More Recent Advances in (Hyper)Graph Partitioning

ÜMIT ÇATALYÜREK, Georgia Institute of Technology, Atlanta, GA, United States of America
KAREN DEVINE, Sandia National Laboratories, ret., United States of America
MARCELO FARAJ, Heidelberg University, Germany
LARS GOTTESBÜREN and TOBIAS HEUER, Karlsruhe Institute of Technology, Germany
HENNING MEYERHENKE, Humboldt-Universität zu Berlin, Germany
PETER SANDERS, Karlsruhe Institute of Technology, Germany
SEBASTIAN SCHLAG, Apple Inc., United States of America
CHRISTIAN SCHULZ, Heidelberg University, Germany
DANIEL SEEMAIER and DOROTHEA WAGNER, Karlsruhe Institute of Technology, Germany

In recent years, significant advances have been made in the design and evaluation of balanced (hyper)graph partitioning algorithms. We survey trends of the past decade in practical algorithms for balanced (hyper)graph partitioning together with future research directions. Our work serves as an update to a previous survey on the topic [29]. In particular, the survey extends the previous survey by also covering hypergraph partitioning and has an additional focus on parallel algorithms.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Networks → Network dynamics**; • **Mathematics of computing → Graph algorithms**

Additional Key Words and Phrases: Graph partitioning, hypergraph partitioning, load balancing

ACM Reference format:
Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* 55, 12, Article 253 (March 2023), 38 pages.
https://doi.org/10.1145/3571808

We acknowledge support by DFG Grants No. SCHU 2567/1-2, No. WA654/19-2, No. SA933/10-2, No. SA933/11-1, and NSF Grant No. CCF-1919021.
Authors' addresses: Ü. Çatalyürek, Georgia Institute of Technology, Atlanta, GA, North Avenue, Atlanta, GA 30332, USA; email: umit@gatech.edu; K. Devine, Sandia National Laboratories, ret., Albuquerque, NM, USA; email: devine.hpc@gmail.com; M. Faraj and C. Schulz, Heidelberg University, Im Neuenheimer Feld 205, Heidelberg, Baden-Württemberg, Germany, 69120; emails: {marcelofaraj, christian.schulz}@informatik.uni-heidelberg.de; L. Gottesbüren, T. Heuer, P. Sanders, D. Seemaier, and D. Wagner, Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, Baden-Württemberg, Germany, 76131; emails: {lars.gottesbueren, tobias.heuer, sanders, daniel.seemaier, dorothea.wagner}@kit.edu; H. Meyerhenke, Humboldt-Universität zu Berlin, Unter den Linden 6, Berlin, Germany, 10099; email: meyerhenke@hu-berlin.de; S. Schlag, Apple Inc., Cupertino, CA, USA; email: sebastian_schlag@apple.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0360-0300/2023/03-ART253 $15.00
https://doi.org/10.1145/3571808

1 INTRODUCTION

Graphs are a universal and widely used way to model relations between objects. They are useful in a wide range of applications from social networks, simulation grids, road networks/route planning, (graph) neural networks, and many more. With exploding data sets, the scalability of graph processing methods has become an increasing challenge. As huge problem instances in the area become abundant, there is a need for scalable algorithms to perform analysis. Often (hyper)graph partitioning is a key technology to make various algorithms scale in practice. More precisely, scalable algorithms for various applications often require a subroutine that partitions a (hyper)graph into k blocks of roughly equal size such that the number of edges that run between blocks is minimized. Balance is most often modeled using a *balancing constraint* that demands that all block weights are below a given upper bound. The number of applications that require such partitions of (hyper)graphs as key subroutine is huge. Social network operators use graph partitioning techniques to load balance their operation and make sure that site response times are low [95]; efficient route planning algorithms rely on a precomputation phase in which a road network needs to be partitioned [43]; scientific simulations that run on supercomputers use (hyper)graph partitioning to balance load and minimize communication between processors [35, 58, 158]; partitioning (graph) neural networks is helpful to speedup training times [193]; and (hyper)graph partitioning helps to improve chip placements in **very-large-scale integration (VLSI)** design [7, 137].

On the other side, the problem is NP-hard [62] even for unweighted trees of maximum degree three [9] and no constant-factor approximation algorithms exist [11]. Thus, heuristic algorithms are used in practice. The purpose of this article is to give a structured overview of the rich literature, with a clear emphasis on explaining key ideas and discussing work that has been published in the last decade and is missing in other overviews. More precisely, our work serves as an update to a previous generic survey on the topic [29]; in particular, we also extend the scope of the survey to hypergraph partitioning algorithms.

There have been other (older) surveys on the topic. The book by Bichot and Siarry [23] covers techniques for graph partitioning such as the multilevel method, metaheuristics, parallel methods, and hypergraph partitioning, as well as applications of graph partitioning. The survey by Schloegel et al. [159] was published around the turn of the millennium and has a focus on techniques for scientific computing, including algorithms for adaptive and dynamic simulations and process mapping algorithms. Static algorithms as well as formulations with multiple objectives and constraints are also discussed. Monien et al. [130] discuss multilevel algorithms. Their description has a focus on matching-based coarsening and local search that use vertex-swapping heuristics. Kim et al. [105] cover memetic algorithms. The last generic survey on the topic by Buluç et al. [29] covers a wide range of techniques and practical algorithms that have been published in or before 2013. For hypergraph partitioning there exist two older surveys [7, 137]. Alpert and Kahng [7] discuss min-cut and ratio cut bipartitioning formulations along with multi-way extensions, constraint-driven partitioning and partitioning with replication. Papa and Markov [137] discuss practical applications of hypergraph partitioning, exact algorithms, and various local search heuristics for the problem as well as software packages and benchmarks. Schlag [154] presents a recent overview of hypergraph techniques in his dissertation.

Our survey[1] is structured as follows. We start by introducing the problems and other preliminaries in Section 3. Then in Section 4, we focus on new applications that emerged in the last decade. We continue with novel sequential techniques in Section 5. Then, we continue with covering

[1]This version of the survey is a shortened version of Reference [37]. The extended technical report also contains material on streaming graph partitioning as well as process mapping.

parallel algorithms in Section 6. We describe experimental methodology in Section 7. We conclude with future challenges in Section 8.

2 OVERVIEW/CLASSIFICATION

The field is currently very active. On the one hand, there are *streaming algorithms* that are very fast and consume little memory yet yield only low-quality solutions. These algorithms assume that internal memory of a single machine proportional to the number of vertices is available and typically load nodes and their neighborhood one by one to directly make an assignment to a block. Hence, the edges of a graph do not have to fit into the memory of the machine. On the other hand, there is a wide range of sequential internal-memory algorithms that tackle partitioning problems that fit into the main memory of a single machine without using parallelism. Recent advances in this area provide a rich set of algorithms that are able to compute partitions of very high quality. These high-quality algorithms often also have shared-memory parallel counterparts. While the sequential internal memory/parallel shared-memory non-metaheuristic algorithms in this area use a reasonable amount of time for most applications, there are also memetic or evolutionary algorithms that invest a lot of resources to achieve even higher quality. Note that memetic or evolutionary algorithms can be sequential internal-memory, shared-memory or even distributed-memory parallel. Due to their high running time, such algorithms are typically used only on graphs having a few hundred thousand nodes. Exact solvers are currently able to solve only very small instances while also requiring a lot of time to solve an instance even for few blocks. For larger numbers of blocks, the exact solvers currently do not work well. Distributed parallel algorithms scale well to large instances, but if these tools do not implement multilevel strategies, they typically have much lower quality than sequential internal memory partitioning algorithms. Even when such algorithms use the multilevel scheme, solution quality of the algorithms operating in this model of computation is typically worse compared to their internal memory/shared-memory counterparts. However, this mode of computation has the advantage that huge instances can be partitioned very quickly and also that it enables researchers to partition huge graphs on cheap machines. Last, researchers also work on algorithms for other platforms like GPUs to be able to fully use the capabilities of existing hardware. However, GPUs have limited memory size and current algorithms running on a GPU compute partitions that cut significantly more edges than high-quality internal memory schemes.

3 PRELIMINARIES

3.1 Notation

Hypergraphs and Graphs. A *weighted undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m hyperedges/nets E with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net e is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*. We extend c and ω to sets in the natural way, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $v \in e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The set $\Gamma(v) := \{u \mid \exists e \in E : \{v, u\} \subseteq e\}$ denotes the neighbors of v. The *degree* of a vertex v is $d(v) := |\mathrm{I}(v)|$. We assume nets to be sets rather than multisets, i.e., a vertex can only be contained in a net *once*. Nets of size one are called *single-vertex* nets. Given a subset $V' \subset V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\})$.

A *weighted undirected graph* $G = (V, E, c, \omega)$ is defined as a set of n vertices V and a set of m edges E with vertex weights $c : V \to \mathbb{R}_{>0}$ and edge weights $\omega : E \to \mathbb{R}_{>0}$. In contrast to hypergraphs, the size of the edges is restricted to two. Let $G = (V, E, c, \omega)$ be a weighted (directed) graph. We use *hyperedges/nets* when referring to hypergraphs and *edges* when referring to graphs. However, we use the same notation to refer to vertex weights c, edge weights ω, vertex degrees $d(v)$, and the set of neighbors Γ. In an undirected graph, an edge $(u, v) \in E$ implies an edge $(v, u) \in E$ and $\omega(u, v) = \omega(v, u)$.

