text{weight}(V)}{k}, \forall\ 1 \leq i \leq k$；对分区大小没有下限。 λ 的值通常为 0.01–0.1（即允许 1–10% 的不平衡度）。 k路图划分问题的目标是在满足平衡约束的同时，最小化 G 的划分代价。图划分问题的输出通常是一个大小为 $n = |V|$ 的数组，用于将顶点映射到各个分区。

2.2. 多级划分。 多层启发式算法 [44] 被广泛用于大规模图分析。其应用包括图划分 [6, 27, 30],聚类 [15, 17, 20], 绘图 [29, 31], 和表示学习 [4, 12]。代数多重网格方法 [10, 46] 以及线性代数中的多层域分解方法 [26, 42] 与图分析中的多层方法密切相关。在多层方法中，我们不直接在大型图上求解问题，而是构建一个图层次结构，其中的图逐级小于原始图，同时保留原始图的结构。随后，我们在最小的图上求解问题，并使用该层次结构将解投影或插值到原始图上。算法 2.1 给出了多级图划分的高级模板。MLCoarsen返回一系列逐渐粗化的图及相应的顶点映射。 ProjectPartition 是一个轻量级例程，用于将解（划分数组）从上一级复制到当前级的顶点上。每一级投影之后都会进行细化，以减小割大小；如果较粗图上的划分无法满足平衡约束，还会通过细化使其满足该约束。

由于多级层次划分中的多级粗化、初始划分和细化三个阶段划分明确，因此本文重点关注多级细化。多级粗化的串行和并行算法已得到广泛研究 [13, 19, 30, 38]。

2.3. GPU：相关工作。大多数图分区器都是为中央处理器设计的，无法在 GPU 上运行。特别是，多层算法中的细化步骤

算法 2.1 多级图划分模板。

输入： 第 2.1 节中定义的图 G 、分区数量 k、平衡度 λ。
输出： 划分数组 $P_0[0..n-1]$，其中 $P_0[v]$ 表示顶点所属的划分 $v \in V$。

1: $\{G_0, \ldots, G_l\}, \{M_0, \ldots, M_l\} \leftarrow \text{MLCoarsen}(G)$
2: $P_l \leftarrow \text{InitialPartition}(G_l, k, \lambda)$
3: $P_l \leftarrow \text{RefinePartition}(G_l, P_l, k, \lambda)$
4: $i \leftarrow l - 1$
5: **当** $i \geq 0$ **时** ▷ l 解除粗化步骤
6: $P_i \leftarrow \text{ProjectPartition}(P_{i+1}, M_{i+1})$
7: $P_i \leftarrow \text{RefinePartition}(G_i, P_i, k, \lambda)$
8: $i \leftarrow i - 1$

难以在 GPU 上实现并行化。据我们所知，第一个 GPU 分区器由 Fagginger Auer 和 Bisseling [16]开发。他们为 GPU 开发了两种算法：一种是多层谱方法，另一种是采用贪心细化的多层算法。他们从未发布过代码。之后的一个 GPU 分区器 [21] 实现了采用基于标签传播的细化算法的多层算法。Sphynx [1, 2] 是一种在 GPU 上运行的谱划分器。它不是多层算法。尽管它在 GPU 上运行得相当快，但在非规则图上，其划分质量明显差于 Metis/ParMetis（最高可差 $50\times$）。因此，本文不再进一步考虑 Sphynx。

2.4. 细化。 划分细化问题的目标与图划分相同。细化算法用于改进输入划分；在多层方法中，该划分是多层层次结构中更粗层级的输出。无论生成给定划分所采用的方法如何，它也可以在多层图划分的语境之外使用。细化在本质上是局部的；当前划分状态的信息被用于生成下一个状态。细化方法通常使用一种名为增益的顶点属性，该属性根据当前划分状态定义。对于二划分，增益描述了将一个顶点从其当前部分移动到另一部分后割大小的减少量。如果割大小会增加，该值则为负数。当 $k > 2$时，我们使用增益 来表示一次顶点移动 的预期割大小减少量（将单个顶点从其当前部分移动到特定目标分区）。

细化算法通常在顶点集合上进行迭代。该集合可以是所有顶点，也可以是顶点的子集，例如边界集合，或其他感兴趣的子集。迭代次数通常是一个较小的常数，并且希望一次细化遍历的运行时间关于 $m = |E|$ 呈线性。若存在邻域中的顶点 u，使得部分$(v) \neq$ 部分(u)，则顶点 v 属于边界集合。由于边界集合之外的任何顶点都不可能具有增益为正的顶点移动，因此细化迭代通常只考虑该集合。

2.5. 细化：相关工作。本文关注并行划分细化方案。近期的几篇论文 [3, 25, 32–34] 表明，并行细化技术能够达到与其所衍生的串行细化算法相近或更好的质量。我们将算法归为四大类，并在下文中加以介绍。

2.5.1. 标签传播。有几种细化算法与标签传播（LP）社区检测算法的一次迭代具有相似之处 [37]。因此，我们将它们归为一类。在这些算法中，会检查每个顶点的邻域，以确定该顶点与哪个部分的连接最紧密。如果这样做不会违反平衡约束，就将该顶点移动到该部分。典型的串行实现会在每次迭代中以任意顺序访问图中的每个顶点，且每个顶点至多访问一次。常见的排序方式包括随机打乱和按顶点自然顺序排列。更复杂的排序方式则利用优先队列来确定能够使割大小降幅最大的顶点。这种技术无法摆脱局部最小值：在局部最小值处，不存在任何单个顶点能够在不违反平衡约束的情况下移动并降低割大小。在并行实现中，每个处理器拥有一个顶点子集，并按某种顺序访问自己拥有的顶点。如果在一次迭代检查完所有顶点后进行屏障同步，则该并行实现称为同步；如果部分变化立即生效，则称为异步。在并行环境下，可以通过原子方式更新分区大小来维持平衡约束。Mt-Metis [32]、mt-KaHIP [3]、KaMinPar [23]和 Mt-KaHyPar [25]均在多层级设置中实现了该算法的不同变体，将其作为一种细化选项或主要细化方法。PuLP [41]则在多层级框架之外，将该技术用于直接划分，并采用随机初始划分。据我们所知，最新的 GPU 分区细化算法 [21]采用同步方案来填充移动缓冲区。它每次考虑缓冲区中的前x 个移动，并从所有 2^x 种执行或不执行各项移动的排列中确定最佳方案。这使移动缓冲区的处理速率受到实际限制。

2.5.2. 局部化 FM 搜索。 Mt-KaHIP 的多试次局部搜索（MLS） [3] 和 Mt-KaHyPar 的并行 k 路 FM（KFM） [24,25] 搜索顶点移动序列，这些序列可能以负增益移动开始，但总体上能够改善割大小。这些算法放宽了广为人知的 FM 算法 [18]。每种算法都会从少量边界顶点开始多次 FM 风格的搜索，而标准 FM 则从所有边界顶点开始一次搜索。局部搜索会反复从优先队列顶端选择一个顶点进行移动；该队列按照将每个顶点移动到其他部分所能获得的最大增益排序。每次顶点移动都需要将其邻域插入队列，或更新已经在队列中的邻居。搜索会在队列耗尽或触发停止条件时结束。该停止条件基于搜索能够取得进一步改善的统计可能性。MLS 或 KFM 的一次迭代从边界顶点的无序列表开始，并从该列表中启动多次搜索，直到列表为空。多个局部搜索可以并行执行，但不超过可用线程数的限制。在一次迭代结束时，每次搜索执行的顶点移动会合并为一个序列，并提交该序列中满足平衡约束的最佳前缀。MLS 和 KFM 的区别在于：用于初始化每次搜索的顶点数量、并发搜索之间顶点移动的可见性，以及每次搜索如何拼接到全局序列中。根据作者的实验 [3]，mt-KaHIP 中的 MLS 细化生成的分区质量高于串行分区器 KaHIP [40] 的 fast 和 eco 配置，以及并行分区器 Mt-Metis 和 ParHIP。采用 KFM 细化的 Mt-KaHyPar- D（默认配置）生成的分区质量高于 mt-KaHIP [24]。

2.5.3. Hill-Scanning。 Hill-scanning 细化 [33] 是局部化 FM 搜索的另一种变体，但每次搜索在获得净正增益后会立即结束。搜索构建的移动序列称为一个 山丘，山丘的大小不能超过最大值（在 Mt-Metis 中为 16 个顶点）。总增益为正的山丘会应用于分区，否则将被丢弃。Hill-scanning 与 MLS/KFM 最重要的区别在于不再需要撤销移动。Hill-scanning 通过将顶点静态划分给各个处理器来利用并行性，但构建山丘的处理器可以使用另一个处理器所拥有的顶点。因此，山丘之间可能发生重叠。两个山丘之间的重叠可以在后续迭代中得到修正，但如果这样做会违反平衡约束，则可能无法修正。研究表明，Hill-scanning 技术的串行实现能够达到与其他串行细化方案 [33]相近或更高的质量，这些方案包括采用递归二分的 Fiduccia-Mattheyses（FM）、k路成对FM 和 Multi-try FM（MLS 和 KFM 的较弱前身）。与串行运行相比，使用 24 个线程运行时，Hill-scanning 生成的割大小约下降 0.5% [33]。mt-KaHIP 的作者 [3] 发现，当处理器数量较多时，Mt-Metis 中实现的 Hill-scanning 很难维持平衡约束。

2.5.4. 网络流方法。最大流最小割求解器作为划分细化算法取得了巨大成功 [22, 24, 40]。Mt-KaHyPar-Q（高质量配置）通过围绕两部分之间的边界扩展一个区域来构建网络流问题。它使用推送重标记算法的并行实现，在该区域内计算最小割；如果满足平衡约束，这个新割便会取代旧割。虽然基于流的方法在结果质量上优于其他细化方法，但其计算开销也大得多。

