SIAM J. SCI. COMPUT.
Vol. 46, No. 5, pp. B700–B724

JET: MULTILEVEL GRAPH PARTITIONING ON GRAPHICS PROCESSING UNITS*

MICHAEL S. GILBERT†, KAMESH MADDURI†, ERIK G. BOMAN‡, AND SIVA RAJAMANICKAM‡

Abstract. The multilevel heuristic is the dominant strategy for high-quality sequential and parallel graph partitioning. Partition refinement is a key step of multilevel graph partitioning. In this work, we present Jet, a new parallel algorithm for partition refinement specifically designed for graphics processing units (GPUs). We combine Jet with GPU-aware coarsening to develop a k-way graph partitioner, the Jet partitioner. The new partitioner achieves superior quality when compared to state-of-the-art shared memory partitioners on a large collection of test graphs.

Key words. graph partitioning, GPUs, multilevel, refinement

MSC codes. 68R10, 68W10, 05C85

DOI. 10.1137/23M1559129

See reproducibility of computational results at end of the article.

1. Introduction.

Parallel graph partitioning [11] is a key enabler for both large-scale graph analytics [35, 39] and high-performance scientific computing [7, 36]. Graph partitioning is the task of creating approximately equally sized disjoint sets of vertices in the graph, while simultaneously minimizing the *cutsize,* the number of edges connecting vertices in different sets. Most graph partitioning software tools and algorithms use the *multilevel* heuristic. The multilevel heuristic constructs a sequence of progressively smaller graphs in a *coarsening* phase, finds a solution to the problem (partitioning in this case) on the smallest graph, and then *uncoarsens* the solution to fit the top-level graph. The uncoarsening step also *improves* the solution using information from each graph in the sequence in a process called *refinement.* Refinement algorithms for graph partitioning work by *moving vertices* to improve the quality of the solution. The graph partition refinement problem is well studied in the context of

*Submitted to the journal's Software, High-Performance Computing, and Computational Science and Engineering section March 15, 2023; accepted for publication (in revised form) March 21, 2024; published electronically October 16, 2024.
https://doi.org/10.1137/23M1559129

Funding: Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute. This research was also supported in part by the U.S. National Science Foundation grants CCF-1955971, CCF-2119236, and CNS-2120361.

†Pennsylvania State University, University Park, PA, USA (msg5334@psu.edu, madduri@cse.psu.edu).

‡Sandia National Laboratories, Albuquerque, NM, USA (egboman@sandia.gov, srajama@sandia.gov).

shared-memory algorithms for multicore systems [3, 23, 33]. Our work considers the problem of partition refinement on graphics processing units (GPUs), with a focus on matching or exceeding partition quality obtained with fast multicore partitioners.

Our refinement algorithm, named Jet, decouples the two primary tasks of refinement algorithms: improving the cutsize and maintaining a balanced solution. This enables our algorithm to move larger sets of vertices without relying too much on fine-grained synchronization. This is critical for obtaining a high degree of parallelism. Moreover, we utilize a novel heuristic for selecting a set of vertices to move during the cutsize reduction phase. The heuristic enables our refinement to move larger sets of vertices in each pass, and to escape local minima. In this heuristic, we use information from the current partition state to assign a priority value to each vertex, and then approximate the expected value for the next partition state from these priorities. The expected value of the next partition state in the neighborhood of each vertex determines whether the vertex should move. On the majority of test graphs we experiment with, this heuristic results in higher-quality partitions than other parallel refinement schemes.

We develop a partitioner for GPUs utilizing our previous work on coarsening [19] and Jet, our novel refinement algorithm. GPU acceleration enables our partitioner to achieve consistently faster partitioning times compared to other partitioners. Our partitioner also achieves consistently smaller cutsizes on graphs from varied domains such as finite element methods, social networks, and semiconductor simulations.

The following are the key algorithmic contributions and performance highlights:

- We present Jet, a novel hiqh-quality, GPU-parallel, k-way refinement algorithm. Our experiments indicate that Jet outperforms the Multitry Local Search algorithm in terms of graph partition quality.
- We present a k-way graph partitioner, the Jet partitioner, that leverages GPU acceleration to attain 2× faster partitioning times than competing methods in a majority of test cases. We also modify the GPU implementation to adapt it for multicore execution.
- We demonstrate superior quality when compared to state-of-the-art shared memory partitioners on a diverse test set of over 60 graphs.

2. Background and prior work.

2.1. Problem definition. Consider a graph G with n vertices (or nodes) and m edges. We assume the graph is undirected and has no self-loops or parallel edges. Vertices can have associated positive integral weights. Edge weights are positive integers representing the strength of the connection of two vertices. Vertex-weight pairs are denoted by V, and weighted edge triples by E. For a positive integer k, a k-way partition of G is a set of pairwise disjoint subsets of V (or *parts* $\{p_1, p_2, \ldots, p_k\} = P$) such that $\cup_{i=1}^{k} p_i = V$. The weight/size of a part p_i is the sum of the weights of its constituent vertices. Given a partition, the *cut set* is the set of edges $\langle u, v, w_{uv}\rangle \in E$ with u and v in different parts. The sum of the weights of edges in the cut set is called the *cost* (or *cutsize,* or *edge cut* in case of unweighted graphs) of the partition. A *balance constraint* in the form of a nonnegative real constant λ places a limit on the part weights: $\text{weight}(p_i) \leq (1+\lambda)\frac{\text{weight}(V)}{k}\ \forall\ 1 \leq i \leq k$; there is no lower bound on the size of a part. The value of λ is typically 0.01–0.1 (or a 1–10% allowed imbalance). The objective of the k-way graph partitioning problem is to minimize the *cost* of the partition of G while satisfying the balance constraint. The output of the partitioning problem is typically an array of size $n = |V|$ mapping vertices to the parts.

Algorithm 2.1 A template for multilevel graph partitioning.

Input: Graph G as defined in section 2.1, number of parts k, balance λ.
Output: A partition array $P_0[0..n-1]$, where $P_0[v]$ indicates the partition that vertex $v \in V$ belongs to.

$\{G_0, \ldots, G_l\}, \{M_0, \ldots, M_l\} \leftarrow$ MLCoarsen(G)
$P_l \leftarrow$ InitialPartition(G_l, k, λ)
$P_l \leftarrow$RefinePartition(G_l, P_l, k, λ)
$i \leftarrow l - 1$
while $i \geq 0$ **do** ▷ l *Uncoarsening* steps
  $P_i \leftarrow$ ProjectPartition(P_{i+1}, M_{i+1})
  $P_i \leftarrow$ RefinePartition(G_i, P_i, k, λ)
  $i \leftarrow i - 1$

2.2. Multilevel partitioning. The multilevel heuristic [44] is extensively used in large-scale graph analysis. Its applications include graph partitioning [6, 27, 30], clustering [15, 17, 20], drawing [29, 31], and representation learning [4, 12]. The family of algebraic multigrid methods [10, 46] and multilevel domain decomposition methods [26, 42] in linear algebra is closely related to multilevel methods for graph analysis. In a multilevel method, instead of solving a problem on a large graph, we build a hierarchy of graphs that are progressively smaller than the original graph and yet preserve the structure of the original graph. We then solve the problem on the smallest graph and *project* or *interpolate* the solution to the original graph using the hierarchy. Algorithm 2.1 gives the high-level template for multilevel graph partitioning. MLCoarsen returns a sequence of coarser graphs and the corresponding vertex mappings. ProjectPartition is a lightweight routine that copies the solution (partition array) from the previous level to the vertices at the current level. Refinement is applied after projection on each level to reduce the cutsize, and to satisfy the balance constraint if the coarser partition could not satisfy it.

Since there is a clear separation of multilevel coarsening, initial partitioning, and refinement in multilevel level partitioning, we focus on multilevel refinement in this work. Sequential and parallel algorithms for multilevel coarsening have been extensively studied [13, 19, 30, 38].

2.3. GPU: Related work. Most graph partitioners are designed for CPUs and do not run on GPUs. In particular, the refinement step in the multilevel algorithm is difficult to parallelize on a GPU. The first GPU partitioner we know of was developed by Fagginger Auer and Bisseling [16]. They developed two algorithms for GPU: one was multilevel spectral, and the other was multilevel with greedy refinement. Their code was never released. A later GPU partitioner [21] implemented a multilevel algorithm with a label propagation-based refinement algorithm. Sphynx [1, 2] is a spectral partitioner that runs on GPUs. It is not multilevel. Although it runs quite fast on GPUs, the cut quality is significantly worse (up to 50×) than Metis/ParMetis on irregular graphs. Therefore, in this paper, we do not consider Sphynx any further.

2.4. Refinement. The objective of the partition refinement problem is identical to graph partitioning. Refinement algorithms improve an input partition; in the multilevel method, this partition is an output from coarser levels in the multilevel hierarchy. It can also be used outside the context of multilevel partitioning, regardless

of the method used to produce the given partition. Refinement is *local* in nature; information about the current partition state is used to generate the next state. Refinement methods frequently use a vertex attribute called the *gain,* which is defined according to the current partition state. For bipartitioning, the gain describes the decrease in cutsize for moving a vertex from its current part to the other part. This quantity is negative if the cutsize would increase. When $k > 2$, we use *gain* to indicate the expected decrease in cutsize for a *vertex move* (moving a single vertex from its current part to a specific destination part).

Refinement algorithms typically operate in *iterations* over a set of vertices. The set can be either all vertices, or a subset of vertices such as the *boundary* set, or some other subset of interest. The number of iterations is typically a small constant, and it is desirable that the running time of one refinement pass be linear in $m = |E|$. A vertex v is in the boundary set if there exists a vertex u in the neighborhood of v such that $\text{part}(v) \neq \text{part}(u)$. As no vertex outside the boundary set can have a positive gain vertex move, it is common for refinement iterations to exclusively consider this set.