Partitions and Clusterings. A *k-way partition* of a (hyper)graph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \dots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$, and $V_i \cap V_j = \emptyset$ for $i \neq j$. We call a k-way partition Π *ε-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\left\lceil \frac{c(V)}{k} \right\rceil$ for some parameter ε.[2] We call a block V_i *overloaded* if $c(V_i) > L_{\max}$. For each net e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a net e is the cardinality of its connectivity set, i.e., $\lambda(e) := |\Lambda(e)|$. A net is called a *cut net* if $\lambda(e) > 1$; otherwise (i.e., if $|\lambda(e)| = 1$), it is called an *internal* net. A vertex u that is incident to at least one cut net is called a *border vertex*. The number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{V_i \cap e\}|$. A block V_i is *adjacent* to a vertex $v \notin V_i$ if $\exists e \in \mathrm{I}(v) : V_i \in \Lambda(e)$. We use $\mathrm{B}(v)$ to denote the set of all blocks adjacent to v. Given a k-way partition Π of H, the *quotient graph* $Q := (\Pi, \{(V_i, V_j) \mid \exists e \in E : \{V_i, V_j\} \subseteq \Lambda(e)\})$ contains an edge between each pair of adjacent blocks. A *clustering* $C = \{C_1, \dots, C_l\}$ of a hypergraph is a partition of its vertex set. In contrast to a k-way partition, the number of clusters is not given in advance, and there is no balance constraint on the actual sizes of the clusters C_i.

A *k-way (hyper)edge partition* of a (hyper)graph H is a partition of its (hyper)edge set into k *blocks* $\Pi = \{E_1, \dots, E_k\}$ such that $\bigcup_{i=1}^{k} E_i = E$, $E_i \neq \emptyset$ for $1 \leq i \leq k$, and $E_i \cap E_j = \emptyset$ for $i \neq j$. We call a k-way (hyper)edge partition Π *ε-balanced* if each block $E_i \in \Pi$ satisfies the *balance constraint*: $\omega(E_i) \leq (1 + \varepsilon)\lceil \frac{\omega(E)}{k} \rceil$ for some parameter ε. A vertex is called a *cut vertex* if it contains two or more edges in different blocks. The (weighted) *vertex-cut* of a *k-way (hyper)edge partition* is the total weight of cut vertices.

3.2 The k-way (Hyper)Graph Partitioning Problem

Problem Definition. The *k-way (hyper)graph partitioning problem* is to find an ε-balanced k-way partition Π of a (hyper)graph $H = (V, E, c, \omega)$ that *minimizes* an objective function over the cut nets for some value of ε. The two most commonly used cost functions in case we are dealing with hypergraphs are the *cut-net* metric $\mathfrak{f}_c(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $\mathfrak{f}_\lambda(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\ \omega(e)$, where E' is the *cut-set* (i.e., the set of all cut nets) [44, 49]. While the cut-net metric sums the weights of all nets that connect more than one block of the partition Π, the connectivity metric additionally takes into account the actual number λ of blocks connected by the cut nets. When partitioning graphs, the objective is often to minimize $\sum_{i<j} \omega(E_{ij})$ (weight of all cut edges), where $E_{ij} := \{\{u, v\} \in E \mid u \in V_i, v \in V_j\}$. Note that for graphs the objective functions $\mathfrak{f}_c(\Pi)$ and $\mathfrak{f}_\lambda(\Pi)$ revert to edge-cut (i.e., the sum of the weights of those edges that have endpoints in different blocks). Apart from these common cost functions, other specialized objective functions exist [29]. Figure 1 shows an example partition of a graph. Throughout the article, we use the term high quality if the respective objective function is small compared to other tools, and low quality if the opposite is the case.

4 APPLICATIONS

Research on graph partitioning algorithms has always been motivated by its numerous applications. Some traditional applications are parallel processing (e.g., in scientific computing), VLSI design, route planning and image segmentation; see also the previous survey [29]. Yet, in recent years several new applications have emerged.

[2] The $\lceil \cdot \rceil$ in this definition ensures that there is always a feasible solution for inputs with unit vertex weights. For general weighted inputs, there is no commonly accepted way how to deal with feasibility; see also References [74, 88].

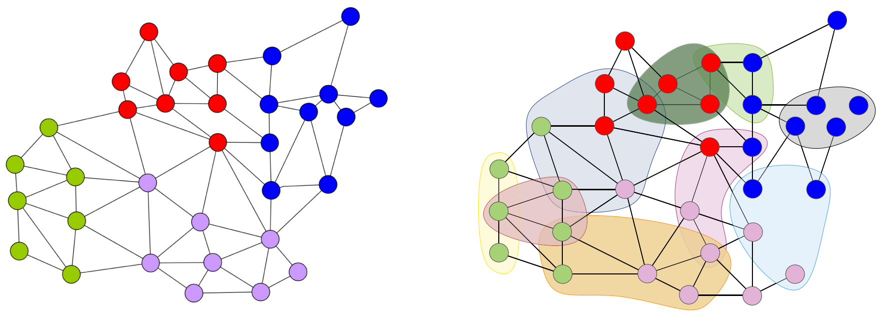

Fig. 1. Left: An example graph that is partitioned into four blocks indicated by the colors. Vertices are the small colored circles. The partition has an edge cut of 17, the block weights $|V_i|$ are green (7), red (8), blue (10), purple (8). Right: An example hypergraph that is partitioned into four blocks indicated by the colors. Hyperedges are indicated by black lines or colored blobs around the vertices (if more than two nodes are contained in a net). Blocks have the same weight as in the graph case. The connectivity objective is 22 and the number of nets cut is 20.

Distributed Databases and Query Optimization. A prominent application is distributed database sharding [14, 41, 95, 109, 187]. Vertices represent data records, nets represent queries that access records, and blocks correspond to shards (machines). Minimizing connectivity thus corresponds to minimizing the average number of shards involved in a query, which optimizes the latency and query processing time.

A similar application is to boost IO throughput in the Google search engine backend by improving cache utilization. Archer et al. [13] initialize a voting table that assigns search requests, using graph partitioning on a bipartite graph where vertices are search terms and queries, and an edge exists for each term contained in a query. Subsequent simulation-and-refinement further boosts the prediction accuracy (item in cache) of the voting table.

Li et al. [116] use graph partitioning to construct a set similarity search index. The approach first filters candidate sets using the index, then checks the remaining candidates brute-force. Vertices represent sets, edges connect sets that are k-nearest neighbors (or within a similarity threshold, depending on the query type), and cut size is correlated with pruning efficiency.

Data Distribution and Scheduling. Djidjev et al. [47] use graph partitioning to accelerate and parallelize the computation of density matrices for molecular dynamics simulations. Lattice Boltzmann Fluid Flow simulations [57, 65, 108] fall into the classic load balancing and communication minimization application category of graph partitioning. Yu et al. [192] consider sparse matrix vector multiplication [32], however focused on shared-memory machines with many NUMA nodes instead of distributed systems.

Parallel graph processing systems such as Pregel [124] or Giraph [82] have become wide-spread tools for network analysis tasks. These systems employ a think-like-a-vertex or think-like-an-edge programming paradigm, requiring a balanced partition of vertices or edges across machines. Streaming approaches [91, 123] based on label propagation result in better performance than traditional range-based or hash-based partitioning.

Quantum Circuit Simulation. Several papers employ nested dissection on hypergraphs to find good contraction trees, to speed up the simulation of quantum circuits on classical non-quantum machines [77, 93, 136]. This can be used to experimentally verify the correctness of a quantum circuit, and push back premature claims of having achieved quantum supremacy.

SAT Solving. **Boolean satisfiability (SAT)** formulas can be represented as hypergraphs. One representation is the dual representation with clauses as vertices and variables as nets, consisting of clauses that contain the variable (either negated or not). Mann and Papp [121] use balanced bipartitioning to identify variables that branching solvers should focus on assigning first, such that the formula is split into two once variables in the cut are assigned. The two sub-formulas can be solved independently, which is expected to be faster. Most SAT solvers employ a heuristic (**variable state independent decaying sum (VSIDS)**) that assigns priorities (frequency in observed conflict clauses) to variables, selecting the highest priority to branch on next. The authors consider two variants, the latter of which is more successful: force split the formula first and use VSIDS for tie-breaking, or initialize the priorities with the partition and let VSIDS overrule the decision.

Miscellaneous. Lamm et al. [111] formulate recombine-operators in an evolutionary framework for independent sets based on separator decompositions and graph partitions. Given two independent sets, their vertices in the two blocks of a separator decomposition are exchanged, yielding two offspring independent sets that are refined using local search.

Kumar et al. [110] use a clustering formulation to compute trajectories of moving objects in videos. Detected objects in a frame correspond to vertices and each block corresponds to an object. The authors enforce special constraints such that each partition contains only one object per frame and such that objects do not jump between frames.

Yao et al. [188] consider the placement of control units in software-defined networks to minimize the average latency of messages from switches (nodes) to controllers. First the number of necessary controllers is estimated based on message volume capacities, then the switch graph is partitioned and one control unit is placed in each block, using a separate routine to perform placement inside the blocks.

Quantum chemical simulations such as *density functional theory* exhibit quadratic time complexity, which is why calculations on largely independent sub-systems are used to approximate and accelerate the process. Von Looz et al. [178] propose to use graph partitioning to automatize the sub-system construction process, where small edge cuts correspond to small introduced calculation errors.