3. 我们的分区器。现在，我们将介绍新的多层次GPU划分器，重点讨论划分细化算法。我们持续进行粗化，直到得到的最粗图极小，通常包含 $4k$ 至 $8k$ 个顶点。我们使用 Metis [30] 中的k路划分方法进行初始划分。由于最粗图非常小，初始划分的GPU并行化留待未来工作。

3.1. 粗化。 我们的粗化方法基于 [19]，中讨论的一种 GPU 实现，具体而言是最初为 Mt-Metis 分区器开发的 两跳匹配 方法 [34]。该方法首先执行标准的重边匹配，仅当超过 25% 的顶点未匹配时，才会添加两跳匹配。

两跳匹配可分为三类：叶节点、孪生节点和相对节点。如果一对顶点在图中的距离为两跳，则它们是相对节点。孪生节点是相对节点的一个子集，其中两个顶点的邻域相同。叶节点是孪生节点的一个子集，其度为一。我们从两个方面扩展了先前的工作。首先，我们使用哈希方案进行孪生匹配。其次，我们使用媒合顶点实现相对匹配。媒合顶点是具有未匹配邻居的已匹配顶点，并在这些媒合顶点的邻域内进行匹配。我们不让度极高的顶点充当媒合顶点。此外，我们还将先前工作中的收缩方案替换为细粒度逐顶点哈希方案来进行去重，如算法 3.1 所述。

算法 3.1 边收缩算法。

输入： 第 2.1 节中定义的图 $G = (V, E)$。 映射向量 C。
粗化顶点数 n $_c$.
输出： 粗化边 E_c。

bound ← zeros(n_c)
并行 $v \in V$ **执行**
  bound$[C[v]]$ ← bound$[C[v]] + |E[v]|$
偏移量 ←独占前缀和(bound)
H_{key} ← 空值($|E|$) ▷ 初始化每顶点哈希表
H_{val} ← 零值($|E|$)
对于 $v \in V$ 并行执行 **执行**
  $v_c \leftarrow C[v]$
  $h_{key} \leftarrow H_{key}$[偏移量$[v_c]$..偏移量$[v_c + 1]$] ▷ 要使用的哈希表
  $h_{val} \leftarrow H_{val}$[偏移量$[v_c]$..偏移量$[v_c + 1]$]
  对于 $(u, w) \in E[v]$ 并行执行 **执行**
    $u_c \leftarrow C[u]$
    $i \leftarrow$ 插入或查找(h_{key}, u_c)
    $h_{val}[i] \leftarrow h_{val}[i] + w$
$E_c \leftarrow$ 提取插入操作(H, Hv)

3.2. Kokkos。 我们使用 Kokkos [45] 在代码中实现并行内核。Kokkos 有助于实现性能可移植性，使程序员能够维护一个可针对不同共享内存架构编译的单源程序。我们针对三种不同目标进行编译：使用 CUDA 后端的 NVIDIA GPU、使用 OpenMP 后端的多核 CPU，以及使用串行后端的同类 CPU 上的单线程。Kokkos 编程模型要求将任务表示为一系列小型内核，这些内核适用于三种并行原语之一：并行 for 、归约和扫描。

4. Jet 细化算法。 我们在 GPU 上进行细化时有两个设计目标：达到或超越多核细化技术的质量，以及实现与快速多核细化相当的运行时间。此前面向共享内存多核的细化算法（例如 Hill-Scanning 和 MLS）依赖线程本地优先队列。优先队列有助于寻找能够改善割大小的移动序列，而单次移动无法实现这一点。然而，这些优先队列操作无法提供足够的并发性来实现 GPU 级并行性，因此此类方法不适用于 GPU。大小约束的基于 LP 的迭代可以按任意顺序访问顶点，因此天然适用于多核并行性和 GPU 并行性。然而，大小约束限制了每次遍历中可移动的顶点数量。如果有利移动的分布偏向某些目标分区，这一问题尤其严重。为应对这一挑战，我们的方法 Jet 将一次大小约束的基于 LP 的迭代拆分为两个阶段。第一阶段是无约束 LP 阶段 Jetlp，在忽略大小约束的情况下执行顶点移动。第二阶段是再平衡阶段 Jetr，其任务是将顶点从超大分区移动到非超大分区，使得不再存在超大分区。再平衡阶段必须将割大小的任何增加（或损失）降至最低。基于 LP 的算法通常比基于 FM 的方法产生质量更低的结果，因此我们针对 LP 引入了新的增强方法，以提高质量。

我们的细化算法（算法 4.1）的总体结构是：持续应用 Jetlp，直到任一分区变为超大分区；然后持续应用 Jetr，直到恢复平衡。我们将 Jetlp 或 Jetr 的每次应用都记为一次“迭代”。我们根据割大小记录最佳平衡划分，并在连续超过一定次数的迭代（我们的结果使用 12 次）未遇到新的最佳划分时终止细化。我们还使用容差因子 ϕ，在割大小改善过慢时终止（见第 18 行）。 ϕ 是控制质量与运行时间权衡的最重要超参数，其中 $\phi = 1$ 可获得最佳质量。我们使用 $\phi = 0.999$，它在质量与运行时间之间取得了良好平衡。

算法 4.1 Jet 细化算法。

输入： 第 2.1 节中定义的图 $G = (V, E)$。分区数量 k，平衡因子 λ。划分数组 P_0。
输出： 输出划分数组 P_{best}。

```
1:  P_best ← P_0
2:  P_iter ← P_0
3:  DS ← 初始化数据结构(G, P_0, k)
4:  R ← ∅
5:  当未达到迭代上限时执行
6:      如果 imb(G, P_iter, k) < λ 则
7:          M ← Jetlp(G, P_iter, DS, R)
8:          R ← 顶点集(ML)
9:          重置弱再平衡计数器
10:     else
11:         如果 尚未达到弱再平衡限制 则
12:             M ← Jetrw(G, P_iter, DS, k, λ)
13:         else
14:             M ← Jetrs(G, P_iter, DS, k, λ)
15:     P_iter, DS ← updatePartsAndDS(G, P_iter, k, M, DS)
16:     如果 imb(G, P_iter, k) < λ 则
17:         如果 代价(G, P_iter) < 代价(G, P_best) 则
18:             如果 代价(G, P_iter) < φ*代价(G, P_best) 则
19:                 重置迭代计数器
20:             P_best ← P_iter
21:     否则，如果 imb(G, P_iter, k) < imb(G, P_best, k) 则
22:         P_best ← P_iter
23:         重置迭代计数器
```

4.1. 无约束标签传播 - Jetlp。 我们的无约束标签传播是同步的，即对划分状态的更新会延迟到每次迭代结束时进行。算法 4.2 的步骤如下：首先，算法为每个顶点 v 选择一个目标分区 $P_d(v)$，并记录单独执行此移动所产生的增益 $F(v)$。其次，它过滤出满足以下条件的顶点： $P_d(v)$ 与当前分区 $P_s(v)$ 不同，且增益 $F(v)$ 满足约束（公式 4.3）；然后将这些顶点推入无序列表，并将增益指定为优先级值。最后，它使用对下一划分状态期望值的近似值过滤该无序列表。对于通过第一轮过滤的每个顶点，它根据各邻域内的优先级值合并 P_s 和 P_d，从而在该顶点的邻域内确定这一近似值。它提交所有通过第二轮过滤的移动，并

随后更新用于跟踪每个顶点连通性和每个分区大小的数据结构。Jet 这一名称源于其结构与喷气发动机的相似性：目标分区的选择类似于压气机，第一轮过滤类似于燃烧室，第二轮过滤类似于加力燃烧室。

算法 4.2 Jet - 标签传播 (Jetlp)。

输入： 图 $G = (V, E)$。划分数组 P_s。用于查询的 数据结构 DS 顶点-分区连接信息。锁定顶点 $R \subset V$。过滤比率 c。
输出： 移动列表 M，其形式为顶点-目标部分对。

$P_d \leftarrow P_s$
$F \leftarrow$ 负无穷$(|V|)$
对 $v \in V \setminus R$ 并行**执行**
  $A_v \leftarrow$ 相邻分区$(v, DS) \setminus \{P_s[v]\}$
  如果 $A_v \neq \emptyset$ **则**
    $P_d[v] \leftarrow \text{argmax}_{p \in A_v}$连接度$(v, p, DS)$
    $F[v] \leftarrow$ 连接度$(v, P_d[v], DS) -$ 连接度$(v, P_s[v], DS)$
$X \leftarrow$ 增益连接度比过滤器$(V \setminus R, P_s, F, DS, c)$ ▷ 第一个过滤器（公式 4.3）
$F_2 \leftarrow$ 零$(|X|)$
并行遍历 $v \in X$ **执行**
  遍历 $(u, w) \in E[v]$ 并行执行 **执行**
    $p_u \leftarrow P_s[u]$
    如果 $ord(u) < ord(v)$ **则**
      $p_u \leftarrow P_d[u]$
    如果 $p_u = P_d[v]$ **则**
      $F_2[v] \leftarrow F_2[v] + w$
    否则如果 $p_u = P_s[v]$ **那么**
      $F_2[v] \leftarrow F_2[v] - w$
$M \leftarrow$ 非负增益过滤器(X, P_d, F_2) ▷ 第二个过滤器