2.5. Refinement: Related work. In this work, we are interested in parallel partition refinement schemes. Several recent papers [3, 25, 32, 33, 34] have demonstrated that parallel refinement techniques can obtain a quality that is similar to or better than the sequential refinement algorithms from which they are derived. We group algorithms into four broad categories and describe them below.

2.5.1. Label propagation. Several refinement algorithms share similarities with an iteration of the Label Propagation (LP) community detection algorithm [37]. Thus, we group them into a common category. In these algorithms, the neighborhood of each vertex is examined to determine the part to which the vertex is most connected. The vertex is then moved to this part if doing so does not violate the balance constraint. A typical serial implementation visits each vertex of a graph at most once per iteration, in arbitrary order. Common orderings include random shuffles and the natural order of the vertices. More complex orderings make use of priority queues to determine the vertex which results in the largest decrease in cutsize. This technique cannot escape local minima, which occur when no single vertex can be moved for a decrease in cutsize without violating the balance constraint. In parallel implementations, each processor owns a subset of the vertices, and each processor visits the vertices it owns in some order. A parallel implementation is synchronous if there is a barrier synchronization after all vertices are inspected in an iteration, or termed asynchronous if part changes are immediately applied.

The balance constraint can be maintained in a parallel setting by atomically updating the part sizes. Mt-Metis [32], mt-KaHIP [3], KaMinPar [23], and Mt-KaHyPar [25] all implement variations in a multilevel setting as a refinement option or the primary refinement method. PuLP [41] implements this technique for direct partitioning outside of a multilevel framework, using random initial partitions. The latest GPU partition refinement algorithm [21] that we know uses a synchronous scheme to fill a move buffer. It considers the top x moves in the buffer at a time and determines the best of all 2^x permutations of performing or not performing each move. This imposes a practical limit on the rate at which the move buffer can be processed.

2.5.2. Localized FM search. Mt-KaHIP's multitry local search (MLS) [3] and Mt-KaHyPar's parallel k-way FM (KFM) [24, 25] search for sequences of vertex moves that may begin with a negative gain move but collectively improve the cutsize. These

algorithms relax the well-known Fiduccia–Mattheyses (FM) algorithm [18]. Each algorithm begins multiple FM-style searches seeded from a small number of boundary vertices, whereas the standard FM performs a single search seeded from all boundary vertices. A local search repeatedly selects a vertex to move from the top of a priority queue, which is keyed by the maximum gain of moving each vertex to another part. Each vertex move requires inserting its neighborhood into the queue, or updating its neighbors already within the queue. A search ends when it has exhausted its queue or when a stopping condition is triggered. This stopping condition is based on the statistical likelihood that a search will yield further improvement. A single iteration of MLS or KFM begins from an unordered list of boundary vertices and seeds many searches from this list until the list is empty. Multiple local searches can occur in parallel, up to the limit of available threads. At the end of an iteration, the vertex moves performed by each search are joined into a single sequence, and the best prefix of this sequence that satisfies the balance constraint is committed. MLS and KFM differ in terms of how many vertices are used to initialize each search, the visibility of vertex moves between concurrent searches, and how each search is concatenated into the global sequence. The MLS refinement in mt-KaHIP produces higher quality partitions than the fast and eco configurations of the serial partitioner KaHIP [40] and the parallel partitioners Mt-Metis and ParHIP, according to the experiments of the authors [3]. Mt-KaHyPar-D (default configuration) using KFM refinement produces higher quality partitions than mt-KaHIP [24].

2.5.3. Hill-scanning. Hill-scanning refinement [33] is another variant of localized FM search, except that each search immediately ends when achieving a net positive gain. The sequence of moves built by a search is termed as a *hill,* and hills cannot grow beyond a maximum size (16 vertices within Mt-Metis). Hills which attain positive total gain are applied to the partition; otherwise they are discarded. The most significant difference between hill-scanning and MLS/KFM is the elimination of any need to revert moves. Hill-scanning exploits parallelism by statically dividing the vertices among the processors, but a processor that is building a hill can use vertices owned by another processor. In this way, hills may overlap. Overlap between two hills can be corrected in successive iterations, but this may not happen if doing so would violate the balance constraint. A serial implementation of the hill-scanning technique was shown to attain similar or superior quality to other serial refinement schemes [33] including Fiduccia–Mattheyses (FM) with recursive bisection, k-way pairwise FM, and Multitry FM (a weaker precursor to MLS and KFM). The cutsizes produced by hill-scanning degrade by about 0.5% when run with 24 threads instead of serially [33]. The authors of mt-KaHIP [3] found that hill-scanning as implemented in Mt-Metis has substantial difficulty maintaining the balance constraint when the number of processors is large.

2.5.4. Network flow methods. Max-flow min-cut solvers have seen great success as partition refinement algorithms [22, 24, 40]. Mt-KaHyPar-Q (high quality configuration) creates a network flow problem by growing a region around the boundary between two parts. It uses a parallel implementation of the push-relabel algorithm to compute a minimum cut inside this region, and this new cut replaces the old cut if it satisfies the balance constraint. While flow-based methods outperform other refinement methods in terms of result quality, they are also considerably more expensive.

3. Our partitioner. We now discuss our new multilevel GPU partitioner with an emphasis on the partition refinement algorithm. We coarsen until the coarsest

Algorithm 3.1 Edge Contraction Algorithm.

Input: The graph $G = (V, E)$ as defined in section 2.1. The mapping vector C. Coarse vertex count n_c.
Output: The coarse edges E_c.

```
bound ← zeros(n_c)
for v ∈ V in parallel do
    bound[C[v]] ← bound[C[v]] + |E[v]|
offsets ← exclusivePrefixSum(bound)
H_key ← nulls(|E|)                          ▷ Initialize per-vertex hash table
H_val ← zeros(|E|)
for v ∈ V in parallel do
    v_c ← C[v]
    h_key ← H_key[offsets[v_c]..offsets[v_c + 1]]      ▷ Hash table to use
    h_val ← H_val[offsets[v_c]..offsets[v_c + 1]]
    for (u, w) ∈ E[v] in parallel do
        u_c ← C[u]
        i ← insertOrLookup(h_key, u_c)
        h_val[i] ← h_val[i] + w
E_c ← extractInsertions(H, Hv)
```

graph obtained is extremely small, typically between $4k$ and $8k$ vertices. We use the k-way partitioning method in Metis [30] to perform the initial partitioning. Since the coarsest graph is very small, GPU parallelization of the initial partitioning is left for future work.

3.1. Coarsening. Our coarsening approach is based on a GPU implementation discussed in [19], specifically the *two-hop matching* approach originally developed for the Mt-Metis partitioner [34]. This approach begins with a standard heavy-edge matching and only adds two-hop matches if more than 25% of all vertices are unmatched.

Two-hop matchings can be split into three categories: leaves, twins, and relatives. A pair of vertices are relatives if they are separated by a distance of two in the graph. Twins are a subset of relatives where the neighborhoods of both vertices are the same. Leaves are a subset of twins that have degree one. We extended our previous work in two ways. First, we use a hashing scheme to perform twin matching. Second, we implement relative matching using *matchmaker vertices.* Matchmaker vertices are matched vertices with unmatched neighbors, and matches are performed within the neighborhoods of these matchmakers. We exclude vertices with very high degree from acting as matchmakers. We have also replaced our contraction scheme from our previous work with a fine-grained per-vertex hashing scheme for deduplication, as outlined in Algorithm 3.1.

3.2. Kokkos. We use Kokkos [45] to implement the parallel kernels in our code. Kokkos facilitates performance portability, allowing the programmer to maintain a single-source program that can be compiled for different shared-memory architectures. We compile for three different targets: NVIDIA GPUs using the CUDA backend, multicore CPUs using the OpenMP backend, and single threads of the same CPUs using a serial backend. The Kokkos programming model involves expressing a task as a sequence of small kernels that fit one of three parallel primitives: parallel-for, reduction, and scan.

4. Jet refinement algorithm. We have two design goals for refinement on the GPU: matching or exceeding the quality of multicore refinement techniques, and running time that is comparable to fast multicore refinement. Prior shared-memory multicore-centric refinement algorithms such as hill-scanning and MLS rely on thread-local priority queues. Priority queues are useful for finding sequences of moves that improve the cutsize where single moves cannot. However, these priority queue operations do not expose adequate concurrency for GPU-scale parallelism, and therefore such approaches are not viable on the GPU. Size-constrained LP-based iterations can visit the vertices in any order and therefore lend themselves naturally to both multicore and GPU parallelism. However, the size constraint limits the number of vertices that can be moved in each pass. This can be especially problematic if the distribution of beneficial moves is biased towards certain destination parts. To address this challenge, our method, Jet, splits a size-constrained LP iteration into two phases. The first phase is an unconstrained LP phase, Jetlp, that performs vertex moves while ignoring size constraints. The second phase is a rebalancing phase, Jetr, which has the task of moving vertices from oversized parts to nonoversized parts such that no oversized parts remain. It is paramount for the rebalancing phase to minimize any increase in cutsize (or loss). LP-based algorithms generally produce lower-quality results than FM-based methods, so we introduce novel augmentations to LP for improved quality. The overall structure of our refinement algorithm (Algorithm 4.1) is to apply Jetlp until any part becomes oversized and then apply Jetr until balance is restored. We denote each application of either Jetlp or Jetr as an "iteration." We record the best balanced partition in terms of cutsize and terminate refinement when we exceed a certain number of iterations (we use 12 for our results) without encountering a new best partition. We also use a tolerance factor ϕ to terminate when the cutsize is improving too slowly (see line 18). ϕ is the most important hyperparameter to control the quality/runtime tradeoff, where $\phi = 1$ gives the best quality. We use $\phi = 0.999$, which gives a good balance between quality and runtime.