5 SEQUENTIAL TECHNIQUES

5.1 Classic (Hyper)Graph Partitioning Techniques

This section discusses partitioning techniques repeatedly used in this survey. We briefly outline the multilevel paradigm and the most common local search algorithms. These algorithms move vertices according to gain value. The gain value $g_u(V_j)$ reflects the change in the objective function when we move a vertex u from its current to a target block V_j (e.g., reduction in the edge cut).

The Multilevel Paradigm. The most successful approach to solve the (hyper)graph partitioning problem is the *multilevel* paradigm. It consists of three phases. In the *coarsening* phase, a hierarchy of successively smaller and structurally similar (hyper)graphs are created by contracting matchings or clusters of vertices. Once the (hyper)graph is small enough, an *initial partitioning* algorithm obtains a partition of the coarsest (hyper)graph. In the *uncoarsening* phase, the partition is projected to the next larger (hyper)graph in the hierarchy, and, at each level, *local search* algorithms improve the objective function (e.g., edge cut). Figure 3 illustrates the multilevel paradigm.

A contraction of several vertices into a supervertex aggregates their weight in the supervertex. To further reduce the size of the coarser (hyper)graph, one can also remove all edges that become identical except for one, at which their weight is aggregated (self-loops or single-vertex nets in the

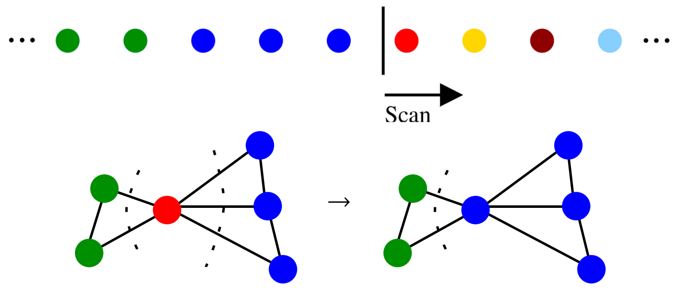

Fig. 2. Label propagation algorithm in graphs. Top: nodes are scanned in a specific order. Colors of nodes indicate their block. The algorithm initially assigns each node its own label and then visits the vertices in some order, assigning the current vertex the label that is most frequent in its neighborhood.

hypergraph case are discarded). This way, one obtains a partition with the same balance and cut properties when projecting a partition to the next larger hypergraph in the hierarchy.

The Label Propagation Algorithm. The label propagation algorithm, illustrated in Figure 2, was originally proposed to detect community structures in large-scale networks [142, 194] but was also used as a refinement technique in the partitioning context [101]. The label propagation algorithm works in rounds, and each vertex u is associated with a label $L[u]$. Initially, each vertex is assigned its own label (i.e., $L[u] = u$). In each round, the vertices are visited in some order, and whenever a vertex u is visited, it adapts its label to the label appearing most frequently in its neighborhood (ties are broken randomly). The algorithm proceeds until it reaches a predefined number of rounds or none of the vertices changes its label in a round. The algorithm can be used as a clustering algorithm in the coarsening phase [126] or a local search algorithm when the current k-way partition is used for the initial label assignment.

The Fiduccia-Mattheyses Algorithm. Fiduccia and Mattheyses [55] presented the first linear-time heuristic for the balanced bipartitioning problem (referred to as FM algorithm). The algorithm works in passes, and a vertex can change its block at most once in a pass. The FM algorithm uses two priority queues (one for each block). Initially, it inserts each boundary vertex into the priority queue, along with the gain of moving the vertex to the opposite block. Each step repeatedly performs the move with the highest gain that does not violate the balance constraint and, subsequently, updates the gain of all non-moved neighbors. A pass ends when all vertices are moved, or the balance constraint prevents further moves. The FM algorithm also performs negative gain moves and is therefore able to escape from local optima. In the end, it reverts to the best seen solution during the pass.

5.2 Recent Advances in Multilevel Partitioning

This section discusses recent developments in the multilevel partitioning context. We start with two new multilevel partitioning schemes preferable in situations where either the number of blocks is large or high solution quality is required. We then take a closer look at the different phases of the multilevel scheme and highlight recent algorithmic improvements.

Deep Multilevel Partitioning. A k-way partition of a (hyper)graph can be obtained either by ***recursive bipartitioning*** **(RB)** or *direct k-way partitioning*. The former computes a bipartition of the input (hyper)graph and then recurses on both blocks until the (hyper)graph is divided into the desired number of blocks. The latter partitions the (hyper)graph directly into k blocks and applies k-way local search algorithms to improve the solution.

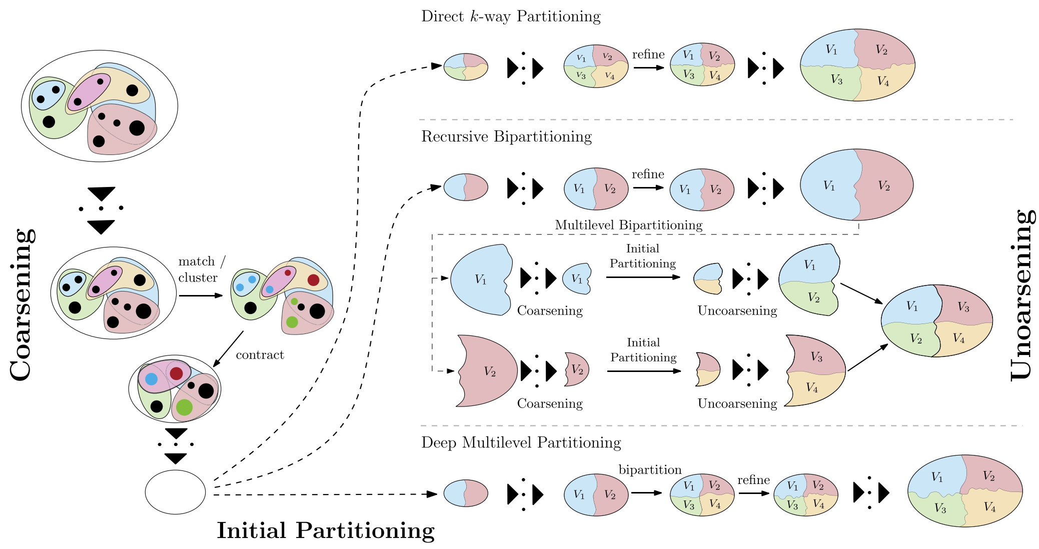

Fig. 3. The multilevel paradigm and its different instantiations to obtain a k-way partition.

Recently, these approaches have been generalized to *deep multilevel partitioning* [74]. The approach continues coarsening until only $2X$ vertices are left (where X is an input parameter) and computes an initial bipartition of the coarsest (hyper)graph. In the uncoarsening phase, it bipartitions a block when its size becomes larger than $2X$ vertices as long as there are less than k blocks. Thus one arrives at a k-partition in the end. See Figure 3 for an illustration.

The deep multilevel approach combines the strengths of the RB and direct k-way scheme: It recursively bipartitions blocks of size $O(1)$ in the uncoarsening phase and thereby enables using k-way local search algorithms, while the RB scheme recursively bipartitions blocks of size $O(k)$ and direct k-way reverts to RB for initial partitioning [3, 17, 101, 160]. Figure 3 illustrates the different instantiations of the multilevel scheme.

n-Level (Hyper)Graph Partitioning. The depth of the multilevel hierarchy offers a trade-off between running time and solution quality of a multilevel algorithm [6]. More levels provide "more opportunities to refine the current solution" [6] at different granularities but require fast local search algorithms to achieve reasonable running times. The most extreme version of the multilevel paradigm is to contract only a single vertex at each level, which induces a hierarchy with almost n levels. This technique was first studied by Osipov et al. [135] for graph partitioning and later improved further by Schlag et al. [3, 154, 155] for hypergraph partitioning. The approach is made feasible by using a highly localized variant of the classical FM algorithm [55] that initializes the priority queue only with the uncontracted nodes and expands the search to neighbors of moved nodes [135, 147]. Furthermore, the implementation uses a gain cache to avoid expensive recomputations of move gains [3] (reducing the running time of the FM algorithm by 45%) and an adaptive stopping rule that terminates a search early if it becomes unlikely to find further improvements [135] (reducing the running time by an order of magnitude). A parallel version of the n-level scheme exists [72, 73] that uncontracts a fixed number of vertices in parallel on each level (instead of a single vertex).

Coarsening. The coarsening phase aims to compute successively coarser approximations of the input (hyper)graph such that its structural properties are maintained and act in some sense as a filter that removes as much unnecessary information from the search space as possible [179]. In the past years, research focused on clustering techniques for complex networks and enhancing the coarsening process with information about the community structure of the (hyper)graph.

Meyerhenke et al. [127] use the size-constraint label propagation algorithm [142] to compute a vertex clustering, which is then contracted to form the multilevel hierarchy. Clustering algorithms can reduce the size of complex networks (power-law degree distribution) more efficiently than previously used matching-based approaches. The latter has the problem that vertices incident to high-degree vertices often remain unmatched. Therefore, Davis et al. [42] propose several techniques to reduce the number of unmatched vertices. Two non-adjacent unmatched vertices are matched if they share a common neighbor (also known as two-hop matching [98]) or if they are adjacent to two vertices that are already matched. Additionally, one can extend two matched vertices with an unmatched neighbor (three-way match).