4.1.1. 用于解决LP局限性的变更。基于LP的细化的同步实现存在两个局限性。首先，无法通过负增益顶点移动来改善割大小。其次，同一迭代中的顶点移动可能会相互产生不利影响。我们提出了一种解决这两个问题的方法：顶点后燃器。顶点后燃器是一种基于启发式的冲突解决方案，允许进行负增益顶点移动。我们使用“后燃器”这一术语，是因为它是可能顶点移动列表上的第二级筛选器；在典型的基于LP的细化算法中，只有第一级筛选器。给定潜在顶点移动列表 X，我们根据其邻域中下一划分状态的近似值，重新计算 X 中每个顶点的增益。通过使用排序ord将P_s与 P_d合并，可以生成该近似值。由于排序ord的影响，为重叠邻域生成的近似值并不一致。在应用顶点后燃器之前，会为 X 中的所有顶点固定 P_d，因此重新计算每个顶点 $v \in X$ 的增益时，只涉及该移动特有的部分 P_d（v）和 P_s（v）。对于顶点 $v \in X$的每个邻居u，如果$ord(u) < ord(v)$，我们会假设 u 将移动到 P_d（u），并据此计算 v的增益。否则，我们假设 u 保留在 P_s（u）中。这样一来，最初具有负增益的顶点移动可能会根据 X中的其他移动变为正增益，反之亦然。最终的移动列表 M 被选为X的一个子集，其中仅包含重新计算后具有非负增益的 X 中的移动。令

F（x） $=$连接度（x， P_d（x）） $-$连接度（x， P_s（x））是每个顶点移动的优先级值，其值由每个顶点移动在真空中的增益值给出。ord 定义如下：

$$\begin{cases} ord(u) < ord(v) & u \in X \wedge F(u) > F(v) \\ ord(u) < ord(v) & u \in X \wedge F(u) = F(v) \wedge u < v \\ ord(u) > ord(v) & \text{otherwise} \end{cases} \tag{4.1}$$

4.1.2. 负增益移动。 该过滤启发式方法的有效性取决于 X 的组成。如果 X 选择得过于保守（即仅选择正增益顶点移动），那么后燃相较于标准 LP 不会带来额外收益。如果不对 X 加以限制（即使用整个边界顶点集），那么后燃的结果将比标准 LP 更差。为了确定 X 的组成，我们必须首先确定每个顶点 v 的 $P_d(v)$：

$$P_d(v) = \text{argmax}_{p \in P \setminus \{P_s(v)\}} \text{conn}(v, p) \tag{4.2}$$

如果一个顶点仅与 $P_s(v)$ 相连接，那么它不是边界顶点，因此始终会被排除在 X 之外。选择顶点加入 X 的主要标准如下：

$$-F(v) < \lfloor (c)\text{conn}(v, p_s) \rfloor \vee F(v) \geq 0 \tag{4.3}$$

c 是一个可针对多层级次结构不同层级进行调整的常数。我们通过实验发现，对于该层级结构的最细层级， $c = 0.25$ 最为有效，而对于所有其他层级， $c = 0.75$ 最佳（针对我们的分区器而言）。需要注意向下取整，因为在某些图上，我们的结果对取整方向十分敏感。我们发现，粗化和初始划分算法会影响 c 的最优选择。

4.1.3. 顶点锁定。 我们采用了一种额外技术，旨在帮助边界在连续迭代中以协调的方式迁移。该技术使用锁定位，排除 Jetlp 的一次迭代中在 M 选出的所有顶点，使其无法在 Jetlp 的下一次迭代中被选入 X 。锁定有助于防止振荡，即顶点在连续的 Jetlp 迭代中在两部分之间来回移动。这些振荡可能会增加改变边界形状和位置的难度，从而降低解质量。锁不会影响再平衡迭代，再平衡也不会改变任何顶点的锁状态。

4.2. 再平衡 - Jetr。

4.2.1. 两部分。 我们介绍一种仅适用于 $k = 2$ 的更简单版本的再平衡方法。不失一般性，设 p_a 为超重部分，设 p_b 为另一部分。再平衡的目标是在最小化割大小增量的同时，将顶点从 p_a 移动到 p_b ，直到 p_a 不再超重。我们为 p_a 中的每个顶点分配一个简单的损失值：$\text{loss}(v) = \text{conn}(v, p_a) - \text{conn}(v, p_b)$。损失也可以指顶点集的组合损失：$\text{loss}(Z) = \sum_{v \in Z} \text{loss}(v)$。我们根据损失值的递增顺序，将 p_a 中的顶点排列成列表 L。然后，我们选择 L 的前缀 L_x ，使其最小化以下方程：

$$|(|L_x| - (|p_a| - (1 + \lambda)|V|/k))| \tag{4.4}$$

使用排序来获取 L 的代价很高，因此我们用 L' 近似 L，它按照偏序进行排序。这种偏序由损失值的以下函数推导得出：

$$\text{slot}(x) = \begin{cases} 2 + \lfloor \log_2(x) \rfloor & x > 0 \\ 1 & x = 0 \\ 0 & x < 0 \end{cases} \tag{4.5}$$

我们通过实验发现，随着损失值绝对值的增加，损失值出现的频率往往会降低。我们使用 $\log_2$ 分配槽位值，使得靠近零的位置比远离零的位置拥有更多槽位。这种偏序类似于一种用于计算向量 [5],中前 k 个元素的分桶方法，但它仅近似计算前 k 个元素，以节省时间。每个桶内的插入顺序会受到竞态条件的影响。为了减少 GPU 上各桶大小计数器的原子竞争，我们在每个桶内创建由 $v \bmod \rho$作为键的 ρ 子桶。这种面向桶的方法也非常适合在我们的 $k > 2$ 变体中独立地为多个超重部分计算列表。

THEOREM 4.1. 令 L'_x 为使公式 *4.4* 最小的 L' 前缀。对于具有均匀顶点权重的图，并假设负损失顶点的数量可以忽略不计，则有以下不等式：

$$\text{loss}(L'_x) \leq 2\,\text{loss}(L_x) \tag{4.6}$$

现在我们证明这个定理。 $|L'_x| = |L_x|$ 因为所有顶点的权重都相同。如果 L'_x 和 L_x 都为空集，则定理 4.1 显然成立。否则，令$s = \max_{v \in L'_x} \text{slot}(\text{loss}(v))$。令 S 为 p_a 的子集，其中包含所有槽位值小于或等于 s的顶点。根据定义， L'_x 是 S 的子集。 S 包含 p_a 中损失值小于某个关于 s的函数的所有顶点，因此它是L的前缀。于是， L_x 必定是 S的子集，因为 $|S| \geq |L'_x| = |L_x|$。类似的推理表明， S'（即 p_a 中所有槽位值小于 s的顶点构成的子集）是L_x 和 L'_x的真子集。我们已经证明， $L_x \triangle L'_x$ 是 $S \setminus S'$的子集。根据 $S \setminus S'$的定义， $L_x \triangle L'_x$ 只包含损失值等于 s的顶点。 $|L'_x \setminus L_x| = |L_x \setminus L'_x|$因为 $|L'_x| = |L_x|$。任意两个槽位值相同的顶点，其损失值之差不超过 2 的倍数，因此下列不等式成立：

$$\text{loss}(L'_x \setminus L_x) \leq 2\,\text{loss}(L_x \setminus L'_x) \tag{4.7}$$

由于我们假设负损失顶点数量可以忽略不计，因此损失$(L_x \cap L'_x) \geq 0$、损失$(L'_x \setminus L_x) \geq 0$和损失$(L_x \setminus L'_x) \geq 0$。我们可以在等式两边同时加上损失$(L_x \cap L'_x)$，从而得到等式 4.6。该不等式同样适用于我们的 k 路公式。均匀顶点权重的假设对于确保 $|L_x| = |L'_x|$ 是必要的。如果我们使用非均匀顶点权重，等式 4.6 将不再成立。在这种情况下，可以利用每个集合中槽位值为 s （即 $|L_x \setminus S'|$ 和$|L'_x \setminus S'|$）的顶点总数之比来构造一个新的不等式：

$$\text{loss}(L'_x) \leq 2\,\frac{|L'_x \setminus S'|}{|L_x \setminus S'|}\,\text{loss}(L_x) \tag{4.8}$$

4.2.2. 超过两个分区。 当 $k > 2$ 时，扩展这种再平衡公式并不简单。我们针对任意 k 提出了两种相互独立的扩展，它们都可以归约为 $k = 2$ 公式。与标签传播类似，输出由一个待移动顶点的无序列表及其选定的目标分区组成。设 B 为大小小于某个值 σ 的分区集合。 σ 确定了一个分区被视为有效目标分区时的最大大小，并且其取值应确保有效目标分区的大小与超大分区的大小之间存在死区。第一种公式使用以下损失定义：

$$\text{loss}(v) = \max_{p_b \in B} \text{conn}(v, p_b) - \text{conn}(v, p_a) \tag{4.9}$$

在这一表述中（详见算法 4.3），从超大分区中逐出顶点，使每个超大分区的大小刚好低于大小限制（应处于死区内）。这一过程与采用 $k = 2$ 的表述类似，只是这里存在多个超大分区。请注意，从第 21 行到第 28 行执行的多次扫描实际上只需两次扫描即可完成；为简洁起见，算法 4.3 中省略了这一细节。被逐出的顶点会被分配到有效目标分区中与其连接度最高的分区。如果某个顶点与任何有效目标分区都不相连，则会随机选择一个有效目标位置。在这一表述中，目标分区可能会变成超大分区。不过，死区可防止超大分区成为有效目标。若顶点权重均匀，这就保证最多经过 k 次迭代即可得到平衡划分，因为每次迭代至少会有一个分区进入死区。我们发现，通常所需的迭代次数远少于 k。由于可能需要许多次迭代，我们将这一扩展称为弱再平衡（Jetrw）。

设 A_v 为顶点 v的相邻分区。我们的第二种扩展采用以下损失定义：

$$\text{loss}(v) = \text{mean}_{p_b \in B \cap A_v} \text{conn}(v, p_b) - \text{conn}(v, p_a) \tag{4.10}$$

从超大分区中驱逐顶点的方式与前述公式相同。随后，目标分区会尝试从驱逐顶点集合中获取尽可能接近 $\sigma - |B|$ 个顶点。由于被驱逐的顶点被排列成无序列表，每个目标分区都会从该列表中选择一个连续组。目标分区会根据其容量叠加到无序列表上，形成一维的“饼干切割器”模式。如果顶点权重为单位权重，这一定式可保证单次迭代后不再有超大分区。我们观察到，当需要进行多次迭代时，顶点权重通常占大小约束的很大一部分。由于该扩展在大多数情况下能够通过一次迭代实现平衡，我们将其称为强再平衡（Jetrs）。