4.1. Unconstrained label propagation: Jetlp. Our unconstrained label propagation is synchronous; i.e., updates to the partition state are deferred to the end of each iteration. The steps in Algorithm 4.2 are as follows: first, the algorithm selects a destination part $P_d(v)$ for each vertex v and records the gain $F(v)$ of making this move by itself. Second, it filters the vertices where $P_d(v)$ is different from the current part $P_s(v)$ and the gain $F(v)$ satisfies a constraint (inequality (4.3)); it pushes these vertices to an unordered list and assigns the gain as the priority value. Finally, it filters this unordered list using an approximation of the expected value of the next partition state. It determines this approximation in the neighborhood of each vertex that passed the first filter by merging P_s and P_d according to the priority values within each neighborhood. It commits all moves that pass the second filter and then updates the data structures that track connectivity of each vertex and the sizes of each part. The name Jet derives from a similarity in structure to a jet engine: the selection of destination parts is similar to the compressor, the first filter to the combustion chamber, and the second filter to the afterburner.

4.1.1. Changes to address LP limitations. A synchronous implementation of LP-based refinement has two limitations. First, it is not possible to improve cutsize through negative gain vertex moves. Second, vertex moves in the same iteration can affect each other detrimentally. We introduce a method to address both of these problems: the *vertex afterburner*. The vertex afterburner is a heuristic-based conflict resolution scheme permitting negative-gain vertex moves. We use the term afterburner

Algorithm 4.1 Jet Refinement Algorithm.

Input: The graph $G = (V, E)$ as defined in section 2.1. The number of parts k, balance factor λ. A partition array P_0.
Output: An output partition array P_{best}.

$P_{best} \leftarrow P_0$
$P_{iter} \leftarrow P_0$
$DS \leftarrow$ initDataStructures(G, P_0, k)
$R \leftarrow \emptyset$
while iteration limit not reached **do**
  if imb$(G, P_{iter}, k) < \lambda$ **then**
    $M \leftarrow$ Jetlp(G, P_{iter}, DS, R)
    $R \leftarrow$ vertexSet(ML)
    reset weak rebalance counter
  else
    if weak rebalance limit not reached **then**
      $M \leftarrow$ Jetrw$(G, P_{iter}, DS, k, \lambda)$
    else
      $M \leftarrow$ Jetrs$(G, P_{iter}, DS, k, \lambda)$
  $P_{iter}, DS \leftarrow$ updatePartsAndDS(G, P_{iter}, k, M, DS)
  if imb$(G, P_{iter}, k) < \lambda$ **then**
    if cost$(G, P_{iter}) <$ cost(G, P_{best}) **then**
      if cost$(G, P_{iter}) < \phi *$cost(G, P_{best}) **then**
        reset iteration counter
      $P_{best} \leftarrow P_{iter}$
  else if imb$(G, P_{iter}, k) <$ imb(G, P_{best}, k) **then**
    $P_{best} \leftarrow P_{iter}$
    reset iteration counter

as it is a secondary filter on the list of possible vertex moves; in typical LP-based refinement algorithms, there is only the first filter. Given a list of potential vertex moves X, we recompute the gain for each vertex in X according to an approximation of the next partition state in its neighborhood. This approximation is created by merging P_s with P_d, using an ordering *ord*. Due to the ordering *ord*, the approximations generated for overlapping neighborhoods are not consistent. P_d is fixed for all vertices in X prior to applying the afterburner; therefore recomputing the gain for each vertex $v \in X$ only involves the parts $P_d(v)$ and $P_s(v)$ specific to the move. For each neighbor u of a vertex $v \in X$, if $ord(u) < ord(v)$, we calculate v's gain assuming u will move to $P_d(u)$. Otherwise, we assume u remains in $P_s(u)$. This allows for vertex moves which initially had negative gain to become positive gain, and vice versa, depending on the other moves in X. The final move list M is chosen as a subset of X, containing only the moves in X with nonnegative gain after recalculation. Let $F(x) = \text{conn}(x, P_d(x)) - \text{conn}(x, P_s(x))$ be the priority values for each vertex move, given by the gain values of each vertex move in a vacuum. *ord* is defined as follows:

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

Algorithm 4.2 Jet - Label Propagation (Jetlp).

Input: The graph $G = (V, E)$. Partition array P_s. Data structures DS for querying vertex-part connection info. Locked vertices $R \subset V$. Filter ratio c.

Output: A list of moves M, in the form of vertex-destination part pairs.

```
1: P_d ← P_s
2: F ← negativeInfinity(|V|)
3: for v ∈ V \ R in parallel do
4:    A_v ← adjacentParts(v, DS) \ {P_s[v]}
5:    if A_v ≠ ∅ then
6:       P_d[v] ← argmax_{p∈A_v} conn(v, p, DS)
7:       F[v] ← conn(v, P_d[v], DS) − conn(v, P_s[v], DS)
8: X ← gainConnRatioFilter(V \ R, P_s, F, DS, c)      ▷ First filter (inequality (4.3))
9: F_2 ← zeros(|X|)
10: for v ∈ X in parallel do
11:    for (u, w) ∈ E[v] in parallel do
12:       p_u ← P_s[u]
13:       if ord(u) < ord(v) then
14:          p_u ← P_d[u]
15:       if p_u = P_d[v] then
16:          F_2[v] ← F_2[v] + w
17:       else if p_u = P_s[v] then
18:          F_2[v] ← F_2[v] − w
19: M ← nonNegativeGainFilter(X, P_d, F_2)            ▷ Second filter
```

4.1.2. Negative gain moves. The efficacy of this filter heuristic is sensitive to the composition of X. If X is selected too conservatively (i.e., only positive gain vertex moves), then afterburning does not produce an additional benefit over standard LP. If X is not constrained (i.e., the entire boundary vertex set), then afterburning will produce worse results than standard LP. To determine the composition of X, we must first determine $P_d(v)$ for each vertex v:

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

If a vertex is only connected to $P_s(v)$, it is not a boundary vertex and therefore is always excluded from X. The primary criterion for a vertex to be selected into X is as follows:

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

c is a constant that can be adjusted for different levels of the multilevel hierarchy. We find experimentally that $c = 0.25$ is most effective for the finest level of the hierarchy, whereas $c = 0.75$ is best for all other levels (for our partitioner). It is important to note the floor rounding, as our results on certain graphs are sensitive to the rounding direction. We find that the coarsening and initial partitioning algorithms affect the optimal choice for c.

4.1.3. Vertex locking. We employ an additional technique that is intended to help migrate the boundary in a coordinated fashion over successive iterations. This technique uses a lock bit, which excludes all vertices selected in M by an iteration of Jetlp from being chosen into X in the next iteration of Jetlp. Locking helps to

prevent oscillations, which occur when a vertex moves back and forth between two parts in successive Jetlp iterations. These oscillations may decrease solution quality by increasing the difficulty in changing the boundary's shape and location. Locks do not affect rebalancing iterations, nor does rebalancing change the lock state of any vertex.

4.2. Rebalancing: Jetr.

4.2.1. Two parts. We introduce rebalancing with a simpler version applicable only when $k = 2$. Without loss of generality, let p_a be the overweight part, and let p_b be the other part. The goal of our rebalancing is to move the vertices from p_a to p_b until p_a is no longer overweight, while minimizing the increase in the cutsize. We assign a simple loss value to every vertex in p_a: $\text{loss}(v) = \text{conn}(v, p_a) - \text{conn}(v, p_b)$. Loss can also refer to the combined loss of vertex sets: $\text{loss}(Z) = \sum_{v \in Z} \text{loss}(v)$. We order the vertices of p_a in terms of increasing loss in a list L. We then select the prefix L_x of L that minimizes the following expression:

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

It is expensive to use a sort to obtain L, so we approximate L with L', which is sorted according to a partial ordering. This partial ordering is derived from the following function of the loss value:

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

We found experimentally that the frequency of loss values tends to decrease as the absolute value of the loss value increases. We use $\log_2$ to assign slot values so that there are more slots closer to zero than far away from zero. This partial ordering is similar to a bucketing approach used to calculate the top k elements in a vector [5], but it only approximates the top k elements to save time. The insertion order within each bucket is subject to race conditions. To reduce the atomic contention on the GPU for the size counters of each bucket, we create ρ sub-buckets within each bucket that are keyed by $v \mod \rho$. This bucket-oriented approach also integrates well when computing lists for multiple overweight parts independently in our $k > 2$ variations.