The following two techniques recompute the (hyper)edge weights of the input (hyper)graph and use them as an input for a multilevel algorithm. The new weights encode more information about the importance of a (hyper)edge and should prevent a coarsening algorithm from collapsing (hyper)edges into a single vertex that are in the cut set of a good partition. Glantz et al. [64] define the weight of an edge $e \in E$ as the minimum conductance value of all bipartitions induced by a spanning tree in which e is a cut edge. The conductance of a bipartition (V_1, V_2) is $cond(V_1, V_2) := \frac{\omega(E'(V_1, V_2))}{\min\{vol(V_1), vol(V_2)\}}$ where $vol(V_i)$ is the weight of all edges incident to vertices in V_i. Chen and Safro [162] use the maximum algebraic distance [39, 145] between two pins of a net as its weight.

Lotfifar and Johnson [119] compute a net (hyperedge) clustering and assign a vertex to the cluster containing most of its incident nets. Their matching-based coarsening algorithm uses the restriction that matched vertices must be part of the same cluster. Heuer and Schlag [90] also use vertex clustering to restrict contractions to densely coupled regions of a hypergraph. Their algorithm transforms the hypergraph into its bipartite graph representation and then uses the Louvain algorithm [26] maximizing the modularity objective function to exploit its community structure.

Shaydulin and Safro [163] use ideas from algebraic multigrid as well as stable matching in which the preference lists determine a good combination of nodes for aggregation-based coarsening. The authors integrate their approaches into the Zoltan tool. Their experimental results with Zoltan, hMetis and PaToH demonstrate that given the same refinement, the proposed schemes are at least as effective as traditional matching-based schemes, while outperforming them on many instances.

Initial Partitioning. Coarsening usually proceeds until $\Omega(k)$ vertices remain. Partitioners based on the direct k-way scheme often use multilevel recursive bipartitioning to obtain an initial k-way partition of the coarsest (hyper)graph [3, 17, 101, 160]. Heuer [87] showed that this leads to better initial partitions than flat partitioning techniques. To obtain an initial bipartition, many partitioners run a *portfolio* of different flat bipartitioning algorithms multiple times (e.g., greedy graph growing, random or spectral partitioning) followed by label propagation or FM local search and continue uncoarsening with the best bipartition out of these runs [3, 33, 98]. Preen and Smith [141] showed that the decision when to stop coarsening is often instance-dependent and proposed an adaptive stopping criterion (instead of using the same constant number of vertices for all instances).

Refinement. Sanders and Schulz [148] propose a flow-based refinement technique for bipartitions that is scheduled on block pairs for k-way partitions. The idea is to choose a subset R of vertices around the cut of a bipartition $\Pi = \{V_1, V_2\}$. A flow network is constructed by contracting $V_1 \setminus R$ to the source and $V_2 \setminus R$ to the sink. The resulting maximum flow induces a possibly improved cut, which may violate the balance constraint. Therefore, $|R|$ is chosen adaptively and

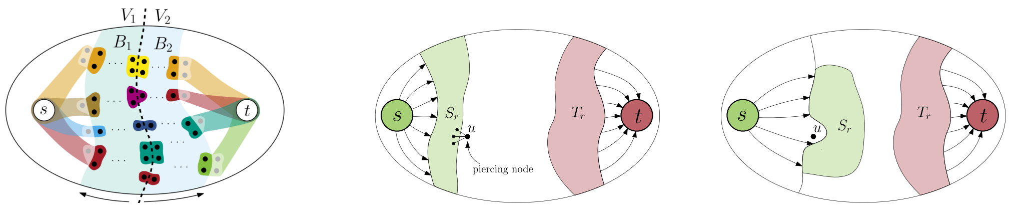

Fig. 4. Left: a flow network that is constructed around a bipartition of a hypergraph. Running a max-flow min-cut algorithm on this network is used to find a reduced cut in the original network. Right: illustration of one step in FlowCutter [80, 186], where the smaller side (source-side) plus a piercing node are contracted to the corresponding terminal.

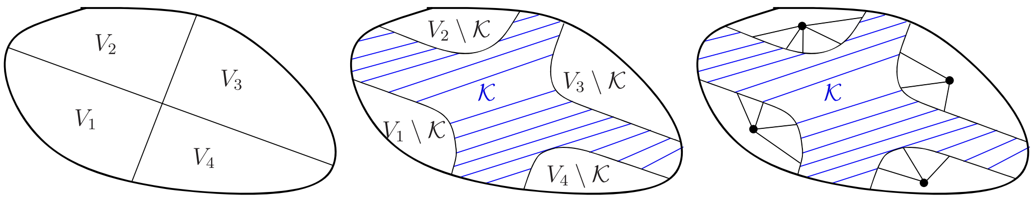
V_2
V_3
V_1
V_4
$V_2 \setminus \mathcal{K}$
$\mathcal{K}$
$V_3 \setminus \mathcal{K}$
$V_1 \setminus \mathcal{K}$
$V_4 \setminus \mathcal{K}$
$\mathcal{K}$

Fig. 5. ILP-based refinement. From left to right: a graph that is partitioned into four blocks; the set $\mathcal{K}$ close to the boundary that will stay in the ILP model; the model in which the sets $V_i \setminus \mathcal{K}$ have been contracted. The latter is used as input to an ILP graph partitioning solver and the improved result is adopted.

a strongly connected-component calculation is used to characterize the set of *all* minimum cuts that can then be searched for a balanced bipartition. Heuer et al. [89] generalize this approach to hypergraphs. Gottesbüren et al. [69, 70] accelerate flow-based hypergraph refinement by running the flow algorithm directly on the hypergraph. By incorporating FlowCutter [80, 186], which solves incremental flow problems to trade off cut size for better balance, they explore the solution space more effectively, which leads to partitions with smaller cuts/connectivity. Examples are shown in Figure 4. FlowCutter [80, 186] is an iterative algorithm that starts with the minimum cut (and corresponding bipartition) between a given source and sink. If the bipartition is imbalanced, then the smaller block is grown by contracting all of its vertices plus an additional vertex (piercing node) to the corresponding terminal. This vertex is chosen incident to the cut, and if possible without increasing the cut-size in the following iteration. The resulting flow problems are nested in the sense that the terminal sets only grow, which makes the flow assignment from the previous iteration feasible but its maximality is violated.

Henzinger et al. [84] use **integer linear programming (ILP)** to refine k-way partitions directly. Analogously to flow-based refinement, a small region of vertices allowed to move is selected. The remaining vertices are contracted to one super-vertex per block. The ILP formulation is then run on this model graph. An example is shown in Figure 5. Techniques to speed up the ILP are giving a heuristic solution to the solver and symmetry breaking by fixing sufficiently heavy super-vertices to their corresponding blocks. Ugander and Backstrom [174] use linear programming to select from a set of possible moves a subset that yields the largest reduction in cut while satisfying the balance constraint.

In experiments, flow-based (hyper)graph refinement turns out to be a key ingredient of high-quality partitioning, i.e., it has a much better cost-benefit-ratio than other approaches to improve quality like using ILPs, evolutionary techniques, restarts, V-cycles, and so on.

Partitioning With Unevenly Distributed Vertex Weights. Whereas real-world applications often use vertex and edge weights to accurately model the underlying problem, the (hyper)graph partitioning research community commonly works with unweighted instances [88, 154]. Multilevel algorithms incorporate techniques to prevent the formation of heavy vertices (restricting the maximum allowed vertex weight or incorporate the weight of a vertex into the coarsening rating function as a penalty term), but considerably struggle if such vertices are already present in the input [88], as is the case for many real-world instances [5].

There are two approaches to compute balanced k-way partitions under a tight balance constraint in a multilevel partitioner: (i) ensure that initial partitioning finds a balanced partition [88] and that refinement applies node moves to the partition only when they satisfy the balance constraint, or (ii) allow intermediate balance violations [30, 31, 52, 74] and use rebalancing techniques to ensure that the final k-way partition is balanced [74, 120, 150, 180]. Given that finding a balanced k-way partition is a NP-hard problem [61] (reducible to the most common version of the job scheduling problem), both approaches do not guarantee balance.

Recently, Heuer et al. [88] proposed a technique that enables partitioners based on recursive bipartitioning to reliably compute balanced partitions in practice. The idea is to preassign a small portion of heaviest vertices to one of the two blocks (treated as fixed vertices) and optimize the objective function on the remaining vertices.

5.3 Evolutionary Computation

A **genetic or evolutionary algorithm (GA)** starts with a population of individuals (in our case, partitions of the (hyper)graph) and evolves the population over several generational cycles or rounds. If a genetic algorithm is combined with local search, then it is called a **memetic algorithm (MA)** [105]. In each round, the GA uses a selection rule to select good individuals based on the fitness of the individuals of the population and combines them to obtain improved offspring [66]. The combination step is typically done using a recombination operation. When an offspring is generated, an eviction rule is used to select a member of the population to be replaced by the new offspring. For an evolutionary algorithm, it is of major importance to preserve diversity in the population [18]; i.e., the individuals should not become too similar to avoid premature convergence of the algorithm. This is usually achieved by using mutation operations and by using eviction rules that take similarity of individuals into account. All of the algorithms in the literature follow a rather generic overall scheme. Hence, we focus on the description of the recombination operations.

Benlic and Hao [21] cluster large sets of vertices together that have been assigned to the same block in each individual to perform a recombination operation and combine their operator with tabu search. The recombination operation is motivated by the observation that given a number of high-quality solutions, there is always a high number of vertices that are clustered together throughout these solutions.