与 Jetrs 相比，Jetrw 在最小化损失方面有效得多，尽管它可能需要更多迭代才能收敛到平衡划分。我们的观察表明，在以下任一情形中，Jetrs 都需要更少的迭代才能收敛：正则图、 k 的值较小，以及不平衡比率较大。我们提出将这两种公式结合起来：先应用 Jetrw 若干次迭代（在算法 4.1 中记作 b_{max} ），如果划分仍然不平衡，则再应用 Jetrs。我们发现，即使先进行一次 Jetrw 迭代，再进行一次 Jetrs 迭代，也能获得无限次 Jetrw 迭代所带来的大部分收益。我们的完整

再平衡（Jetr）由两次 Jetrw 迭代和一次 Jetrs 迭代组成。如果由于顶点权重较大而需要更多迭代，则使用 Jetrs 执行这些迭代。对于这两种再平衡变体，我们发现，如果某个顶点的相应顶点权重大于 1。5($|p_a| - \frac{|V|}{k}$)，则限制该顶点离开超大分区是有益的。我们在构造 L' 之前应用这一限制。

算法 4.3 Jet - 弱再平衡。

输入： 图 $G = (V, E)$。不平衡分区数组 P_s。数据结构 DS 用于查询顶点-分区连接信息。 k。 λ。小桶数量 ρ。
输出： 移动列表 M，其形式为顶点-目标分区对。

$P_d \leftarrow P_s$
$o \leftarrow (1 + \lambda)|V|/k$
$\sigma \leftarrow$ 最大目标大小(o)
$A \leftarrow \{p \mid p \in P \wedge |p| > o\}$
$B \leftarrow \{p \mid p \in P \wedge |p| \leq \sigma\}$
$F \leftarrow$零值($|P_d|$)
对 $v \in V$ 并行 **执行**
  如果 $P_s[v] \in A$ 且 vtxWgt(v) $<$ 限制($P_s[v]$, $|V|$, k) **则**
    $A_v \leftarrow$ 相邻分区(v, DS) $\cap B$
    $P_d[v] \leftarrow \text{argmax}_{p \in A_v}$连接度($v$, p)
    if $A_v = \emptyset$ **然后**
      $P_d[v] \leftarrow$ 随机分区(B)
    $F[v] \leftarrow$连接度(v, $P_s[v]$) $-$连接度(v, $P_d[v]$)
$L' \leftarrow$ 桶($|A$|)
对 $v \in V$ 并行执行**执行**
  如果 $P_s[v] \in A$ **则**
    $s \leftarrow$ 槽位($F[v]$)
    writeTo桶($L'[P_s[v]][s][v$ 模 $\rho], v$)
$t \leftarrow 0$
$M \leftarrow$空列表
并行 $p_s \in A$ 执行 **do**
  $m \leftarrow 0$
  $m_{max} \leftarrow |p_s| - o$
  对 $v \in L'[p_s]$ 在 m 上并行扫描**执行**
    $m \leftarrow m+$顶点权重(v)
    如果 $m < m_{max}$ **then**
      $M[t] \leftarrow (v, P_d[v])$
      $t \leftarrow t + 1$

4.3. 数据结构与优化。 我们使用压缩稀疏行（简称 CSR 或 CRS）格式在内存中表示输入图和粗化图。为了支持 Jet 的迭代，我们需要一种数据结构来跟踪每个顶点与每个分区之间的连通性。我们的标签传播迭代必须能够快速识别每个顶点连接度最高和第二高的分区。我们的弱再平衡迭代必须识别超大分区中每个顶点连接度最高的有效目标分区。我们的强再平衡迭代必须计算超大分区中每个顶点与有效目标分区之间的连通性总和。最后，应当能够根据待移动顶点的列表修改这一数据结构，

朴素的实现可能会使用 $|V| * k$ space，为每个顶点与分区的可能组合显式跟踪这些连通性数据。遗憾的是，当 k 取合理值时，这会占用过多空间，并且在所有使用场景下遍历效率都很低。我们的实现基于以下观察：对于任意顶点 v，与其具有非零连通性的分区数量最多为 min(k,degree(v)). 我们采用类似于 CSR 图格式的表示方式来表示顶点-分区连通性矩阵。我们的数据结构分配的空间等于以下方程：

$$|V| + 2\sum_{v \in V} \min(k, \text{degree}(v)) \tag{4.11}$$

CSR表示中的每一行都被视为一个哈希表（以分区ID为键），用于创建和更新。为了确定相对于各个使用场景、满足某些筛选条件的连接度最高的分区，我们会对哈希表进行线性搜索。对于较小的哈希表，这种线性搜索的效率要高得多，因此我们限制空条目的数量。尽管对于每个顶点而言，$min(k,\ degree(v))$ 是可能的最大分区连接数，但我们观察到，许多图（尤其是正则图，甚至许多非规则图）在实践中具有少得多的非零连接。例如，一个度为 100 的顶点可能只有一到两个非零分区连接，其中 $k = 128$ 。我们将哈希表大小设置为略大于构建时的初始连通性。这可能导致哈希表达到此有限容量后，无法继续执行插入操作。发生这种情况时，我们会扩展哈希表容量并重新计算其内容。我们为每个哈希表分配少量额外空间，以限制执行此操作的频率。

为了在选定顶点移动列表 M 后更新此数据结构，

算法 4.4 Jet - 更新部分连接度。

输入： 图 $G = (V,\ E)$。划分数组 P_s。用于查询的 数据结构 DS
顶点-分区连接信息和锁定状态。顶点移动列表 M。
输出： 已更新的数据结构 DS

对于 $v \in$ 顶点集(M) 并行**执行**
  $p_s \leftarrow P_s[v]$
  对于 $(u,\ w) \in E[v]$ 并行**执行**
    $h \leftarrow$ getHashmap(DS, u)
    $h[p_s] \leftarrow h[p_s] + w$
    如果 $h[p_s] = 0$ **则**
      setOpen($h,\ p_s$)
对于 $(v,\ p_d) \in M$ 并行**执行**
  对于$(u,\ w) \in E[v]$ **并行执行**
    $h \leftarrow$ getHashmap(DS, u)
    如果 $p_d \notin h$ **则**
      插入($h,\ p_d$)
    $h[p_d] \leftarrow h[p_d] + w$

我们分两个阶段进行更新（见算法 4.4）。第一阶段针对 M 中每个顶点的每个邻居，递减其在相应源分区中的分区连通度，并将任何连通度降至零的分区替换为一个开放条目。第二阶段针对 M 中每个顶点的每个邻居，递增其在相应目标分区中的分区连通度

，必要时还会在哈希表中创建新条目。如果某些行当前的哈希表大小不足，第二阶段中新条目的创建可能会失败。我们标记相应的行，然后在第三阶段重新计算对应的哈希表（算法 4.4 中未显示）。所有阶段都利用原子操作来确保正确性，但竞态条件会影响任意给定分区将被分配到哈希表中的哪个条目。这种竞态条件也存在于数据结构的初始构建过程中。在 Jetlp 和 Jetrw 阶段，这可能会影响确定某个顶点的连通度最高分区时的平局处理方式。再加上 Jetrw 和 Jetrs 中桶插入所涉及的竞态条件，这些是 Jet 细化算法中仅有的非确定性来源。算法 4.4 的优点在于，它只更新与顶点移动相邻的行，以及这些行中受影响的特定条目。我们还实现了另一种更新算法：当顶点移动数量超过图中顶点总数的 10% 时使用该算法。在这种替代算法中，每次应用移动后，我们都会根据新的划分状态，重建与移动顶点相邻的每一行的整个哈希表。与算法 4.4 相比，这种方法减少了内存访问的不规则性，但代价是需要执行更多工作。

5. 实验设置。 我们的实验从割大小和总体执行时间两方面评估分区器的性能。我们将 GPU分区器与其他最先进的多核多层分区器进行比较，包括采用 Hill-Scanning 的 Mt-Metis v0.7.2、采用 MLS 的 mt-KaHIP v1.00、KaMinPar v1.0 和 Mt-KaHyPar-D v1.3.2，以及串行分区器 Metis v5.1.0。我们选择比较启用 Hill-Scanning 的 Mt-Metis 和启用 MLS 的 mt-KaHIP，是因为这些分别是其分区器可用的最高质量细化选项。我们采用 Mt-KaHyPar 的默认配置，因为在 v1.3.2 中，这是使用 KFM细化的最高质量配置。由于其他 GPU分区器 [16, 21], 的代码不可用，我们无法与其进行比较。在后续工作中， [21], 其割大小结果在所有测试图上都略差于 Metis 和 Mt-Metis（不使用 Hill-Scanning）。我们在 $k = 32$、$k = 64$、$k = 128$ 和 $k = 256$ 上进行评估，将不平衡度设为 3%；同时还在 $k = 128$ 上进行评估，将不平衡度设为 1% 和 10%。这构成了每个图和分区器共六个实验。尽管大多数分区器可以处理 k 的任意值，但 mt-KaHIP 不支持；因此，我们的实验使用 k 的值，且这些值为 2 的幂。对于每个图、实验和分区器的组合，我们通过多次运行收集中位割大小和中位运行时间。执行次数取决于分区器：mt-KaHIP MLS 运行五次，KaMinPar、Mt-Metis HS 和 Mt-KaHyPar-D 运行 11 次，Metis 运行三次，而我们的分区器运行 21 次。每个分区器的试验次数与其相应的运行时间大致成反比。我们按照实验配置，并根据图分类，分别给出与每个对比分区器的比较结果。