THEOREM 4.1. *Let L'_x be the prefix of L' that minimizes expression* (4.4). *In a graph with uniform vertex weights, and assuming the number of vertices with negative loss is negligible, we have the following inequality:*

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

We now prove this theorem. $|L'_x| = |L_x|$ because all vertices have the same weight. Theorem 4.1 holds trivially if both L'_x and L_x are the empty set. Otherwise, let $s = \max_{v \in L'_x} \text{slot}(\text{loss}(v))$. Let S be the subset of p_a consisting of all vertices with a slot value less than or equal to s. L'_x is a subset of S by definition. S contains all the vertices in p_a with loss values smaller than a function of s; therefore it is a prefix of L. L_x must then be a subset of S, as $|S| \geq |L'_x| = |L_x|$. Similar logic shows that S', the subset of all vertices in p_a with slot values less than s, is a strict subset of both L_x and L'_x. We have shown that $L_x \triangle L'_x$ is a subset of $S \setminus S'$. $L_x \triangle L'_x$ only contains vertices with loss values equal to s, by the definition of $S \setminus S'$. $|L'_x \setminus L_x| = |L_x \setminus L'_x|$

because $|L'_x| = |L_x|$. Any two vertices with the same slot value have loss values within a multiple of 2 of each other; therefore the following inequality is true:

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

$\text{loss}(L_x \cap L'_x) \geq 0$, $\text{loss}(L'_x \setminus L_x) \geq 0$, and $\text{loss}(L_x \setminus L'_x) \geq 0$ due to our assumption that there are negligible negative loss vertices. We can add $\text{loss}(L_x \cap L'_x)$ to both sides to obtain inequality (4.6). This inequality also holds for our k-way formulations. The assumption for uniform vertex weights is necessary to ensure $|L_x| = |L'_x|$. If we have nonuniform vertex weights, inequality (4.6) no longer holds. In this case, the ratio between the total number of vertices in each set having slot value s (i.e., $|L_x \setminus S'|$ and $|L'_x \setminus S'|$) can be used to form a new inequality:

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

4.2.2. More than two parts. When $k > 2$, extending this rebalancing formulation is not trivial. We propose two separate extensions for arbitrary k that both reduce to the $k = 2$ formulation. Similar to label propagation, the output consists of an unordered list of vertices to move and their chosen destinations. Let B be the set of parts with size less than a value σ. σ determines the maximum size for a part to be considered a valid destination and is chosen such that there is a deadzone between the size of valid destination parts and the size of oversized parts. The first formulation uses the following definition of loss:

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

In this formulation (detailed in Algorithm 4.3), vertices are evicted from the oversized parts such that each oversized part is just smaller than the size limit (this should be within the deadzone). This process is similar to the formulation with $k = 2$, except that there are multiple oversized parts. Note that the multiple scans performed from line 21 to line 28 can be accomplished with just two scans, although we omit this detail from Algorithm 4.3 for brevity. The evicted vertices are sent to their best connected part among the valid destination parts. It is possible that the vertex is not connected to any valid destination part, in which case a random valid destination is chosen. In this formulation, it is possible for destination parts to become oversized. However, the deadzone prevents oversized parts from becoming valid destinations. This guarantees at most k iterations to achieve a balanced partition as at least one part will move into the deadzone in each iteration if the vertex weights are uniform. We observe that the typical number of iterations required is substantially less than k. We denote this extension as weak rebalancing (Jetrw) due to the potential need for many iterations. Let A_v be the adjacent parts of vertex v. Our second extension uses the following definition of loss:

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

Vertices are evicted from oversized parts in the same manner as the prior formulation. The destination parts then try to acquire as close to $\sigma - |B|$ vertices from the evicted set as possible. Given that the evicted vertices are arranged in an unordered list, each destination partition selects a contiguous group from this list. Destination partitions are overlayed onto the unordered list according to their capacity, forming a one-dimensional "cookie-cutter" pattern. This formulation guarantees that no oversized

parts remain after a single iteration if vertex weights are unit. We observe that vertex weights are often a significant fraction of the size constraint when more than one iteration is necessary. We denote this extension as strong rebalancing (Jetrs) due to its ability to achieve balance in one iteration in most scenarios.

Jetrw is much more effective at minimizing loss than Jetrs, even though it may require more iterations to converge upon a balanced partition. Our observations indicate that Jetrs requires fewer iterations to converge in any of the following scenarios: regular graphs, small values of k, and large imbalance ratios. We propose a combination of the two formulations, where we apply Jetrw for a certain number of iterations (denoted as b_{max} in Algorithm 4.1), and then apply Jetrs if the partition is still unbalanced. We find that even a single iteration of Jetrw followed by an iteration of Jetrs can achieve much of the benefit of an unlimited number of iterations of Jetrw. Our full rebalancing (Jetr) consists of two iterations of Jetrw followed by a single iteration of Jetrs. If more iterations are necessary due to large vertex weights, these are performed with Jetrs. For both rebalancing variants, we find it beneficial to restrict a vertex from leaving an oversized partition if its respective vertex weight is greater than $1.5(|p_a| - \frac{|V|}{k})$. This restriction is applied before we construct L'.

4.3. Data structures and optimization. We represent our input graphs and coarse graphs in-memory using the compressed-sparse-row (abbreviated as CSR or CRS) format. We require a data structure to track connectivity of each vertex to each partition in order to facilitate Jet's iterations. Our label propagation iterations must be able to quickly identify the first and second most connected parts for each vertex. Our weak rebalancing iteration must identify the most connected valid destination part for each vertex in an oversized part. Our strong rebalancing iteration must sum the connectivity among valid destination parts for each vertex in an oversized partition. Finally, it should be possible to modify this data structure given a list of vertices to move. A naive implementation might use $|V| * k$ space to explicitly track this connectivity data for each possible pair of a vertex and part. Unfortunately, this uses far too much space with otherwise reasonable values for k and is inefficient to traverse in all use cases. Our implementation is based on the observation that for any vertex v, the number of partitions to which it can have nonzero connectivity is at most $\min(k, \text{degree}(v))$. We utilize a formulation similar to the CSR graph format to represent the vertex-part connectivity matrix. Our data structure allocates space equal to the following expression:

$$2|V| + 1 + 2\sum_{v \in V} \min(k, \text{degree}(v)). \tag{4.11}$$

Each row in this CSR representation is treated as a hashtable (keyed on the partition id) for creation and updates. To determine the most connected parts that satisfy some filter criteria relative to each use case, we linearly search the hashtables. This linear search is substantially more efficient for smaller hashtables, so we limit the number of empty entries. Although $min(k, degree(v))$ is the maximum possible part connections for each vertex, we observe that many graphs (particularly regular graphs but even many irregular graphs) have a much smaller number of nonzero connections in practice. For instance, it is possible for a degree 100 vertex with $k = 128$ to only have one or two nonzero part connections. We set the hashtable size to be slightly larger than the initial connectivity upon construction. This may cause insertions into the hashtable to fail once this limited capacity is reached. When this occurs, we expand the hashtable capacity and recalculate its contents. We assign a small amount of extra space to each hashtable to limit the frequency for which this is necessary.

Algorithm 4.3 Jet - Weak Rebalancing.

Input: The graph $G = (V, E)$. An unbalanced partition array P_s. Data structures DS for querying vertex-part connection info. k. λ. Minibucket count ρ.
Output: A list of moves M, in the form of pairs of vertex-destination parts.

$P_d \leftarrow P_s$
$o \leftarrow (1 + \lambda)|V|/k$
$\sigma \leftarrow \text{maxDestSize}(o)$
$A \leftarrow \{p \mid p \in P \wedge |p| > o\}$
$B \leftarrow \{p \mid p \in P \wedge |p| \leq \sigma\}$
$F \leftarrow \text{zeros}(|P_d|)$
for $v \in V$ in parallel **do**
  if $P_s[v] \in A$ and $\text{vtxWgt}(v) < \text{limit}(P_s[v], |V|, k)$ **then**
    $A_v \leftarrow \text{adjacentParts}(v, DS) \cap B$
    $P_d[v] \leftarrow \text{argmax}_{p \in A_v} \text{conn}(v, p)$
    if $A_v = \emptyset$ **then**
      $P_d[v] \leftarrow \text{randomPart}(B)$
    $F[v] \leftarrow \text{conn}(v, P_s[v]) - \text{conn}(v, P_d[v])$
$L' \leftarrow \text{buckets}(|A|)$
for $v \in V$ in parallel **do**
  if $P_s[v] \in A$ **then**
    $s \leftarrow \text{slot}(F[v])$
    $\text{writeToBucket}(L'[P_s[v]][s][v \bmod \rho], v)$
$t \leftarrow 0$
$M \leftarrow \text{emptyList}$
for $p_s \in A$ in parallel **do**
  $m \leftarrow 0$
  $m_{max} \leftarrow |p_s| - o$
  for $v \in L'[p_s]$ parallel scan on m **do**
    $m \leftarrow m + \text{vtxWgt}(v)$
    if $m < m_{max}$ **then**
      $M[t] \leftarrow (v, P_d[v])$
      $t \leftarrow t + 1$

In order to update this data structure once the vertex move list M is chosen, we update in two passes (see Algorithm 4.4). The first pass decrements the part connectivity of every neighbor of each vertex in M for the respective source partitions and creates an open entry in place of any part that reaches a connectivity of zero. The second pass increments the part connectivity of every neighbor of each vertex in M for the respective destination partitions, potentially creating new entries in the hashtable when necessary. The creation of new entries in the second pass may fail for some rows if the current hashtable size for that row is insufficient. We mark the respective rows and then recalculate the corresponding hashtables in a third pass (not shown in Algorithm 4.4). All passes leverage atomic operations to ensure correctness, but a race condition affects which entry in the hashtable any given part will be assigned to. This race condition also exists for the initial construction of the data structure. In the Jetlp and Jetrw phases, this can affect how ties are broken when determining the most connected part for a vertex. Together with the race condition for bucket insertions in Jetrw and Jetrs, these are the only sources of nondeterminism in the Jet

Algorithm 4.4 Jet - Update Part Connectivities.