Sanders and Schulz introduced a distributed evolutionary algorithm, **KaFFPaE (KaFFPaEvolutionary)** [149]. The recombination operation uses a modified version of the multilevel graph partitioning solver within **Karlsruhe high-quality partitioning (KaHIP)** [148] that will not contract edges that are cut in one of the input partitions. Thus, the better of the two input individuals can be used as initial partitioning and local search can efficiently exchange good parts of solutions on multiple levels of the multilevel hierarchy. Moreover, the recombination operation guarantees that the offspring is at least as good as the better of the two input individuals.

Ruiz and Segura [146] present a memetic algorithm using a weighted matching-based recombination and diversity preservation that is similar to the algorithm by Benlic and Hao [21]. More precisely, given two individuals, the recombination operation computes a matching in a bipartite graph where the left hand side represents the blocks of the first individual and the right hand side

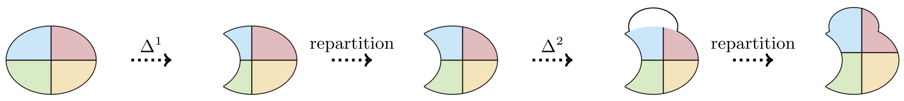

Fig. 6. Illustration of the repartitioning process. A graph that has an initial four-way partition goes through two subsequent changes: Δ^1 where portions of the graph are removed, Δ^2 where new elements (vertices and edges) are inserted in the graph. After each of these changes, a repartitioning routine adapts the previous four-way partition to the new graph state while optimizing for one or more metrics such as edge-cut, imbalance, and migration of elements between blocks.

represent the blocks of the second individual. Edges between the vertices are weighted by the size of the intersection of the corresponding blocks. In this bipartite graph, a matching is computed and the corresponding intersections are used as the core of the sets in the new partition (offspring). Moreover, the algorithms use local search with negative cycle detection to further reduce the edge-cut of the computed partitions [151]. When an offspring is inserted into the population, the individual having the highest similarity is evicted. Here, similarity between two individuals is based on the weight of the matching in the bipartite graph defined above.

Henzinger et al. [84] provide a recombination operator that is based on integer linear programming and integrate this into KaFFPaE [149]. The new recombination operation builds and solves an integer linear program from multiple individuals of the population. Roughly speaking, the authors take l individuals (partitions) and build an overlap graph by contracting pairs of nodes that are in the same block in every partition. The original partitioning problem is then solved on the (much smaller) overlap graph using integer linear programming that is initialized with the block affiliations according to the partition that has the lowest cut value. When multiple partitions have the same cut value, one is chosen at random.

Andre et al. [10] generalize KaFFPaE to the multilevel memetic hypergraph partitioner KaHyParE. They also introduce a multilevel multi-point recombination operator that is applied to the best individuals. This operator penalizes contraction of vertices that appear in nets that are cut in many parents. Preen and Smith [141] refine KaHyParE with an adaptive scheme to stop coarsening.

Besides memetic algorithms for hypergraph and graph partitioning, there are also algorithms for other problem variations. For example, Moreira et al. [131] and Popp et al. [139] gave memetic algorithms for acyclic (hyper)graph partitioning in which the input is an acyclic (hyper)graph and the partition has to fulfill an acyclicity constraint. Moreover, Schulz and Sanders [152] gave a distributed evolutionary framework to compute k-way vertex separators in graphs.

5.4 Repartitioning

Graph repartitioning is an extension of graph partitioning that copes with dynamic graphs, i.e., graphs whose set of vertices and edges are modified over time. Assuming that a graph is partitioned, dynamic changes on its components can make its blocks imbalanced and its edge-cut or vertex-cut larger. Figure 6 illustrates the evolution of a graph over time followed by repartitioning efforts, In general, a rough upper bound for repartitioning quality can be obtained by partitioning the whole graph from scratch every time it is modified. Since this is expensive, many repartitioners use faster approaches.

Vaquero et al. [176] propose a distributed repartitioning algorithm for large-scale graph processing systems. In their algorithm, hashing is used as initial partitioning for all vertices. Based on local information, a label propagation scheme iteratively migrates vertices to blocks where most of their neighbors are located until a convergence is achieved. This procedure natively deals with modifications in the graph. Their experiments show that their approach is able to keep a better edge-cut and application performance upon graph changes in comparison to the hashing algorithm. The authors did not compare their algorithm against other state-of-the art repartitioning algorithms.

Xu et al. [185] propose LogGP, a repartitioning algorithm with centralized coordination for graph processing systems. Their algorithm analyzes and reuses historical information to improve the partitioning. Particularly, LogGP builds a hypergraph by combining the graph with hyperedges that represent previous historical partitions. This hypergraph is then partitioned by a novel streaming pin partitioning algorithm. During the execution, the system uses statistical inferences from running logs to optimize the partitioning process. In their experiments, LogGP outperforms state-of-the-art streaming and repartitioning algorithms with respect to edge-cut and application runtime.

Nicoara et al. [134] propose a lightweight repartitioning algorithm and integrate it in their own distributed social network database system Hermes. Within Hermes, the initial partitioning is performed with hashing or Metis, while their repartitioning algorithm is triggered when there is a change in the vertex set of the graph. Repartitioning uses iterative local improvement of balance and edge cut. The associated data is moved only when the improvement process is finished. In their experiments with real-world workloads, their algorithm produces roughly the same edge-cut while migrating up to an order of magnitude less data in comparison with Metis (repartitioning from scratch). Moreover, their algorithm improves around 1.7 times over hashing with respect to aggregate throughput.

Huang and Abadi [94] propose Leopard, an algorithm that solves graph repartitioning while also replicating vertices. Leopard can potentially take advantage of any one-pass vertex partitioning algorithm. This is possible, since these algorithms are vertex-centered, so they can be used locally on a dynamic graph at any moment. Moreover, Leopard is the first algorithm to integrate vertex replication with the repartitioning. This replication provides fault tolerance and improves locality, which even increases the partitioning quality, since it is mostly based on local information. Their experiments show that Leopard without vertex replication maintains a partition quality comparable to running Metis from scratch at any moment. However, Leopard with vertex replication further reduces edge-cut by up to an order of magnitude.

Kiefer et al. [104] propose a repartitioning algorithm optimized for systems whose performance does not scale linearly with the input size. The authors start by formulating a penalized version of the graph partitioning problem by defining the weight of a block as the sum of weights of its vertices plus a penalty function that grows monotonically with the cardinality of the block. In practice, this formulation penalizes resource consumption, which models the nonlinear behavior of the performance of some real-world systems. The authors modify a static multilevel algorithm for graph partitioning to solve the penalized version of the problem, then this multilevel algorithm is used within an adapted version of a hybrid repartitioning strategy. Their hybrid repartitioning algorithm works by triggering a local refinement whenever the balance constraint is violated and computing a whole new partition in the background after a given number of refinements. More specifically, this new partition is computed from scratch; afterward its blocks are mapped onto the blocks of the previous partition to minimize migration costs (see extended version of this survey [37] or supplementary material for more details on process mapping). Then it replaces the current partition only if the new edge-cut compensates the migration overhead. The authors implement their algorithms on the Metis [98] framework and show that their static algorithm takes 28% more time than Metis on average while keeping a linear complexity on the amounts of vertices, edges and blocks. Finally, their repartitioning algorithm is able to keep imbalance, edge-cut, and migration time low throughout graph updates.

Fan et al. [54] study the problem of repartitioning vertices (edges) to simultaneously minimize edge-cut (vertex-cut) and the modifications to the initial partition. The authors prove robust intractability even for restricted cases of the problem. Moreover, they provide nontrivial proofs that this problem is unbounded with respect to edge-cut and vertex-cut. Based on the presented theorems, the authors derive a strategy to convert successful partitioning algorithms into repartitioning algorithms while ensuring small migration cost and keeping the same partition quality bounds of the original algorithm. As proof of concept, the authors apply their strategy on algorithms such as Fennel [172] (vertex partitioning) and **higher degree replicated first (HDRF)** [28] (edge partitioning). Finally, the authors validate their algorithm experimentally against the repartitioning algorithm Hermes [134] and algorithms to partition from scratch such as ParMetis [97]. In their experiments, their algorithms are up to an order of magnitude faster then their original counterparts while retaining a comparable or even better partition quality.

5.5 Objective Functions and Problem Variations

The simple balanced (hyper)graph partitioning problems studied in most of this survey does not capture all aspects relevant to applications. Therefore, additional issues such as multiple constraints, directed edges, matrix partitioning, process mapping, and various ways of modelling communication costs have been considered.

5.5.1 Objective Functions. Kaya et al. [102] present an experimental study on the influence of different partitioning models and metrics on the performance of parallel **sparse matrix-vector multiplication (SpMxV)**. To do so, real-life and artificial sparse matrices are partitioned using several matrix partitioning models. The authors then measure the running time of matrix-vector multiplications using three different SpMxV implementations and determine the influence of different partition metrics on the running time by performing a regression analysis. The experimental evaluation shows that *the right* partition model and metric depends on the number of processing elements used, the specific SpMxV implementation and the size and structure of the matrix. More specifically, minimizing the total number of messages sent is more important than minimizing the per-processing element send volume, in particular if the number of processing elements is ≥ 64. The checkerboard partition model [34] reduces most of the partition metrics and generally achieves the lowest running time.