5.1. 有效性测试。 为了确定我们的细化方法的有效性，我们将其与 mt-KaHIP 的 MLS 和 Mt-KaHyPar-D 的 KFM 直接进行比较。由于这些是我们所比较的质量最高的分区器，因此它们是 Jet 细化的理想基准。为了将细化作为唯一变量，我们从对比分区器中导出粗化图层次结构和初始划分，并将其导入我们的程序。随后，我们使用 Jet 细化在导入的层次结构上细化该解。对于每次运行，我们计算两种细化方法所得最终割大小结果的比率。对于每个图，我们从多次测试运行中收集中位数比率（与 MLS 比较 5 次，与 KFM 比较 11 次）。我们还

对细化时间进行相同处理。我们还进行该实验的反向操作，将我们的粗化图和初始划分导入对比分区器。为确保比较公平，我们排除细化前具有不平衡划分的粗化图，并导出具有平衡划分的最粗图的划分结果。这是必要的，因为 MLS 假设输入划分是平衡的。在这些实验中，我们使用 $k = 64$，不平衡度等于 3%，并使用我们的 CPU平台，以进行公平的细化时间比较。

5.1.1. 运行时间。 我们将运行时间分为三类：粗化、初始划分和细化。我们分析运行时间扩展性相对于k 和不平衡度的变化。我们使用多个图族分析运行时间扩展性相对于图规模的变化。我们给出多核扩展性数据以及GPU与CPU性能。

5.2. 测试图。 我们的测试集（参见补充文件中的表 1）包含 Suitesparse图存储库中所有非零元至少为 5000 万但少于 7.5 亿的图 [14]（不包括 MAWI图）。此外，我们还纳入了 Open Graph Benchmark [28] 中的一些其他图（ppa、citation、products），以及由 Web算法实验室 [8, 9] 发布的一些社交网络（dblp10、amazon08、hollywood11、enwiki21），还有一个来自 Walshaw图基准 [43]的图（fe rotor）。我们还添加了一个 2000x4000 的矩形网格（grid）和一个 200x200x200 的立方体网格（cube）。我们通过执行以下步骤对所有图进行预处理：移除自环，将所有有向边转换为无向边，移除重复边，并提取最大连通分量。随后将这些图进一步分为九类。

5.3. 测试系统。 我们在两个不同的系统上进行测试。第一个系统配备32核 Ryzen 3970X Threadripper、256GB 内存（四通道 DDR4）。第一个系统用于运行竞争分区器的实验，以及用于获取我们的分区器扩展性结果的串行和多核实验。我们的 MLS 与 Jet 细化有效性测试也在该系统上运行。我们使用64线程运行每个分区器。第二个系统是一台虚拟机，配备 Intel Xeon Gold 6342 CPU 的12个虚拟核心、90GB 内存，以及一块拥有80GB 显存的 Nvidia A100 GPU。第二个系统用于运行我们的分区器的主要实验，以及 Jetlp 组件有效性实验。两个系统均运行 Ubuntu 20.04。对于 A100 平台，我们使用 CUDA Toolkit 11.6.2 版本的 NVCC 编译代码；对于 ThreadRipper 3970x 平台，则使用 g++ 版本 10.2.0。我们使用 Kokkos 和 Kokkos-Kernels 库的3.6.1发布版本，以及 Metis 库的5.1.0版本。

6. 分区器性能评估。

6.1. 质量。我们的 GPU分区器在割大小方面优于采用 Hill-Scanning（HS）的 Mt-Metis、KaMinPar 和 Metis。如图1所示，与 Mt-Metis HS 和 Metis 相比，我们的分区器在超过 90% 的测试实例上表现更好；与 KaMinPar 相比，在超过 80% 的实例上表现更好；与 mt-KaHIP MLS 相比，在约 70% 的实例上表现更好；与 Mt-KaHyPar-D 相比，在超过 60% 的实例上表现更好。

6.1.1. 实验配置。在表 1 中，我们注意到，除一个实验外，在所有实验配置中，我们的方法在割大小方面均比 Mt-Metis HS、KaMinPar 和 Metis 好 6% 以上。异常值是 $k = 256$ 实验，在该实验中，我们的方法比 KaMinPar 好 4.9%。总体而言，Mt-KaHIP MLS 产生的割比我们的方法差约 2.5%。唯一优于我们的方法的竞争者是 Mt-KaHyPar-D，其总体割大小好 0.5%。我们注意到，我们的方法在 $k = 128$ 和

表 1：我们将 Jet 分区器与各种分区器进行比较，报告使用该分区器获得的中位割大小的几何平均值与使用 Jet 分区器获得的中位割大小的几何平均值之比。数值大于 1 表示 Jet 分区器的性能更好。分区数量和平衡约束设置有所变化。

Partitioner	$k = 32$ i=3%	$k = 64$ i=3%	$k = 128$ i=3%	$k = 256$ i=3%	$k = 128$ i=1%	$k = 128$ i=10%
mt-KaHIP MLS	1.020	1.020	1.022	1.021	1.043	1.026
KaMinPar	1.084	1.074	1.063	1.049	1.067	1.073
Mt-Metis HS	1.111	1.094	1.084	1.073	1.075	1.100
Metis	1.099	1.085	1.072	1.063	1.069	1.088
Mt-KaHyPar-D	0.995	0.997	0.994	0.991	0.995	1.000

不平衡度为 10% 时，相较于其他各分区器，其表现优于 $k = 128$和不平衡度为 3% 的配置。在 $k = 128$ 和不平衡度为 1% 时，相较于 mt-KaHIP MLS 和 KaMinPar，我们的方法相对于相应的 3% 配置表现更好；而相较于 Mt-Metis HS 和 Metis，表现则相对更差。随着 k 的值增加，相较于 KaMinPar、Mt-KaHyPar-D、Mt-Metis HS 和 Metis，我们的方法表现相对更差。

6.1.2. 图类别。 我们使用图 2a 和图 2b，按照图类型对我们的分区器的优势和劣势进行分类。我们的分区器在有限元问题、优化问题、社交网络、半导体问题和人工复杂网络上表现出色。在社交网络中，我们的分区器仅未能在所有 amazon08 和 dblp10 实例（它们是测试集中的两个最小社交网络）以及一个 com-Orkut 实例上生成最佳划分。此外，我们在生物学图上具有中等优势：我们的方法在大多数 ppa 和 cage15 实例上获得了最佳划分，而 Mt-KaHyPar-D 在大多数 k-mer 图实例上获得了最佳划分。我们的弱项包括人工网格、网络爬取和道路网络。除网络爬取外，这些类别中的大多数图都具有潜在的二维结构。我们将在有效性测试部分探讨我们在网络爬取和二维问题上表现不佳的原因。

6.2. 运行时间。 我们的 GPU分区器始终比 CPU 竞争对手更快，在超过 85% 的测试实例中，其运行时间短于任何竞争对手。我们发现，我们的分区器在超过 99% 的测试实例中比 mt-KaHIP MLS 更快（见图 3），并且在超过 40% 的测试实例中快了二十倍以上。与 KaMinPar 相比，图 3 显示，我们的方法在超过 90% 的实例中更快，并且在超过 65% 的实例中至少快两倍。与 Mt-Metis HS 相比，我们的运行时间在超过 90% 的测试实例中更优，并且在超过 60% 的实例中至少快两倍。在超过 40% 的测试实例中，我们的分区器比 Metis 快 20 倍以上，与 mt-KaHIP MLS 的表现相近。在超过 70% 的测试实例中，我们的分区器至少比 Mt-KaHyPar-D 快 10 倍。mt-KaHIP 的运行时间表现令人意外；在图 4b 中，只有在我们的 $k = 32$ 和 $k = 64$ 实验中，其几何平均运行时间才优于 Metis。在同一图中，我们的分区器在不同实验配置下呈现出相似的性能趋势。在图 4a 中，我们的分区器在人工复形、生物学、道路网络、半导体和社交网络图上取得了出色的运行时间结果。在有限元图上，我们的分区器与 Mt-Metis HS 的运行时间性能相近；在网页