Input: The graph $G = (V, E)$. A partition array P_s. Data structures DS for querying vertex-part connection info and lock status. A list of vertex moves M.
Output: Updated Data structures DS

```
1: for v ∈ vertexSet(M) in parallel do
2:    p_s ← P_s[v]
3:    for (u, w) ∈ E[v] in parallel do
4:       h ← getHashmap(DS, u)
5:       h[p_s] ← h[p_s] - w
6:       if h[p_s] = 0 then
7:          setOpen(h, p_s)
8: for (v, p_d) ∈ M in parallel do
9:    for (u, w) ∈ E[v] in parallel do
10:      h ← getHashmap(DS, u)
11:      if p_d ∉ h then
12:         insert(h, p_d)
13:      h[p_d] ← h[p_d] + w
```

refinement algorithm. Algorithm 4.4 has the benefit of only updating rows adjacent to the vertex moves, and only the entries specifically affected in those rows. We also implement an alternative update algorithm, which we use when the number of vertex moves constitutes more than 10% of the total vertices in the graph. In this alternative algorithm, we reconstruct the entire hashtable for every row adjacent to a moved vertex from the new partition state after applying each move. This reduces the irregularity of memory accesses over Algorithm 4.4, at the cost of more work.

5. Experimental setup. Our experiments evaluate the performance of our partitioner in terms of both cutsize and overall execution time. We compare our GPU partitioner to other state-of-the-art multicore multilevel partitioners including Mt-Metis v0.7.2 with Hill-scanning, mt-KaHIP v1.00 with MLS, KaMinPar v1.0, and Mt-KaHyPar-D v1.3.2, as well as the serial partitioner Metis v5.1.0. We choose to compare to Mt-Metis with Hill-Scanning enabled and to mt-KaHIP with MLS enabled because these are the highest quality refinement options available for their respective partitioners. We utilize the default configuration of Mt-KaHyPar as this is the highest quality configuration to use KFM refinement in v1.3.2. We are unable to compare with either of the other GPU partitioners [16, 21], as their code is not available. In the later work [21], their cutsize results were slightly worse than both Metis and Mt-Metis (without Hill-scanning) on all graphs tested. We evaluate on $k = 32$, $k = 64$, $k = 128$, and $k = 256$ with the imbalance set to 3%, as well as $k = 128$ with imbalance set to 1% and 10%. This constitutes a total of six experiments per graph and partitioner. Although most of these partitioners can operate on arbitrary values of k, mt-KaHIP cannot; therefore, our experiments are on k values that are powers of 2. For each combination of graph, experiment, and partitioner, we collect the median cutsize and median runtime across a number of runs. The number of runs performed is dependent on the partitioner: we perform five runs for mt-KaHIP MLS, 11 runs for KaMinPar, Mt-Metis HS, and Mt-KaHyPar-D, three runs for Metis, and 21 runs for our partitioner. The trials per partitioner are approximately inversely proportional to their respective runtimes. We present breakdowns versus each opposing partitioner by experiment configuration and in terms of graph classification.

TABLE 1

We compare the Jet partitioner to various partitioners, reporting the ratios of the geometric mean of median cutsizes obtained with the partitioner to the geometric mean of the median cutsizes with the Jet partitioner. A value greater than 1 *indicates that the Jet partitioner performs better. The number of parts and the balance constraint setting are varied.*

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

5.1. Effectiveness tests. In order to determine the effectiveness of our refinement method, we compare directly with mt-KaHIP's MLS and Mt-KaHyPar-D's KFM. As these are the highest quality partitioners to which we compare, these serve as ideal benchmarks for Jet refinement. In order to isolate refinement as the only variable, we export the coarse graph hierarchy and initial partitioning from the opposing partitioner and import it into our program. We then refine the solution on the imported hierarchy using Jet refinement. For each run, we compute the ratio of the final cutsize result obtained by the two refinement methods. We gather the median ratio out of a number of test runs for each graph (5 versus MLS; 11 versus KFM). We do the same for the refinement time. We also perform the reverse of this experiment, exporting our coarse graphs and initial partitioning into the opposing partitioner. To ensure a fair comparison, we exclude coarse graphs that have an imbalanced partition prior to refinement and export the partitioning for the coarsest graph with a balanced partition. This is necessary because MLS assumes a balanced input partition. For these experiments, we use $k=64$ and the imbalance equals 3%, and we use our CPU platform to produce a fair refinement time comparison.

5.1.1. Runtime. We present a breakdown of our runtimes into three categories: coarsening, initial partitioning, and refinement. We analyze the runtime scaling versus k and the imbalance. We analyze the runtime scaling versus graph size using several graph families. We present multicore scaling numbers and GPU versus CPU performance.

5.2. Test graphs. Our test set (see Table SM1 of the supplementary material) contains all graphs with at least 50 million nonzeros but less than 750 million nonzeros from the Suitesparse graph repository [14] (excluding mawi graphs). We also include a few miscellaneous graphs (ppa, citation, products) from Open Graph Benchmark [28] and some social networks (dblp10, amazon08, hollywood11, enwiki21) published by the Laboratory for Web Algorithmics [8, 9] and one graph (fe_rotor) from the Walshaw Graph Benchmark [43]. We also add a 2000x4000 rectangular mesh (grid) and a 200x200x200 cubic mesh (cube). We preprocess all graphs by performing the following steps: we remove self-loops, convert all directed edges to undirected edges, remove duplicate edges, and extract the largest connected component. The graphs are further grouped into one of nine classes.

5.3. Test systems. We conduct our tests on two different systems. Our first system runs on a 32-core Ryzen 3970x Threadripper, with 256 GB of RAM (quad-channel DDR4). The first system runs the experiments for the competing partitioners, as well as the serial and multicore experiments for our partitioner's scal-

ing results. Our MLS versus Jet refinement effectiveness test also runs on this system. We run each partitioner using 64 threads. The second system is a virtual machine with 12 virtual cores of an Intel Xeon Gold 6342 CPU, 90 GB RAM, and an Nvidia A100 GPU with 80 GB of VRAM. The second system runs the primary experiments for our partitioner, and the Jetlp component effectiveness experiments. Both systems run Ubuntu 20.04. Our code is compiled with NVCC using Cuda Toolkit version 11.6.2 for the A100 platform, and g++ version 10.2.0 on the ThreadRipper 3970x platform. We use release versions 3.6.1 of both Kokkos and Kokkos-Kernels libraries, and Metis library version 5.1.0.

6. Partitioner performance evaluation.

6.1. Quality. Our GPU partitioner outperforms Mt-Metis with Hill-scanning (HS), KaMinPar, and Metis in cutsize. As shown in Figure 1, our partitioner is

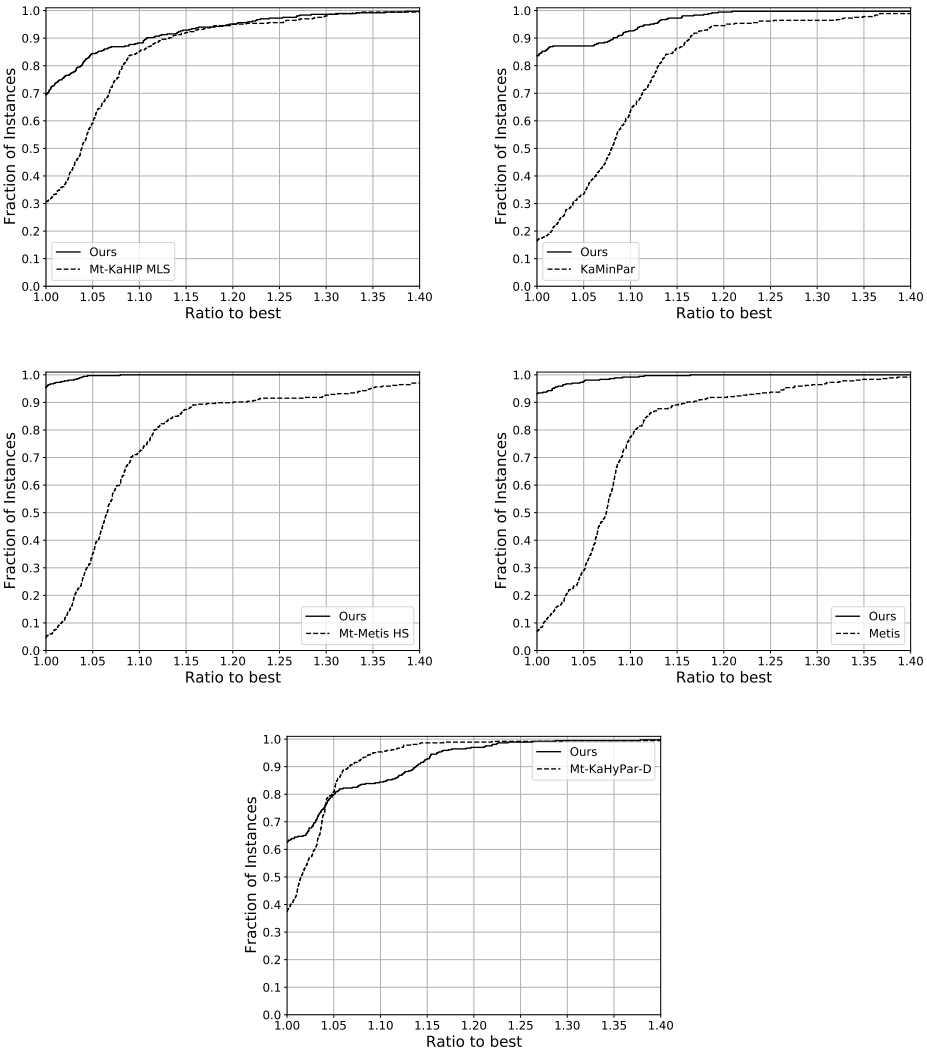


FIG. 1. *We use performance profiles to compare cutsize obtained using our partitioner to others.*

better on more than 90% of test instances than Mt-Metis HS and Metis, more than 80% of instances versus KaMinPar, about 70% of instances versus mt-KaHIP MLS, and more than 60% of instances versus Mt-KaHyPar-D.