Originally proposed by Schloegel et al. [157], the multi-objective approach attempts to minimize multiple objective functions simultaneously. To this end, Deveci et al. [44] recently introduced the multi-objective hypergraph partitioner UMPa. While previous work [24, 173] on this problem variation minimized multiple objectives in multiple phases, where the first phase optimizes the primary objective, which may not be worsened by later phases, UMPa uses a single-phased approach to optimize a primary metric in combination with a secondary and tertiary metric. This is done by greedily moving boundary vertices such that the primary metric is optimized, while using the secondary (and tertiary) metrics for tie-breaking if multiple eligible target blocks maximize the primary (or secondary) metric. The choice of the best target block of a boundary vertex depends on its gain values, i.e., the change of the metric if the vertex is moved to another block. To this end, the authors describe how gain values can be computed efficiently for the communication volume and number of messages metrics as well as both metrics on a per-block basis.

5.5.2 Multi-constraint Graph Partitioning. Balanced graph partitions are usually constrained by an upper limit on the weight of each block. In the multi-constraint graph partitioning model introduced by Karypis and Kumar [99], each vertex of the graph is associated with a vector of weights, and a balance constraint is imposed on each weight. Multi-constraint partitioning was extended for hypergraphs [38], enabling 2D coarse-grain partitioning of sparse matrices/graphs [34]. Recently, Recalde et al. [144] introduced an exact algorithm for the multi-constraint graph partitioning problem. Two integer programming formulations are provided, and the authors prove several families of inequalities associated with the respective polyhedras. Using a branch-and-bound-based approach, Recalde et al. can solve real-world instances with 30 vertices in approximately half a minute of CPU time. Moreover, several of the proven families of inequalities significantly reduce the number of branch-and-bound vertices and the optimality gap.

5.5.3 Directed Acyclic (Hyper)graph Partitioning. If the input instance is a **directed acyclic graph (DAG)**, then one is often interested in finding a partition with an acyclic quotient graph. Moreira et al. [132] show that perfectly balanced graph partitioning is NP-complete even under the additional acyclicity constraint, and that there are no constant-factor approximation algorithms for $k \geq 3$. Herrmann et al. [85, 86] and Moreira et al. [132] both propose multilevel heuristics for DAG partitioning. Popp et al. [139] adapt these techniques to directed acyclic hypergraph partitioning.

Moreira et al. [132] coarsen the graph using size-constraint label propagation [127]. Since this algorithm is not adapted to DAGs, contracting the computed clusters creates coarse graphs with cycles. Thus, the partitioner computes an initial acyclic partition *before* coarsening. Restricting clusters to single blocks then ensures that the partition of the input graph can also be used as a partition of the coarsest graph. Herrmann et al. [86] present an acyclic clustering algorithm, which ensures that coarse graphs are acyclic. The top-level of a DAG vertex v is defined as the longest distance from any source of the DAG to v. Based on the top-level of adjacent vertices, conditions for inter- and intra-cluster edges are formulated and it is shown that a clustering respecting those conditions yields an acyclic coarse graph. Moreira et al. compute an initial partition by cutting the topologically ordered vertices of the graph into k chunks of equal size. Herrmann et al. adapt greedy graph growing to initial bipartitioning [85] and initial k-way partitioning [86]. Moreover, bipartitioning the graph as if it was undirected (using, for instance, Metis [98]) and fixing the acyclicity constraint afterwards [86] seems promising. For refinement, both Herrmann et al. and Moreira et al. adapt the FM algorithm to DAG partitioning by restricting the set of possible vertex moves to those that do not violate the acyclicity constraint. Herrmann et al. [86] note that this condition can be checked efficiently if the partition has only two blocks.

5.5.4 Symmetric Rectilinear Matrix Partitioning. Yaşar et al. [189, 190] consider the symmetric rectilinear (Cartesian) sparse matrix partitioning problem. Here, the columns and rows of a sparse matrix are partitioned using the same partition vector (in contrast to the more general rectilinear matrix partitioning problem [78, 122], in which different vectors are used). This yields a tiling of the sparse matrix with square tiles on the diagonal. The goal is to find a partition that minimizes the weight (i.e., the sum of all nonzero matrix entries assigned to a tile) of the heaviest tile or, given an upper limit on the weight of a tile, to find a partition subject to the constraint with a minimum number of cuts. The authors show that both problem variations are NP-complete [189], propose heuristics to optimize either objective, and provide efficient implementations thereof. Running time is reduced by sparsifying the matrix. Using these techniques, the algorithm can partition a graph representing the Twitter social network with approximately 1.5 billion edges in less than 3 seconds on a modern 24-core system.

6 PARALLEL ALGORITHMS

Most of the initial work on graph partitioning involved sequential algorithms. These algorithms have been extended to work in distributed-memory environments, in particular for balancing processor workloads in parallel applications. In this use case, a distributed-memory application already has a distribution of the graph; for memory scalability, the entire graph is not stored in every processor. Thus, distributed-memory partitioning algorithms do not typically have a global view of the entire graph; they often make partitioning decisions based on partial views of local graph data. As a result, they can have lower solution quality than their sequential counterparts.

Still, distributed-memory algorithms are crucial for graphs that are too large to fit in a single memory space, and for applications wishing to partition their data dynamically to adjust for changing computational workloads.

In recent years, progress has been made on shared memory algorithms as shared memory architectures offer greater flexibility than distributed-memory architectures. For example, random memory accesses or atomic updates can be done orders of magnitude faster compared to distributed-memory machines. Because shared-memory algorithms have a global view of the graph, they can achieve the same solution quality as their sequential predecessors. They are not feasible, however, for extremely large graphs that do not fit in a single memory space.

6.1 Shared Memory

Most of the algorithms discussed in this section are multilevel algorithms. Therefore, we structure the discussion into the separate contributions in each phase. The multilevel algorithms we consider are Mt-Metis by Lasalle and Karypis [113], Mt-KaHiP by Akhremtsev et al. [4], Mt-KaHyPar by Gottesbüren et al. [71] (hypergraphs), KaMinPar by Gottesbüren et al. [74] (deep multilevel), and BiPart by Maleki et al. [120]. The following work is focused on transferring the sequential approaches to shared memory as faithfully as possible, while sometimes incorporating lessons learned from distributed memory.

6.1.1 Coarsening. Most coarsening algorithms are based on greedy matching or greedy clustering. These algorithms visit vertices in some order (e.g., random), calculate the ratings (e.g., heavy-edge) for joining neighboring clusters (or match with neighbor), and then join the highest rated cluster. Visiting vertices in parallel yields faithful parallelizations of the sequential approaches. The challenge is to prevent inconsistent clustering decisions between threads.

Çatalyürek et al. [36] propose parallel schemes for agglomerative clustering and greedy matching that use locking, in addition to a lock-free version of greedy matching that resolves conflicts in a second pass. The lock-based algorithms first try to lock the visited vertex, calculate ratings, and then iterate through the candidates. When a better candidate is found, its lock is tested. If successful, then the old candidate is replaced and unlocked. For the lock-free resolution scheme, the matches are stored in a global array M without protecting write access. After one pass, vertices u with $M[M[u]] \neq u$ are matched with themselves $M[u] \leftarrow u$. These are vertices whose match was visited concurrently and matched to a different vertex. In their evaluation, the lock-based algorithms fare equally well as the sequential versions in terms of cardinality and heavy-edge metric, whereas the algorithm resolving conflicts in a second pass falls off by 10% in the heavy-edge metric.

The resolution-based matching scheme is also used by LaSalle and Karypis [113] in Mt-Metis. On inputs with skewed degree distributions (such as complex networks), maximal matchings are small and thus coarsening converges too slowly. If too few vertices are matched after a pass, then pairs of non-adjacent vertices that have identical neighbors and low degree are matched. If still too few vertices are matched, then any vertex pair that shares a common neighbor can be matched. This technique is dubbed two-hop matching [115].

Akhremtsev et al. [4] instead use parallel size-constrained label propagation clustering [126] to coarsen skewed inputs more effectively. No locks are employed, so cyclic cluster joins may occur. The cluster size constraint ensures that initial partitioning can find a feasible partition. Cluster sizes are updated with atomic instructions. The size constraint is checked before the update, but this offers no atomic consistency. Hence, the instructions' results are checked to guarantee the size constraint (and revert if exceeded). For coarsening it is not necessary to strictly adhere to the size constraint (whereas it is for refinement to guarantee balance). Label propagation is also used in KaMinPar [74], but the atomic check is omitted for coarsening (not for refinement).

For Mt-KaHyPar, Gottesbüren et al. [71] use parallel agglomerative clustering but defer the locking after the rating and target cluster selection, locking only the visited vertex and target cluster. Vertices trying to cyclically join each other are detected and resolved on-the-fly, recursively merging the associated clusters.

6.1.2 Initial Partitioning. For initial partitions, LaSalle and Karypis [113] use recursive bipartitioning. Each thread sequentially computes a bipartition, from which the best one is selected. For recursion, the threads are statically split into two groups, working on the two separate subgraphs. This is later improved [115] to threads cooperating on one bipartition if the graph is sufficiently large instead of independent sequential trials. Akhremtsev et al. [4] compute independent k-way partitions with sequential KaHiP [148]. The downsides of these two approaches are potentially severe load imbalance (recursive bipartitioning) or being a sequential bottleneck. To overcome this, Gottesbüren et al. [71] use a work-stealing task scheduler instead of splitting threads for recursive bipartitioning with parallel (un)coarsening. For flat bipartitions, a portfolio of sequential algorithms is run independently in parallel [87, 155].