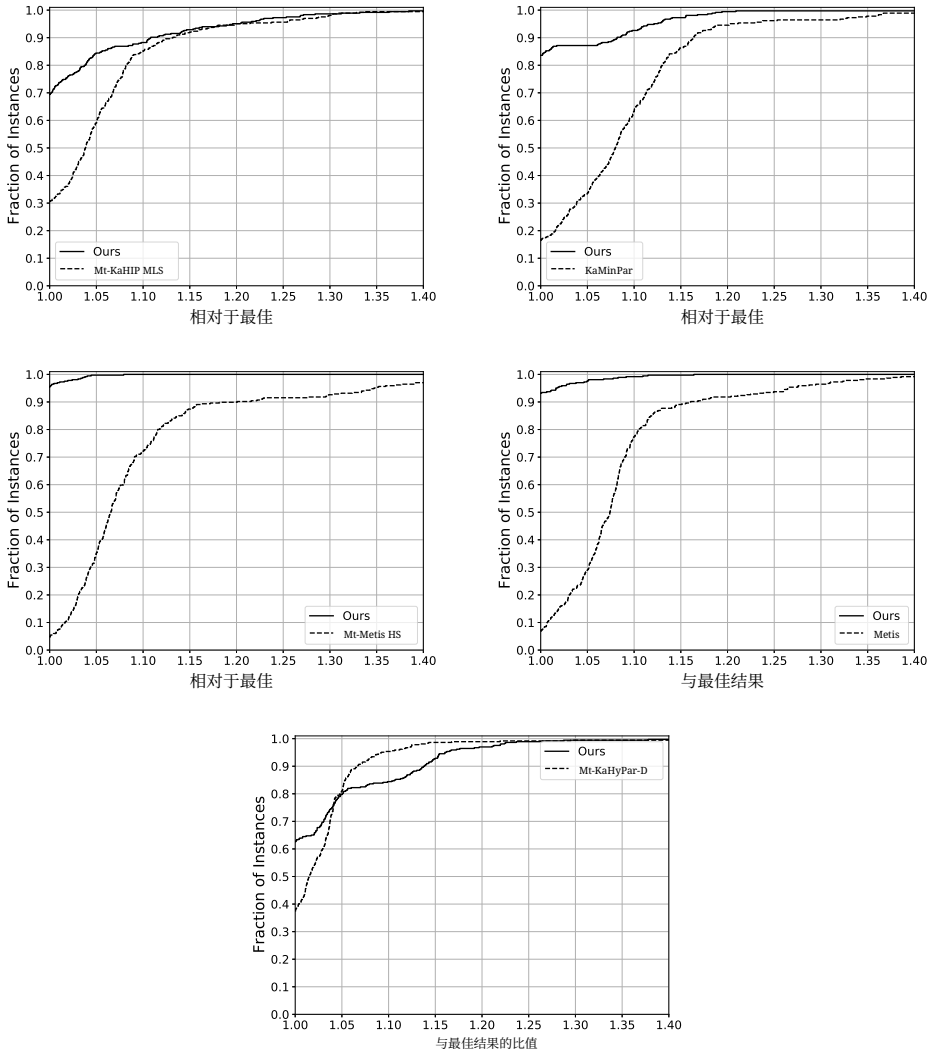


图 1：我们使用性能剖面比较使用我们的分区器与其他分区器获得的割大小。

爬取图上，它也实现了与 KaMinPar 相近的性能。KaMinPar 在 circuit5M 图上的性能远优于我们的分区器。

6.2.1. 时间分解。 在表 2 中，我们展示了图分区器在 GPU 上进行划分时，各个子任务所耗费的平均时间。我们按照图类别组织这些数据，并列出了该图类别中所有实验的平均总运行时间。在所有类别中，初始划分所占的总运行时间最多为 10.6%。在人工复形、半导体和网页爬取图上，粗化占据主导地位；而在其他图上，解粗化占据主导地位。对于度分布极不规则的大多数图，粗化往往占据主导地位；而对于更加规则的图，解粗化占据主导地位。

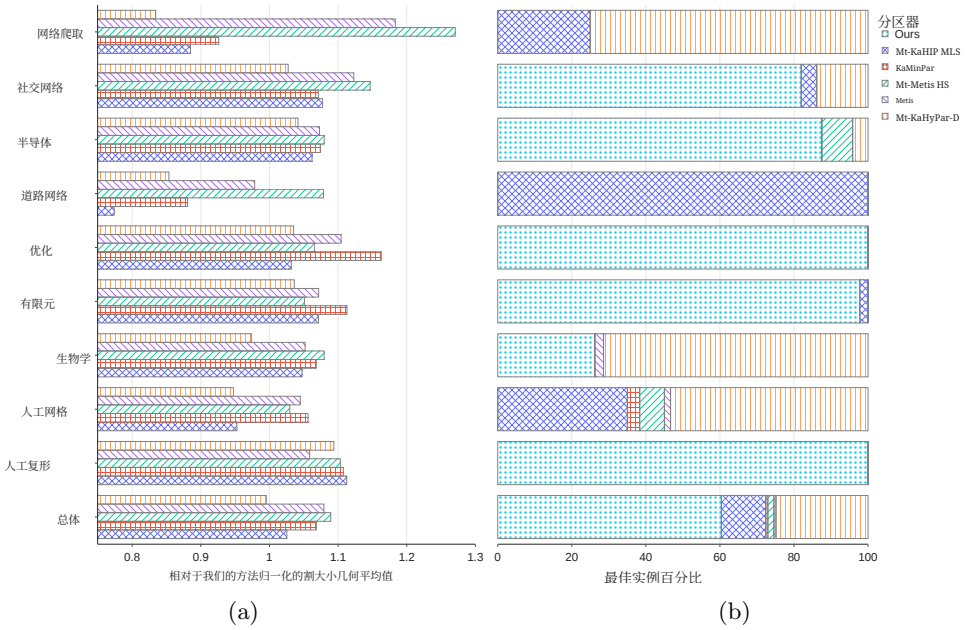


图2：按类别划分的割大小结果

表 2：按子任务划分的划分时间分解及平均总时间（秒）。

Graph Class	解除粗化（%）	粗化（%）	初始划分（%）	平均时间（秒）
网络爬取	30.3	64.6	5.1	1.38
社交网络	58.2	31.2	10.6	1.11
半导体	38.9	56.2	4.9	1.63
道路网络	52.6	43.8	3.6	0.43
优化	50.4	45.3	4.3	0.63
有限元	55.4	34.9	9.7	0.27
生物学	63.9	28.3	7.9	1.72
人工网格	64.6	31.2	4.2	0.41
人工复形	36.9	54.7	8.4	1.83

7. 细化性能评估。

7.1. 有效性测试。 在我们的划分有效性测试（表4）中，我们直接评估了 Jet 细化算法的性能。在第 6.1.2 节中，我们发现，我们的分区器在有限元问题、社交网络、人工复形网络、半导体问题、优化问题以及部分生物学问题上能够产生更优的划分结果。相反，我们发现，对于源自二维结构或网络爬取的问题，它产生的划分结果较差。此外，Mt-KaHyPar-D 在 k-mer图上获得了更优的划分结果。下面我们将详细介绍细化算法如何影响划分结果。

7.1.1. 优势。 总体而言，在采用 Mt-KaHIP 粗化和初始划分时，Jet 生成的割大小比 MLS 好 1.9%；采用我们的粗化和初始划分时，则好 6.2%。与 KFM 相比，使用 Mt-KaHyPar-D 后端时这一数值为 0.7%，使用我们的后端时为 3.4%。我们的细化算

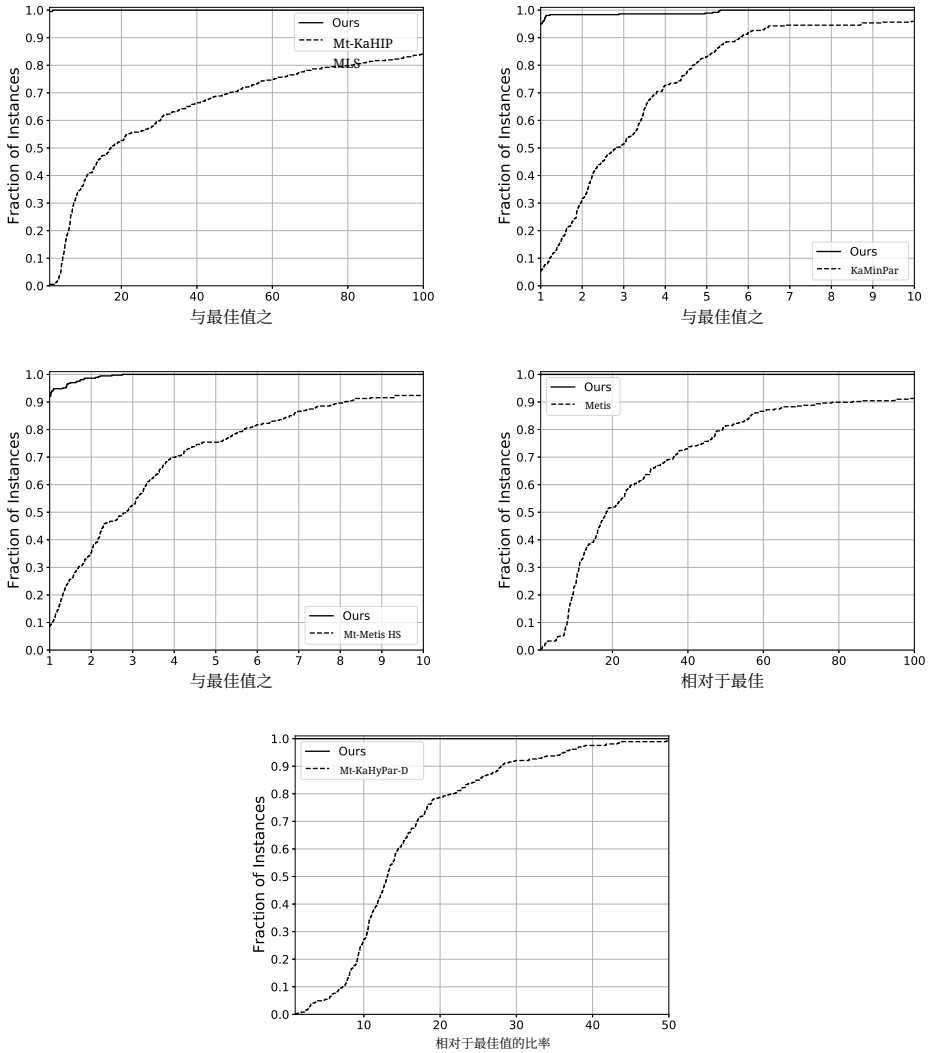
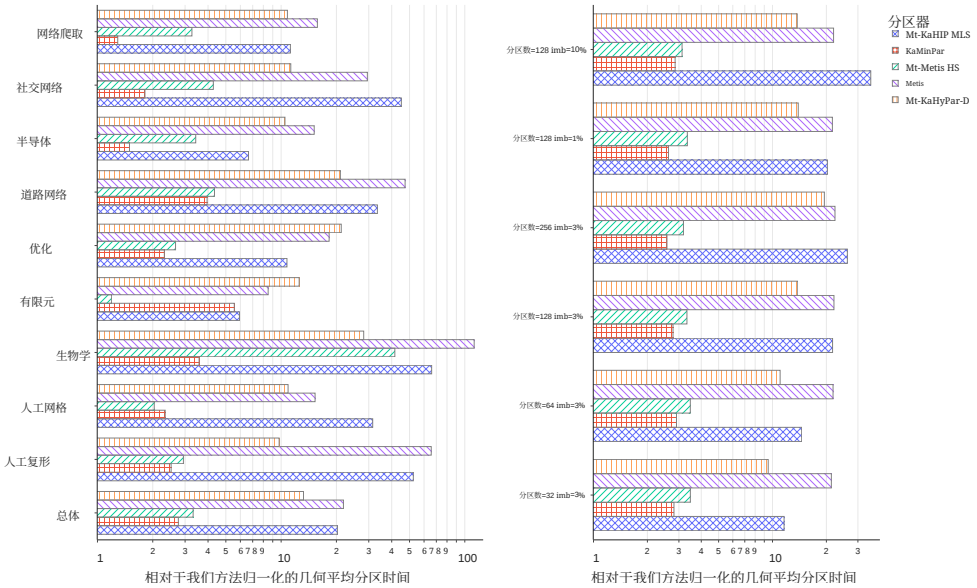