6.1.1. Experiment configs. In Table 1, we note that our cuts are more than 6% better than Mt-Metis HS, KaMinPar, and Metis in cutsize across all experiment configurations except one. The outlier is the $k = 256$ experiment, where ours is 4.9% better than KaMinPar. Mt-KaHIP MLS produces cuts around 2.5% worse than ours overall. The only competitor to outperform ours is Mt-KaHyPar-D, which achieves 0.5% better cuts overall. We note that our cutsize performance at $k = 128$ and imbalance of 10% is relatively better versus each other partitioner than the $k = 128$ and imbalance of 3% configuration. At $k = 128$ and an imbalance of 1%, we are relatively better versus mt-KaHIP MLS and KaMinPar than in the respective 3% configuration, whereas we are relatively worse versus Mt-Metis HS and Metis. We do relatively worse versus KaMinPar, Mt-KaHyPar-D, Mt-Metis HS, and Metis with increasing values of k.

6.1.2. Graph classes. We classify our partitioner's strengths and weaknesses by graph type using Figures 2a and 2b. Our partitioner is dominant on finite element problems, optimization problems, social networks, semiconductor problems, and artificial complex networks. Of the social networks, our partitioner only failed to produce the best cut on all amazon08 and dblp10 instances, which are the two smallest social networks in our test set, and one instance for com-Orkut. We have a moderate strength on biology graphs, with ours obtaining the best cuts for most ppa and cage15 instances, and Mt-KaHyPar-D obtaining the best cuts on most of the kmer graph instances. Our weaknesses include the artificial meshes, web crawls, and road networks. Excluding the web crawls, most of the graphs in these classes have an

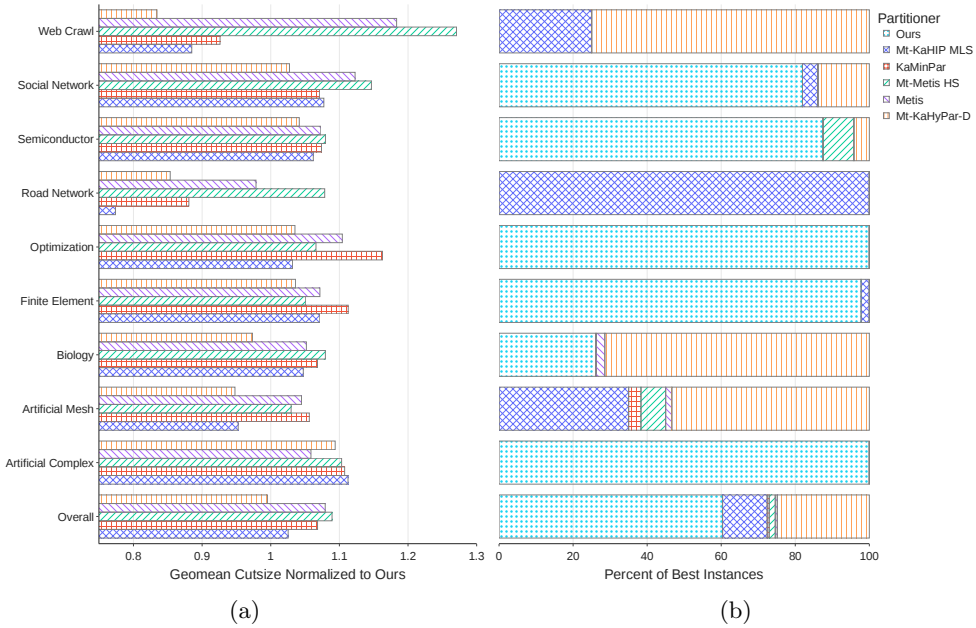


FIG. 2. *Cutsize results by class.*

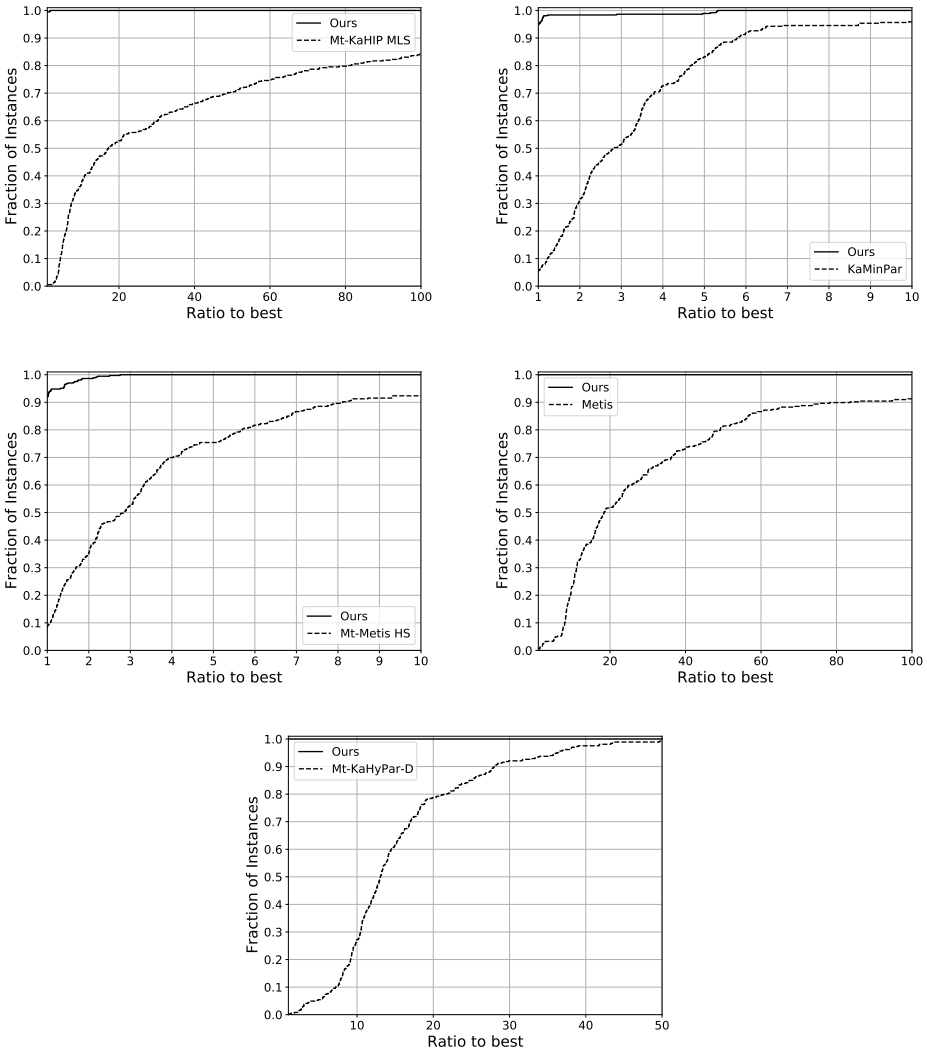
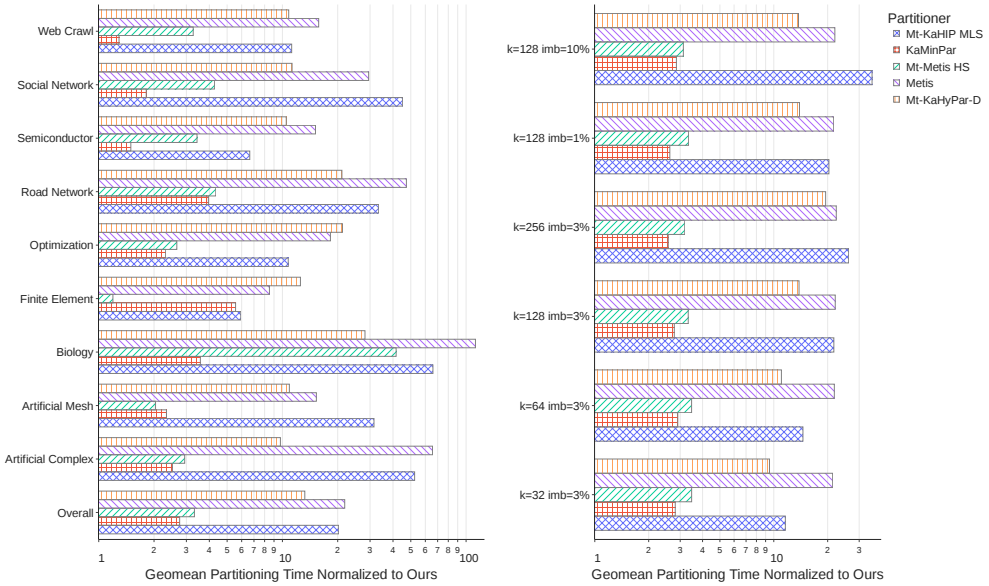


FIG. 3. *We use performance profiles to compare partitioning time of the Jet partitioner (ours) on the A*100 *GPU to the execution time of other partitioners. The other partitioners are executed on the AMD Ryzen Threadripper* 3970*x CPU.*

underlying 2D structure. We explore the reason for our poor performance on the web crawls and 2D problems in the effectiveness test section.

6.2. Runtime. Our GPU partitioner is consistently faster than our CPU competitors, with shorter runtimes than any competitor in more than 85% of the test instances. We found that our partitioner was faster than mt-KaHIP MLS on more than 99% of the test instances (see Figure 3), and more than twenty times faster on more than 40% of the test instances. Compared to KaMinPar, Figure 3 shows that ours is faster in more than 90% of the instances and at least twice as fast in more than 65% of the instances. Compared to Mt-Metis HS, our runtime was better in more than 90% of the test instances and at least twice as fast in more than 60% of the instances. Our partitioner is faster than Metis by more than 20x in over 40% of the test instances, similar to mt-KaHIP MLS. Our partitioner is faster than Mt-KaHyPar-D



(a) By Graph Class. (b) By Experiment.