The deep multilevel approach [74] offers additional parallelism through repetitions (forks) at different coarsening levels. At these stages the graphs may be too small to make efficient use of parallel (un)coarsening, such that splitting threads off for repetitions comes at little to no running time penalty. Using randomized components (coarsening, initial bipartitioning, refinement) thus offers multiple diversified solutions from which the best is chosen.

Slota et al. [165] use parallel k-source BFS to compute flat k-way partitions. Vertices join the blocks of their parents. Their approach does not use coarsening, which justifies using a flat parallel algorithm at this stage.

Maleki et al. [120] parallelize greedy hypergraph growing [32, 98] for bipartitioning. All vertices are assigned to block V_0. Then the gains of moving the vertices in block V_0 to V_1 are computed. The $\sqrt{|V|}$ highest rated vertices are moved, before the gains of vertices remaining in block V_0 are recomputed. This is repeated until the bipartition is balanced.

6.1.3 Refinement. The two most crucial questions that must be addressed in the refinement phase are how to maintain a balanced partition and how to make sure that concurrent moves do not result in worse cuts. Some refinement algorithms are more difficult to parallelize than others. Unsurprisingly, the more sophisticated techniques, in particular those able to escape local minima, are more difficult to parallelize, as intermediate negative gain moves are required. In particular FM and KL are known to be P-complete [153], which makes the existence of poly-log depth algorithms unlikely.

Label Propagation. The most straightforward approach to parallelize is label propagation: simply visit vertices in parallel. While gains can be incorrect due to concurrent moves in their neighborhood, this does not affect cut optimization too much in practice. Balance can be ensured by using atomic instructions to update block weights [4, 71], or by first collecting moves and approving in a second step [75, 95, 120]. During refinement, it is important to enforce the size constraint by checking the result of the atomic instruction, to guarantee a balanced partition.

Greedy. LaSalle and Karypis [113] parallelize Metis' greedy refinement by statically assigning vertices to threads and running the sequential algorithm on the local vertices. The sequential algorithm performs FM on boundary vertices, but stops once no positive gain moves remain. This eases parallelization, since no moves must be reverted, and thus there is no serial move order to observe. However, it does eliminate the ability to escape local minima. Moves are communicated to other threads of neighboring vertices via message buffers. The threads frequently check their buffers to update local gains. Further, the refinement is split into an upstream and a downstream pass, where vertices are only allowed to move into blocks with higher (respectively, smaller) identifier than their current block. This avoids accidental cut-degrading concurrent moves [97] but restricts the search space, since some blocks quickly become close to overloaded. Balance checks are done optimistically with locally updated block weights. Since this may result in balance violations, synchronization points after passes are used to revert some moves to restore balance.

Parallel Localized FM. Localized FM [148] is a variant of FM that starts with a few boundary vertices and expands its search space to neighbors of moved vertices. It is good at escaping local minima due to allowing negative gain moves, but not wasting too much time on unpromising areas through short search sprints. This approach can be parallelized by performing multiple independent localized searches, as opposed to standard FM, which is difficult to parallelize efficiently [153]. Akhremtsev et al. [4] organize the boundary vertices in a queue that is randomly shuffled. Threads repeatedly poll seed vertices from the queue and perform localized FM around their seeds. Moves are not communicated to other threads. Instead, each thread maintains a local partition – an array for block weights and a hash table for partition IDs. The rationale for keeping moves private is that reverting (negative gain) moves at the end of a localized search confuses other searches. Searches may overlap in their local vertices, but each vertex is moved only once (atomic test-and-set). Once the global queue is empty, the local move sequences of the threads are concatenated into a single sequence. To ensure a balanced partition and no cut degradation, the gains of this sequence are recomputed sequentially, and the prefix that yields the smallest edge cut subject to the balance constraint is applied.

In Mt-KaHyPar, Gottesbüren et al. [71] apply local move sequences to the global partition as soon as a local optimum is found. This provides more accurate information to the other threads. Additionally, the resulting move order more adequately reflects the partition state on which the move decisions of the localized searches are based. Finally, it reduces the memory for hash tables, while keeping the negative gain moves at the end of a localized search private. This is important for hypergraphs where there is more information (e.g., $\Phi(e, i)$) stored in local partition data structures than in graph partitioning. Moves applied to the global partition are rejected if they violate the balance constraint and block weights are maintained with atomic instructions.

Parallel Flow-based Refinement. The flow-based refinement from Section 5.2 offers two parallelism sources that are investigated in Reference [76]. The core routine works on bipartitions, such that it can be applied to different block pairs of a k-way partition in parallel. A basic version only runs non-overlapping block pairs in parallel, but the authors show that overlapping searches are feasible with certain restrictions, and necessary for better parallelism. The second parallelism source is the flow algorithm, where the well-known push-relabel algorithm is nicely amenable to parallelization. The approach is integrated in the state-of-the-art parallel multilevel framework Mt-KaHyPar. Experiments show that the partition quality of the new algorithm is on par with the highest-quality sequential code (KaHyPar), while being an order of magnitude faster when using 10 threads.

Hill Scanning. LaSalle and Karypis extend their greedy refinement to escape local minima to some extent [114] by employing a simplified variant of localized search. Whenever only negative gain moves remain in the thread-local priority queue, a small group of vertices around u (up to 16) is incrementally constructed, with the hope that moving the entire group reduces the cut. The group is constructed in the same way as localized FM expands its search; however, the expansion stops as soon as an improvement is possible. The selected vertices are not restricted to the thread-local ones, as opposed to the greedy algorithm. One restriction is that all vertices must be moved to the same block.

Techniques for Accurate Gains. Gottesbüren et al. [71] double-check gains for localized FM and label propagation using a technique named *attributed gains*. Recall that $\Phi(e, i)$ denotes the number of pins that net e has in block V_i. When vertex u is moved from block V_s to V_t, $\Phi(e, s)$ is (atomically) decremented and $\Phi(e, t)$ is incremented for each $e \in I(u)$. Reducing $\Phi(e, s)$ to zero attributes an $\omega(e)$ connectivity reduction, whereas increasing $\Phi(e, t)$ to one attributes an $\omega(e)$ increase to moving u. If the overall attributed gain of a move is negative, then it is reverted.

The global moves in localized FM are collected in a sequence in the order in which they were applied. As in Mt-KaHiP, Gottesbüren et al. perform a gain recalculation on this sequence in Mt-KaHyPar, for which a parallelization is proposed [71]. Analogously to attributed gains the last pin of a net e moved out of a block is attributed an $\omega(e)$ reduction (if no pin moved in before) and the first pin moved into the block is attributed an $\omega(e)$ increase (if none were contained before). This can be calculated in two passes over the pins of e. Each net incident to a moved vertex is handled independently in parallel, with gain attributions distributed using atomic fetch-and-add.

Finally, the authors [71] show that updating a global gain table [3, 55, 112] can be performed in parallel with atomic instructions. Analogously to attributed gains, gain table updates are triggered by specific observed values when the $\Phi(e, i)$ values are updated. For neighbors of moved vertices their priority queue key is updated with the associated table entries. Further, when a vertex is extracted from the priority queue, its key is checked against the table entries, and reinserted with the new key, if worse. This way, searches gradually update their priority queues to global partition changes without resorting to message passing.

Rebalancing. If vertices are moved concurrently, then it does not suffice to check whether each single move would preserve balance, but rather some synchronization mechanism is necessary. Therefore, some refinement algorithms cannot guarantee balanced partitions, so there is a need for explicit rebalancing algorithms. Furthermore, even if some refinement algorithms can guarantee balance, intermediate balance violations can lead to smaller cuts after a rebalancing step.

Slota et al. [165] use label propagation with gains multiplied by $L_{\max}/|c(V_i)|$ to favor moving into lighter blocks [123]. Maleki et al. [120] use the same approach as their graph growing parallelization for rebalancing two-way partitions, starting with a non-empty lighter block that is gradually filled.

Lasalle and Karypis [113] integrate rebalancing into their parallel greedy refinement. The thread-local move buffers are traversed in reverse order and moves into overloaded blocks are reverted. Each thread is responsible for restoring excess weight proportional to the amount it moved into that block.

For large k, Gottesbüren et al. [74] use one priority queue per overloaded block with the ratio of highest gain (to a non-overloaded block) and vertex weight as key. Each queue is filled with just enough vertices to remove the excess weight, but neighbors of moved vertices in the same former block are inserted, for the case that designated target blocks become close to overloaded. Parallelism is achieved by emptying different overloaded blocks in parallel.

Gottesbüren et al. [71] allow controlled balance violations in the gain recalculation step of localized FM in Mt-KaHyPar. Label propagation on the next level is often able to rebalance, and explicit rebalancing similar to label propagation is employed on the finest level.

Parallel n-level (Un)Coarsening. Gottesbüren et al. [73] propose an n-level version [3, 135] of Mt-KaHyPar [71]. With sequential n-level, only one vertex is (un)contracted at a time, which is inherently sequential but offers high solution quality through highly localized refinement. In the parallel version, vertex pairs are contracted asynchronously with vertices to be contracted organized in a hierarchical forest data structure. The forest yields precedence conditions (bottom up) for the asynchronous contractions. With this, vertices in different subtrees can be contracted independently in parallel, as soon as their children are finished. Fine-grained locking is employed to edit the hypergraph data structure, and dynamically maintain consistency of the forest.

Uncoarsening introduces parallelism by uncontracting batches of $b > 1$ vertices in parallel. The batches are constructed by traversing the forest in top-down order, assembling contractions that can be reverted independently in a batch. Uncontracting a batch resolves the last dependencies required to uncontract the next batch. The vertices of the current batch serve as seeds for highly localized parallel refinement (label propagation and FM).