图3：我们使用性能剖面比较 Jet 分区器（我们的方法）在 A100 GPU 上的划分时间与其他分区器的执行时间。其他分区器运行于 AMD Ryzen Threadripper 3970x CPU 上。

法在有限元、生物学（不包括 k-mer 图）、社交网络、网络爬取和人工复杂网络上生成了更优的割大小。这排除了我们的细化算法导致分区器在网络爬取上的弱点这一可能性，因此我们的粗化和初始划分成为潜在原因。半导体和优化问题的割大小结果取决于后端。对于这些图中的大多数，我们的后端生成了总体上更优的割大小，而在这种情况下 Jet 表现更佳。对于 k-mer 图，我们发现，在两个后端的综合结果中，Jet 生成的割大小比 MLS 好约 0.8%，比 KFM 差 1.4%。值得注意的是，在使用我们的后端时，Jet 在道路网络上的表现优于 KFM。然而，我们的后端



（a）按图类别　　（b）按实验

图 4：分区时间比较

生成的道路网络总体切割结果比 Mt-KaHIP 或 Mt-KaHyPar-D 更差。

7.1.2. 弱点。对于具有二维结构的问题，包括大多数人工网格和道路网络，Jet 展现出的细化能力不如 MLS 和 KFM。我们推测，这一弱点与某些难以被我们的算法识别的改进类型有关。由于在每次迭代中，我们的标签传播阶段只能考虑当前位于边界上的顶点，因此难以找到能够显著改变边界位置的改进。对于图直径较大的图（例如二维网格），这一问题会更加严重。道路网络和人工网格（立方体网格除外）具有潜在的二维结构，因此其图直径为 $O(\sqrt{|V|})$。k-mer图的图直径同样较大。以我们的网格图为例，该图的图直径为 5998：对于 mt-KaHIP 后端和我们的后端，MLS 与 Jet 的比值分别为 0.902 和 0.906。作为对比，立方体图的图直径较小，为 597（这是三维网格的结果），其 MLS 与 Jet 的比值分别为 0.965 和 1.008。MLS 和 KFM 能够找到显著移动边界的改进，因为它们可以在图的局部区域内执行较长的移动序列。由于受到爬山规模上限的限制，HS 不具备这一能力。

7.1.3. 解粗化时间。在表 5 中，对于两种后端，Jet 的速度都比 MLS 快，且加速比超过 2.4 倍；对于两种后端，Jet 的速度都比 KFM 快至少 3 倍。除半导体图相较于 MLS，以及人工复杂图相较于两种竞争方法的情况外，Jet 在大多数图类别中都始终能够实现这些加速比。我们将 Jet 出色的解粗化速度归因于其批量同步设计、用于跟踪顶点-分区连通性的高效数据结构，以及不使用优先

表3：几何平均值（基线切割规模）/ 几何平均值（版本切割规模）。

基线 + 锁	基线 +弱 Afterburner	基线 + 完整 Afterburner	完整 Jetlp
1.000	1.009	1.030	1.052

表4：细化有效性——中位数比率的几何平均值（他们的割大小 / 我们的割大小）。

	MLS 与 Jet 对比	MLS 与 Jet 对比	K 路 FM 与 Jet 对比	K 路 FM 与 Jet 对比
Graph Class	Mt-KaHIP 后端	我们的后端	Mt-KaHyPar-D 后端	我们的后端
All	1.019	1.062	1.007	1.034
网络爬取	1.019	1.100	1.004	1.043
社交网络	1.082	1.173	1.040	1.079
半导体	0.998	1.063	1.006	1.029
道路网络	0.905	0.901	0.972	1.024
优化	0.985	1.013	0.998	1.010
有限元	1.019	1.037	1.000	1.021
生物学	1.034	1.059	0.995	1.023
人工网格	0.934	0.977	0.951	0.986
人工复形	1.148	1.167	1.132	1.122

队列。尽管在本实验中，为了进行公平比较，我们的代码采用了 CPU平台，但需要指出的是，为 MLS 或 KFM 实现 GPU 版本都并非易事。

7.1.4. 组件有效性。 在表3中，我们评估了与基线同步LP相比，我们的Jetlp阶段中各项设计选择的影响。我们的基线算法仅将顶点移动到其最佳连接分区中，完全省略了后燃器内核，并忽略锁定位。我们比较了LP阶段的四个版本。第一个替代版本是在基线算法的基础上加入顶点锁定。第二个替代版本是在基线算法的基础上加入较弱版本的加力燃烧室，该版本仅考虑具有正增益或零增益的顶点移动。第三个替代版本是在基线算法的基础上加入完整加力燃烧室，即考虑第4.1.2节所述的负增益顶点。第四个版本是完整 Jetlp 算法，即在基线算法的基础上加入顶点锁定和完整加力燃烧室。表3中的结果表明，与不考虑负增益顶点移动相比，加力燃烧室在能够考虑负增益顶点移动时表现明显更好。有趣的是，单独使用顶点锁定并不能带来任何收益，但与完整加力燃烧室结合后，相比不使用锁的完整加力燃烧室，其收益提高了2.2%。完整 Jetlp 相比基线算法带来的割大小收益从人工复形图中可忽略的0.1%到人工网格中显著的11.8%不等。此外，我们还研究了 ϕ 对最终版本执行时间和割大小结果的影响。我们发现，将细化容差值 ϕ 降低到0.99，可使反粗化时间缩短55%，但与默认值0.999相比，割大小恶化1.1%。将 ϕ 提高到0.9999，会使反粗化时间恶化34%，但与默认值相比，割大小改善0.5%。提高 ϕ 所带来的割大小收益，在人工网格中最为明显，在人工网格和网络爬取中最不明显。

7.2. 并行扩展性。 我们通过分析测试系统的相对性能，继续进行性能分析。在表6中，我们将多核 AMD Ryzen 3970x 中央处理器系统的性能与 Nvidia A100 GPU 系统进行比较，同时将中央处理器上的多核性能与串行性能进行比较。我们列出了总反粗化时间的结果，以及仅针对最细层

表 5：细化有效性：中位数比率的几何平均值（它们的反粗化时间 / 我们的反粗化时间）。

Graph Class	MLS 与 Jet Mt-KaHIP 后端	MLS 与 Jet 我们的后端	K 路 FM 与 Jet Mt-KaHyPar-D 后端	K 路 FM 与 Jet 我们的后端
All	2.418	2.498	3.005	3.328
网络爬取	1.713	2.457	3.455	4.806
社交网络	2.827	2.188	2.395	2.197
半导体	0.997	1.291	2.684	3.372
道路网络	3.083	3.271	4.787	5.961
优化	2.165	2.114	3.868	4.806
有限元	2.109	2.214	3.768	4.424
生物学	5.335	6.379	3.562	3.892
人工网格	1.759	1.519	2.903	3.005
人工复形	0.867	1.291	1.314	1.516

表6：解粗化加速比、总体细化和最细层细化。

Graph Class	A100 对比 3970x 32核 整体	A100 对比 3970x 32核 最细层	3970x 32核 vs 串行 整体	3970x 32核 vs 串行 最细层
网络爬取	5.14×	7.99×	9.28×	9.72×
社交网络	7.71	9.94	12.45	12.82
半导体	6.04	8.18	7.83	7.77
道路网络	6.53	14.77	7.95	8.46
优化	5.48	9.04	13.04	10.44
有限元	3.50	6.27	10.72	12.74
生物学	9.22	13.14	14.04	11.71
人工网格	4.33	8.96	9.72	10.61
人工复形	8.38	12.00	16.12	16.04

图的细化时间结果。32核的解粗化加速比介于7.8倍和16.1倍之间，较低端的表现并不理想。最细层细化加速比处于类似的7.8倍至16倍范围内。32核加速比不佳的部分原因在于内存带宽限制，以及为GPU 性能所采用的某些实现选择对 CPU 平台并不那么有效。GPU与中央处理器之间的解粗化加速比介于3.5倍和9.2倍之间。然而，GPU与中央处理器之间的最细层细化加速比更高，介于6.3倍和14.8倍之间。这很可能是因为主机设备同步时间在较小粗化图的细化时间中所占比例更大。

8. 结论。 我们展示了一种利用 GPU 加速来缩短划分时间，同时提供先进划分质量的分区器。与多个先进分区器相比，在六种实验配置下，我们的分区器在测试集中的九个图类别中的五个上展现出更优的质量。我们的运行时间在全部九个图类别上均更优。我们将这些结果归因于我们新颖的划分细化算法 Jet。Jet 建立在标签传播之上，通过解决许多常见缺点并针对 GPU 可扩展性进行优化，实现了改进。Jet 在九个图类别中的六个上提供了与两种先进 FM 细化并行实现相近或更优的割大小，并在九个图类别中的七个上实现了更优的运行时间。我们发现，二维网格状图上的质量是 Jet 的主要弱点，这与其他标签传播算法一致。通过将图粗化至极小规模，我们的分区器能够大幅减少初始划分所需的时间。我们计划研究提升 Jet 质量的方法，并在分布式内存划分器中展示 Jet。