FIG. 4. *Partitioning time comparison.*

by at least 10x in over 70% of the test instances. The runtime performance of mt-KaHIP is surprising; in Figure 4b, it only achieves superior geometric mean runtimes to Metis on our $k = 32$ and $k = 64$ experiments. In the same figure, our partitioner shows similar performance trends across experiment configurations. In Figure 4a, our partitioner achieves strong runtime results on artificial complex, biology, road network, semiconductor, and social network graphs. Our partitioner achieves similar runtime performance to Mt-Metis HS on the finite element graphs, and it also achieves similar performance to KaMinPar on the web crawl graphs. KaMinPar achieves much better performance than our partitioner on the circuit5M graph.

6.2.1. Time breakdown. In Table 2, we present the average time spent in each subtask of partitioning by our graph partitioner on the GPU. We organize this data by graph class and include the average total runtime among all experiments in that graph class. Across all classes, initial partitioning is responsible for at most 10.6% of the total runtime. Coarsening dominates on artificial complex, semiconductor, and web crawl graphs, whereas uncoarsening dominates on the other graphs. Coarsening tends to dominate on most graphs where the degree distributions are extremely irregular, whereas uncoarsening dominates on the more regular graphs.

7. Refinement performance evaluation.

7.1. Effectiveness test. In our cut effectiveness test (Table 4), we directly evaluate the performance of our Jet refinement algorithm. In section 6.1.2 we found that our partitioner produces superior cuts on finite element problems, social networks, artificial complex networks, semiconductor problems, optimization problems, and some biology problems. Conversely, we found it to produce inferior cuts on problems derived from a 2D structure or from web crawls. Additionally, Mt-KaHyPar-D achieved superior cuts on the kmer graphs. We now detail how the refinement algorithm affects the partitioning results.

TABLE 2
Partitioning time breakdown by subtask and average total time (seconds).

Graph class	Uncoarsen (%)	Coarsen (%)	InitPart (%)	Avg. time (s)
Web Crawl	30.3	64.6	5.1	1.38
Social Network	58.2	31.2	10.6	1.11
Semiconductor	38.9	56.2	4.9	1.63
Road Network	52.6	43.8	3.6	0.43
Optimization	50.4	45.3	4.3	0.63
Finite Element	55.4	34.9	9.7	0.27
Biology	63.9	28.3	7.9	1.72
Artificial Mesh	64.6	31.2	4.2	0.41
Artificial Complex	36.9	54.7	8.4	1.83

TABLE 3
Geomean(baseline cutsize)/geomean(version cutsize).

Baseline + Locks	Baseline + Weak Afterburner	Baseline + Full Afterburner	Full Jetlp
1.000	1.009	1.030	1.052

TABLE 4
Refinement effectiveness: Geomean of median ratio (their cut/our cut).

Graph class	MLS vs Jet Mt-KaHIP backend	MLS vs Jet Our backend	K-way FM vs Jet Mt-KaHyPar-D backend	K-way FM vs Jet Our backend
All	1.019	1.062	1.007	1.034
Web Crawl	1.019	1.100	1.004	1.043
Social Network	1.082	1.173	1.040	1.079
Semiconductor	0.998	1.063	1.006	1.029
Road Network	0.905	0.901	0.972	1.024
Optimization	0.985	1.013	0.998	1.010
Finite Element	1.019	1.037	1.000	1.021
Biology	1.034	1.059	0.995	1.023
Artificial Mesh	0.934	0.977	0.951	0.986
Artificial Complex	1.148	1.167	1.132	1.122

7.1.1. Strengths. Overall, Jet produces 1.9% better cuts than MLS when using the Mt-KaHIP coarsening and initial partitioning, and 6.2% better cuts using our coarsening and initial partitioning. Compared to KFM, these numbers are 0.7% with the Mt-KaHyPar-D backend and 3.4% with our backend. Our refinement algorithm produces superior cuts for finite element, biology (excluding kmer graphs), social network, web crawl, and artificial complex networks. This rules out our refinement algorithm as the cause for our partitioner's weakness on web crawls, leaving our coarsening and initial partitioning as potential culprits. The cutsize result for semiconductor and optimization problems depends on the backend. Our backend produces the overall better cuts for most of these graphs, and Jet is superior in this case. Regarding kmer graphs, we found that Jet produces about 0.8% better cuts than MLS and 1.4% worse cuts than KFM combined across both backends. Strangely, Jet outperforms KFM on the road networks using our backend. However, our backend generates worse overall cuts for road networks than Mt-KaHIP or Mt-KaHyPar-D.

7.1.2. Weaknesses. For problems with a 2D structure, including most artificial meshes and road networks, Jet demonstrates a refinement capability inferior to that of MLS and KFM. We speculate that this weakness is related to certain types of improvement that are difficult for our algorithm to identify. Improvements that

TABLE 5
Refinement effectiveness: Geomean of median ratio (their uncoarsening time/uur uncoarsening time).

Graph class	MLS vs Jet Mt-KaHIP backend	MLS vs Jet Our backend	K-way FM vs Jet Mt-KaHyPar-D backend	K-way FM vs Jet Our backend
All	2.418	2.498	3.005	3.328
Web Crawl	1.713	2.457	3.455	4.806
Social Network	2.827	2.188	2.395	2.197
Semiconductor	0.997	1.291	2.684	3.372
Road Network	3.083	3.271	4.787	5.961
Optimization	2.165	2.114	3.868	4.806
Finite Element	2.109	2.214	3.768	4.424
Biology	5.335	6.379	3.562	3.892
Artificial Mesh	1.759	1.519	2.903	3.005
Artificial Complex	0.867	1.291	1.314	1.516

substantially change the location of the boundary are difficult to find as our label propagation phase can only consider vertices that currently lie on the boundary within each iteration. This problem is exacerbated by graphs with large diameters, such as 2D meshes. Road networks and artificial meshes (except the cubic mesh) have an underlying 2D structure and, therefore, have graph diameters of $O(\sqrt{|V|})$. The kmer graphs also have large graph diameters. For a concrete example, consider our grid graph, which has a graph diameter of 5998: the MLS-to-Jet ratio is 0.902 and 0.906 for the mt-KaHIP backend and our backend, respectively. The cubic graph for comparison has a smaller graph diameter of 597 (a consequence of being a 3D mesh), and the MLS-to-Jet ratios are 0.965 and 1.008, respectively. MLS and KFM have the capability to find improvements that substantially shift the boundary, as they can perform long sequences of moves in localized regions of a graph. HS lacks this ability, due to the cap on hill-size.

7.1.3. Uncoarsening time. In Table 5, Jet is faster than MLS for both backends by factors greater than 2.4x, and it is faster than KFM for both backends by at least 3x. Jet achieves these speedups consistently across most graph classes, except semiconductor graphs versus MLS and artificial complex graph versus both competitors. We attribute Jet's superior uncoarsening speed to its bulk-synchronous design, efficient datastructures for tracking vertex-part connectivity, and the absence of priority queues. Although we have used the CPU platform for our code in order to obtain a fair comparison in this experiment, we note that a GPU implementation of either MLS or KFM is nontrivial.

7.1.4. Component effectiveness. In Table 3, we evaluate the impact of design choices in our Jetlp phase compared to a baseline synchronous LP. Our baseline only moves vertices into their best connected partition, omits the afterburner kernel in its entirety, and ignores the lock bit. We compare four versions of the LP phase. The first version is the baseline plus vertex locking. The second version is the baseline plus a weaker version of the afterburner, which only considers vertex moves with positive or zero gain. The third version is the baseline plus the full afterburner; that is, it considers negative gain vertices as described in section 4.1.2. The fourth version is the full Jetlp algorithm, that is, the baseline plus vertex locking plus the full afterburner. The results in Table 3 show that the afterburner performs substantially better if it can consider negative gain vertex moves than when it does not. Interestingly, the vertex lock does not provide any benefit alone, but combined with the full afterburner it

TABLE 6
Uncoarsening speedup, overall, and finest level refinement.

	A100 vs 3970x 32-core		3970x 32-core vs serial	
Graph class	Overall	Finest level	Overall	Finest level
Web Crawl	5.14×	7.99×	9.28×	9.72×
Social Network	7.71	9.94	12.45	12.82
Semiconductor	6.04	8.18	7.83	7.77
Road Network	6.53	14.77	7.95	8.46
Optimization	5.48	9.04	13.04	10.44
Finite Element	3.50	6.27	10.72	12.74
Biology	9.22	13.14	14.04	11.71
Artificial Mesh	4.33	8.96	9.72	10.61
Artificial Complex	8.38	12.00	16.12	16.04

provides a benefit of 2.2% versus the full afterburner without the locks. Full Jetlp provides a cutsize benefit over the baseline that varies from a negligible 0.1% for artificial complex graphs to a substantial 11.8% for artificial meshes. Furthermore, we investigate the impact of ϕ on the execution time and cutsize results of our final version. We found that decreasing our refinement tolerance value ϕ to 0.99 improves the uncoarsening time by 55% and worsens the cutsize by 1.1% over our default value of 0.999. Increasing ϕ to 0.9999 worsens the uncoarsening time by 34% and improves the cutsize by 0.5% over the default value. The cutsize benefit of increasing ϕ is most pronounced for artificial meshes and least pronounced for artificial meshes and web crawls.