致谢。 桑迪亚国家实验室是一家多任务实验室，由桑迪亚国家技术与工程解决方案有限责任公司管理和运营；该公司是霍尼韦尔国际公司的全资子公司，受美国能源部国家核安全管理局委托，根据合同 DE-NA-0003525 开展工作。本研究得到百亿亿次计算项目（17-SC-20-SC）的支持。该项目由美国能源部科学办公室与国家核安全管理局共同开展。

参考文献

[1] S. Acer, E. G. Boman, C. A. Glusa, and S. Rajamanickam, *Sphynx: A parallel multi-gpu graph partitioner for distributed-memory systems*, 《并行计算》, 106（2021）, 第102769页, https://doi.org/10.1016/j.parco.2021.102769。[2] S. Acer, E. G. Boman, and S. Rajamanickam, *SPHYNX: Spectral Partitioning for HYbrid aNd aXelerator-enabled systems*, 载于《国际并行与分布式处理研讨会论文集：研讨会》（IPDPSW）, 2020年。[3] Y. Akhremtsev, P. Sanders, and C. Schulz, *High-quality shared-memory graph partitioning*, 《IEEE并行与分布式系统汇刊》, 31（2020）, 第2710–2722页。[4] T. A. Akyildiz, A. A. Aljundi, and K. Kaya, *GOSH: Embedding big graphs on small hardware*, 载于《国际并行处理会议论文集》（ICPP）。[5] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, *Fast k-selection algorithms for graphics processing units*, 《ACM实验算法学报》, （）, https://doi.org/ / 2133803.2345676。[6] S. T. Barnard and H. D. Simon, *Fast multilevel implementation of recursive spectral bisectionfor partitioning unstructured problems*, 《并发：实践与经验》, 6（1994）, 第101–117页。[7] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP*, Oxford University Press, 美国, 2020年。[8] P. Boldi, M. Rosa, M. Santini, and S. Vigna, *Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks*, 载于第届《国际万维网会议论文集》（WWW）, 2011年。[9] P. Boldi and S. Vigna, *The WebGraph framework I: Compression techniques*, 载于第13届《国际万维网会议论文集》（WWW）, 美国曼哈顿, 2004年, 第595–601页。[10] A. Brandt, *Algebraic multigrid theory: The symmetric case*, 《应用数学与计算》, 19（1986）, 第23–56页。[11] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent advances in graph partitioning*, 《算法工程》, （）, 第–页。[12] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, *HARP: Hierarchical representation learning for networks*, 载于AAAI会议论文集。[13] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan, *Algorithm 1003: Mongoose a graph coarsening and partitioning library*, 《ACM数学软件汇刊》, （）。[14] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, 《ACM数学软件汇刊》, 38（2011）。[15] I. S. Dhillon, Y. Guan, and B. Kulis, *Weighted graph cuts without eigenvectors a multilevel approach*, 《IEEE模式分析与机器智能汇刊》, 29（2007）, 第1944–1957页。[16] B. Fagginger Auer, *GPU Acceleration of Graph Matching, Clustering, and Partitioning*, 博士论文, 乌得勒支大学, 2013年。[17] B. Fagginger Auer and R. H. Bisseling, *Graph coarsening and clustering on the GPU*, 《图划分与图聚类》, 588（2012）, 第223页。[18] C. M. Fiduccia and R. M. Mattheyses, *Linear-time heuristic for improving network partitions*, 载于第届《设计自动化会议论文集》（DAC）, 1982年, 第–页。[19] M. S. Gilbert, S. Acer, E. G. Boman, K. Madduri, and S. Rajamanickam,*Performance-portable graph coarsening for efficient multilevel graph analysis*, 载于《IEEE国际并行与分布式处理研讨会论文集》（IPDPS）, 2021年。[20] Y. Goldschmidt, M. Galun, E. Sharon, R. Basri, and A. Brandt, *Fast multilevel clustering*, 技术报告, 魏茨曼科学研究所。[21] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami, *High performance multilevel graph partitioning on gpu*, 载于《2019年高性能计算与模拟国际会议》（HPCS）, 2019年, 第769–778页, https://doi.org/10.1109/HPCS48598.

2019.9188120。[22] L. Gottesbüren, T. Heuer, and P. Sanders,
Parallel Flow-Based Hypergraph Partitioning，载于第20届《国际实验算法研讨会》（SEA），C. Schulz和B. U.car编，莱布尼茨国际信息学论文集（LIPIcs）第233卷，德国达格施图尔，2022年，Schloss Dagstuhl – Leibniz-Zentrum f¨ur Informatik，第5:1–5:21页。[23] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier,
Deep multilevel graph partitioning 29 204，载于第届《欧洲算法年会》（ESA），LIPIcs第卷，Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，2021年，第48:1–48:17页。
[24] L. Gottesbüren, T. Heuer, N. Maas, P. Sanders, and S. Schlag,
Scalable high-quality hypergraph partitioning 2023 2303 17679，https://arxiv.org/abs/。。
[25] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag,
Scalable shared-memory hypergraph partitioning 2021,，载于《算法工程与实验研讨会论文集》（ALENEX），第16–30页。[26] A. Heinlein, A. Klawonn, S. Rajamanickam, and O. Rheinbach,
FROSch: A fast and robust overlapping Schwarz domain decomposition preconditioner based on Xpetra in Trilinos，载于《科学与工程中的域分解方法》第二十五卷，Springer，2020年，第176–184页。
[27] B. Hendrickson and R. Leland, *A multi-level algorithm for partitioning graphs*，载于SC会议，美国加利福尼亚州洛斯阿拉米托斯，1995年12月，IEEE计算机学会，第28页，https://doi.org/ 10.1109/SUPERC.1995.3, https://doi.ieeecomputersociety.org/10.1109/SUPERC.1995.3。[28] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec,
Open Graph Benchmark: Datasets for machine learning on graphs，载于《神经信息处理系统年会论文集》，2020年。[29] Y. Hu and L. Shi, *Visualizing large graphs*，《WIREs计算统计学》，7（2015），第115–136页。[30] G. Karypis and V. Kumar,
A fast and high quality multilevel scheme for partitioning irregular graphs 20 1998，《SIAM科学计算学报》，（）。[31] Y. Koren, L. Carmel, and D. Harel,
Drawing huge graphs by algebraic multigrid optimization 1 2003 645 673，《多尺度建模与模拟》，（），第–页。[32] D. LaSalle and G. Karypis, *Multi-threaded graph partitioning*，载于《IEEE国际并行与分布式处理研讨会论文集》（IPDPS），2013年。[33] D. LaSalle and G. Karypis,
A parallel hill-climbing algorithm for graph partitioning，载于《国际并行处理会议论文集》（ICPP），2016年。
[34] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis,
Improving graph partitioning for modern graphs and architectures，载于《非规则应用：体系结构与算法研讨会》（IA3），2015年。[35] A. Lenharth, D. Nguyen, and K. Pingali, *Parallel graph analytics*，《ACM通讯》，59（2016），第78–87页。[36] A. Pothen,
Graph partitioning algorithms with applications to scientific computing，载于《并行数值算法》，Springer，1997年，第323–368页。[37] U. N. Raghavan, R. Albert, and S. Kumara,
Near linear time algorithm to detect community structures in large-scale networks 76 2007 036106，《物理评论E》，（），第页。[38] I. Safro, P. Sanders, and C. Schulz, *Advanced coarsening schemes for graph partitioning*，《实验算法学报》，19（2015）。[39] S. Sakr,
A. Bonifati H. Voigt A. Iosup K. Ammar R. Angles W. Aref M. Arenas M. Besta P. A. Boncz K. Daudjee E. D. Valle S. Dumbrava O. Hartig B. Haslhofer T. Hegeman J. Hidders K. Hose A. Iamnitchi V. Kalavri H. Kapp W. Martens M. T. Özsu E. Peukert S. Plantikow M. Ragab M. R. Ripeanu S. Salihoglu C. Schulz P. Selmer J. F. Sequeda J. Shinavier G. Szárnyas R. Tommasini A. Tumeo A. Uta A. L. Varbanescu H.-Y. Wu N. Yakovets D. Yan and E. Yoneki,
The future is Big Graphs: A community view on graph processing systems，《ACM通讯》，64（2021），第62–71页。
[40] P. Sanders and C. Schulz, *Engineering Multilevel Graph Partitioning Algorithms*，计算机科学讲义第6942卷，Springer Verlag，2011年，第469–480页，https://doi.org/ 10.1007/978-3-642-23719-5 40。[41] G. M. Slota, K. Madduri, and S. Rajamanickam,
PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks，载于《IEEE国际大数据会议论文集》（Big Data），2014年。[42] B. Smith, P. Bjorstad, and W. Gropp,
Domain decomposition: parallel multilevel methods for elliptic partial differential equations 2004，剑桥大学出版社。
[43] A. J. Soper, C. Walshaw, and M. Cross, *A combined evolutionary search and multilevel optimisation approach to graph-partitioning*，《全球优化学报》，29（2004），

第225–241页，https://api.semanticscholar.org/CorpusID:6904637. [44] S.-H. Teng, *Coarsening, sampling, and smoothing: Elements of the multilevel method*，载于《并行处理算法》，M. T. Heath、A. Ranade 和 R. S. Schreiber 编，Springer，1999年，第247–276页。[45] C. R. Trott, D. Lebrun-Grandie D. Arndt J. Ciesko V. Dang N. Ellingwood R. Gay- , , , , , , , , , , atri E. Harvey D. S. Hollman D. Ibanez et al. *Kokkos 3: Programming modelextensions for the exascale era*，《IEEE并行与分布式系统汇刊》，33 (2021)，第805–817页。[46] J. Xu and L. Zikatanov, *Algebraic multigrid methods*，《Acta Numerica》，26 (2017)，第591–721页，https://doi.org/10.1017/S0962492917000083.