7.2. Parallel scaling. We continue the performance analysis by analyzing the relative performance of our test systems. In Table 6, we compare the performance of our multicore AMD Ryzen 3970x CPU system to our Nvidia A100 GPU system, and also compare multicore performance to serial performance on the CPU. We include results for total uncoarsening time, as well as refinement time for just the finest level graph. The 32-core uncoarsening speedup is between 7.8x and 16.1x, which is not ideal on the lower end. The finest level refinement speedup is within a similar range of 7.8x to 16x. The suboptimal 32-core speedup is partly due to memory-bandwidth constraints, as well as certain implementation choices made for GPU performance that are not as effective for CPU platforms. The GPU versus CPU uncoarsening speedup is between 3.5x and 9.2x. However, the GPU versus CPU finest level refinement speedup is better, landing between 6.3x and 14.8x. This is most likely due to the host-device synchronization time, which represents a larger portion of the refinement time on the smaller coarse graphs.

8. Conclusion. We demonstrate a partitioner that leverages GPU acceleration to decrease partitioning time while delivering state-of-the-art partition quality. Our partitioner demonstrates superior quality on five of nine graph classes in our test set compared to several state-of-the-art partitioners, across six experiment configurations. Our runtimes are superior on all nine graph classes. We attribute these results to our novel partition refinement algorithm, Jet. Jet builds on label propagation by addressing many common drawbacks while optimizing for GPU scalability. Jet delivers cutsizes similar to or better than two state-of-the-art parallel implementations of FM refinement on six out of nine graph classes, and superior runtime on seven out of nine graph classes. We identify quality on two-dimensional mesh-like graphs as the primary weakness of Jet, which is consistent with other label propagation algorithms.

Our partitioner is able to substantially reduce the time spent for initial partitioning by coarsening to extremely small graphs. We plan to investigate methods to enhance Jet's quality and to demonstrate Jet in a distributed memory partitioner.

Reproducibility of computational results. This paper has been awarded the "SIAM Reproducibility Badge: Code and data available", as a recognition that the authors have followed reproducibility principles valued by SISC and the scientific computing community. Code and data that allow readers to reproduce the results in this paper are available in https://scholarsphere.psu.edu/resources/cc9dcf42-f5eb-42f1-80ec-5d50a402fc22 as well as in the supplementary material files Jet_Multilevel_Graph_Partitioning_on_Graphics_Processing_Units_Supplementary_Material.pdf [local/web 267KB and Jet-Partitioner.zip [local/web 47.9KB].

REFERENCES

[1] S. Acer, E. G. Boman, C. A. Glusa, and S. Rajamanickam, *Sphynx: A parallel multi-GPU graph partitioner for distributed-memory systems*, Parallel Comput., 106 (2021), 102769, 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*, in Proceedings of the International Parallel and Distributed Proc. Symp. Workshops (IPDPSW), 2020.

[3] Y. Akhremtsev, P. Sanders, and C. Schulz, *High-quality shared-memory graph partitioning*, IEEE Trans. Parall. Distrib. Syst., 31 (2020), pp. 2710–2722.

[4] T. A. Akyildiz, A. A. Aljundi, and K. Kaya, *GOSH: Embedding big graphs on small hardware*, in Proceedings of the International Conference on Parallel Processing (ICPP), 2020.

[5] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, *Fast k-selection algorithms for graphics processing units*, ACM J. Exp. Algorithmics, 17 (2012), 4.2, https://doi.org/10.1145/2133803.2345676.

[6] S. T. Barnard and H. D. Simon, *Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, 6 (1994), pp. 101–117.

[7] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP*, Oxford University Press, 2020.

[8] P. Boldi, M. Rosa, M. Santini, and S. Vigna, *Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks*, in Proceedings of the 20th International Conference on World Wide Web (WWW), 2011.

[9] P. Boldi and S. Vigna, *The WebGraph framework* I*: Compression techniques*, in Proceedings of the 13th International Conference on World Wide Web (WWW), New York, 2004, pp. 595–601.

[10] A. Brandt, *Algebraic multigrid theory: The symmetric case*, Appl. Math. Comput., 19 (1986), pp. 23–56.

[11] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent advances in graph partitioning*, in Algorithm Engineering, Springer, Cham, 2016, pp. 117–158.

[12] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, *HARP: Hierarchical representation learning for networks*, in Proceedings of the AAAI Conference, 2018.

[13] T. A. Davis, W. W. Hager, S. P. Kolodziej, and S. N. Yeralan, *Algorithm* 1003*: Mongoose, a graph coarsening and partitioning library*, ACM Trans. Math. Software, 46 (2020), 7.

[14] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), 1.

[15] I. S. Dhillon, Y. Guan, and B. Kulis, *Weighted graph cuts without eigenvectors: A multilevel approach*, IEEE Trans. Pattern Anal. Mach. Intell., 29 (2007), pp. 1944–1957.

[16] B. Fagginger Auer, *GPU Acceleration of Graph Matching, Clustering, and Partitioning*, Ph.D. thesis, Utrecht University, 2013.

[17] B. Fagginger Auer and R. H. Bisseling, *Graph coarsening and clustering on the GPU*, Graph Part. and Graph Clustering, 588 (2012), p. 223.

[18] C. M. Fiduccia and R. M. Mattheysis, *Linear-time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference (DAC), 1982, pp. 175–181.

[19] M. S. Gilbert, S. Acer, E. G. Boman, K. Madduri, and S. Rajamanickam, *Performance-portable graph coarsening for efficient multilevel graph analysis*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021.

[20] Y. Goldschmidt, M. Galun, E. Sharon, R. Basri, and A. Brandt, *Fast Multilevel Clustering*, Technical report, Weizmann Institute of Science, 2005.

[21] B. Goodarzi, F. Khorasani, V. Sarkar, and D. Goswami, *High performance multilevel graph partitioning on GPU*, in 2019 International Conference on High Performance Computing & Simulation (HPCS), 2019, pp. 769–778, https://doi.org/10.1109/HPCS48598.2019.9188120.

[22] L. Gottesbüren, T. Heuer, and P. Sanders, *Parallel flow-based hypergraph partitioning*, in Proceedings of the 20th International Symposium on Experimental Algorithms (SEA), Leibniz International Proceedings in Informatics (LIPIcs) 233, C. Schulz and B. Uçar, eds., Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, 5.

[23] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, *Deep multilevel graph partitioning*, in Proceedings of the 29th Annual European Symposium on Algorithms (ESA), LIPIcs 204, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 48.

[24] L. Gottesbüren, T. Heuer, N. Maas, P. Sanders, and S. Schlag, *Scalable high-quality hypergraph partitioning*, ACM Trans. Algorithms, 20 (2024), pp. 1–54, https://doi.org/10.1145/3626527.

[25] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag, *Scalable shared-memory hypergraph partitioning*, in Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), 2021, pp. 16–30.

[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*, in Domain Decomposition Methods in Science and Engineering XXV, Springer, 2020, pp. 176–184.

[27] B. Hendrickson and R. Leland, *A multi-level algorithm for partitioning graphs*, in SC Conference, Los Alamitos, CA, IEEE Computer Society, 1995, 3, https://doi.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*, in Proceedings of the Annual Conference on Neural Inf. Proc. Systems, 2020.

[29] Y. Hu and L. Shi, *Visualizing large graphs*, WIREs Comput. Statist., 7 (2015), pp. 115–136.

[30] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[31] Y. Koren, L. Carmel, and D. Harel, *Drawing huge graphs by algebraic multigrid optimization*, Multiscale Model. Simul., 1 (2003), pp. 645–673, https://doi.org/10.1137/S154034590241370X.

[32] D. LaSalle and G. Karypis, *Multi-threaded graph partitioning*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2013.

[33] D. LaSalle and G. Karypis, *A parallel hill-climbing algorithm for graph partitioning*, in Proceedings of the International Conference on Parallel Processing (ICPP), 2016.

[34] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis, *Improving graph partitioning for modern graphs and architectures*, in Proceedings of the Workshop on Irregular Applications: Architectures and Algorithms (IA3), 2015.

[35] A. Lenharth, D. Nguyen, and K. Pingali, *Parallel graph analytics*, Commun. ACM, 59 (2016), pp. 78–87.

[36] A. Pothen, *Graph partitioning algorithms with applications to scientific computing*, in Parallel Numerical Algorithms, Springer, 1997, pp. 323–368.

[37] U. N. Raghavan, R. Albert, and S. Kumara, *Near linear time algorithm to detect community structures in large-scale networks*, Phys. Rev. E, 76 (2007), 036106.

[38] I. Safro, P. Sanders, and C. Schulz, *Advanced coarsening schemes for graph partitioning*, ACM J. Exp. Algorithmics, 19 (2014), 2.2.

[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*, Commun. ACM, 64 (2021), pp. 62–71.

[40] P. Sanders and C. Schulz, *Engineering multilevel graph partitioning algorithms*, in Algorithms—ESA 2011, Lecture Notes in Comput. Sci. 6942, Springer-Verlag, 2011, pp. 469–480, 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*, in Proceedings of the IEEE International Conference on Big Data (Big Data), 2014.

[42] B. Smith, P. Bjorstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, 2004.

[43] A. J. Soper, C. Walshaw, and M. Cross, *A combined evolutionary search and multilevel optimisation approach to graph-partitioning*, J. Global Optim., 29 (2004), pp. 225–241, https://api.semanticscholar.org/CorpusID:6904637.

[44] S.-H. Teng, *Coarsening, sampling, and smoothing: Elements of the multilevel method*, in Algorithms for Parallel Processing, M. T. Heath, A. Ranade, and R. S. Schreiber, eds., Springer, 1999, pp. 247–276.

[45] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, et al., *Kokkos 3: Programming model extensions for the exascale era*, IEEE Trans. Parall. Distrib. Syst., 33 (2021), pp. 805–817.

[46] J. Xu and L. Zikatanov, *Algebraic multigrid methods*, Acta Numer., 26 (2017), pp. 591–721, https://doi.org/10.1017/S0962492917000083.