Determinism. Researchers have advocated the benefits of deterministic parallel algorithms for several decades [25, 168], including ease of debugging, reasoning about performance, and reproduciblity. A downside is less flexibility in terms of algorithm design choices and potential overheads for synchronizing optimization decisions. For example, the deterministic version of Mt-KaHyPar is a factor of 1.16 slower than an equivalent non-deterministic configuration, and exhibits a factor of 1.029 higher connectivity (both aggregated using geometric mean). Interestingly, this degradation stems from coarsening [75], not refinement.

Maleki et al. [120] propose BiPart [120], a deterministic multilevel recursive bipartitioning algorithm. For coarsening, vertices are matched to their smallest net with ID hashes as tie breakers. All vertices matched to the same net are contracted without restricting coarse vertex weights. Initial partitioning uses the parallel greedy graph growing described above. For refinement, the gain for moving each vertex to the opposite block is computed in parallel. Let l_0, l_1 denote the number of vertices with positive gain in block V_0, V_1. Then the $\min(l_0, l_1)$ vertices with highest gains are swapped, an approach that was already proposed for Social Hash [95]. If the hypergraph has unit vertex weights, then this maintains a balanced partition. However, this algorithm is used within the multilevel framework, where non-uniform vertex weights occur from coarsening. Thus, the graph growing rebalancing described above is employed. Note that gains become stale after a neighbor is moved, and thus the overall cut reduction is not the sum of the gains.

Gottesbüren and Hamann [75] present a deterministic version of Mt-KaHyPar [71]. The label propagation style algorithms for preprocessing, coarsening and refinement of Mt-KaHyPar are re-implemented in a deterministic fashion using synchronous local moving [81]. Move decisions do not depend on other moves in the current round. Rounds are further split into sub-rounds to incorporate more up-to-date information. After each sub-round some of the moves are approved and some are denied, for example due to the balance constraint or a cluster size constraint for coarsening. For refinement, the same method as BiPart [120] and SocialHash [95] is used: swapping highest gain prefixes between block pairs. To incorporate non-unit vertex weights, the cumulative weights of all prefixes of the move sequences are computed. The best prefix combination is then selected similar to a parallel merge.

6.2 Distributed Memory

Distributed (hyper)graph partitioning is one way to handle large inputs that do not fit into the main memory of a single machine. In the distributed-memory model, several processors (PEs) are interconnected via a communication network, and each has its private memory inaccessible to others. Computational tasks on each PE usually operate independently only on local data representing a small subset of the input. Intermediate computational results must be exchanged via dedicated network communication primitives.

Distributed (hyper)graph processing algorithms require that the vertices and edges of the input are partitioned among the processors. Since many applications use balanced (hyper)graph partitioning to obtain a good initial assignment, much simpler techniques are used in distributed partitioners. There exist range-based [128] and hash-based partitioning techniques [166, 167]. The former splits the vertex IDs into equidistant ranges, which are then assigned to the PEs. Since both techniques do not consider the (hyper)graph structure, it could lead to load imbalances or high communication overheads. However, one could also migrate vertices as more information about the structure of the (hyper)graph is available, e.g., when recursing on a subgraph obtained via recursive bipartitioning [22, 45]. If geometric information is available, then one can also use space-filling curves [16, 177].

Each PE then stores the vertices assigned to it and the edges incident to them. The edges stored on a PE can be incident to local vertices or vertices on other PEs (also called *ghost* or *halo* vertices). We say that a PE is adjacent to another PE if they share a common edge. Processors must be able to identify adjacent PEs to propagate updates, e.g., if we move a vertex to a different block, then we have to communicate that change to other PEs in the network such that local search algorithms can work on accurate partition information. However, each communication operation introduces overheads that can limit the scalability of the system. Thus, the main challenge in distributed (hyper)graph partitioning is keeping the global partition information on each PE in some sense up to date while simultaneously minimizing the required communication.

The remainder of this section describes the algorithmic core ideas of recent publications in that field and abstracts from the physical placement of the vertices and the actual representation of the distributed (hyper)graph data structure. However, we assume that each vertex knows on which PE its neighbors are stored.

Local Search. The label propagation heuristic is the most widely used local search algorithm in distributed systems [46, 95, 97, 123, 128, 166, 170, 180]. Other approaches schedule sequential two-way FM [55] on adjacent block pairs in parallel [40, 92, 106]. However, this limits the available parallelism to at most the number of blocks k.

Parallel label propagation implementations mostly follow the *bulk synchronous parallel* model. In a computation phase, each PE computes for its local vertices their desired target block. In the communication phase, updates are made visible to other PEs via personalized all-to-all communication [128, 166]. Meyerhenke et al. [128] use an asynchronous communication model. If the computation phase of a PE ends, then it sends and receives updates to and from other PEs and immediately continues with the next round.

In the parallel setting, the move gain of two adjacent vertices may suggest an improvement when moved individually, but moving both simultaneously may worsen the solution quality. Therefore, some partitioners use a vertex coloring [97] or a two-phase protocol where in the first phase, vertices can only move from a block V_i to V_j if $i < j$ and vice versa in the second phase [46, 113, 170]. Many systems do not use any techniques to protect against move conflicts. This can be seen as an optimistic strategy assuming that conflicts rarely happen in practice.

The *Social Hash Partitioner* [95] (Facebook's internal hypergraph partitioner) also uses the label propagation heuristic to optimize $\text{fanout}(\Pi) := \frac{1}{|E|}\sum_{e \in E} \lambda(e)$ where $\Pi = \{V_1, \ldots, V_k\}$ is a k-way partition. The authors note that the label propagation algorithm can easily get stuck in local optima for fanout optimization and suggest a probabilistic version of the fanout metric, called $\text{p-fanout}(\Pi) := \frac{1}{|E|}\sum_{e \in E}\sum_{V_i \in \Pi} 1 - (1-p)^{\Phi(e, V_i)}$ for some probability $p \in (0, 1)$. The probabilistic fanout function samples the pins of net with probability p and represents the expected fanout for a family of similar hypergraphs. Thus, it should be more robust and reduce the impact of local minima.

Other recently published distributed local search techniques are based on vertex swapping techniques that preserve the balance of the partition. Rahimian et al. [143] present JA-BE-JA that uses such an approach. The algorithm iterates over the local vertices of each PE and for each vertex, it considers all adjacent vertices as swap candidates. If no partner was found, then it selects a random vertex from a sample as a candidate. If the selected vertex is assigned to a different PE, then the instantiating PE sends a request with all the required information such that the receiving PE can verify whether or not the swap operation would improve the edge cut. On success, both vertices change their blocks. Additionally, simulated annealing is used to avoid local minima.

Aydin et al. [16] implement a distributed partitioner that computes a linear ordering of the vertices, which is then split into k equally sized ranges to obtain an initial k-way partition. The idea is similar to space-filling curves [19, 138], but does not require geometric information. The initial ordering is computed by assigning labels to a tree constructed via agglomerative hierarchical clustering. Afterward, it sorts the labels of the leaves to obtain an initial ordering. To further improve the ordering, it solves the *minimum linear arrangement* problem that tries to optimize $\sum_{(u,v)\in E} |\pi(u) - \pi(v)|\omega(u,v)$ where $\pi(u)$ denotes the position of $u \in V$ in the current ordering. To do so, it uses a two-stage MapReduce algorithm that is repeated until convergence: First, each vertex computes its desired new position as the weighted median of its neighbor's positions. Second, the final positions are assigned to the vertices by resolving duplicates with simple ID-based ordering. The second local search algorithm performs vertex swaps. First, it pairs adjacent blocks of the partition. Then, it splits the vertices of each block into r disjoint intervals and randomly pairs intervals between paired blocks. The paired sets are then mapped to the processors that perform the following algorithm: It sorts the vertices in both sets according to their cut reduction if moved to the opposite block and swaps the vertices with the highest combined cut reduction.

Balance Constraint. The label propagation algorithm only knows the exact block weights at the beginning of each computation phase. In the computation phase, block weights are only maintained locally. In the communication phase, the combination of all moves may result in a partition that violates the balance constraint. Thus, partitioners based on this scheme have to employ techniques to ensure balance.

The distributed multilevel graph partitioner ParHIP [128] divides a label propagation round into subrounds and restores the exact block weights with an All-Reduce operation after each subround. Note that this does not guarantee balance but gives a good approximation of the block weights when the number of moved vertices in a subround is small.

Slota et al. [166] implemented a distributed graph partitioner that alternates between a balance and refinement phase, both utilizing the label propagation algorithm. In the refinement phase, each PE maintains approximate block weights $a(V_i) := c(V_i) + \gamma\Delta(V_i)$ where $c(V_i)$ is the weight of block V_i at the beginning of the computation phase, $\Delta(V_i)$ is the weight of vertices that locally moved out, respectively, into block V_i, and γ is a tuning parameter that depends on the number of PEs. Each PE then ensures locally that $a(V_i) \leq L_{\max}$ for all $i \in \{1, \ldots, k\}$. In the balancing phase, the gain of moving a vertex to block V_i is multiplied with $\frac{L_{\max}}{a(V_i)}$. As a consequence, moves to underloaded blocks become more attractive. In a subsequent publication [167], the approach is generalized to the multi-constraint partitioning problem, where each vertex is associated with multiple weights.

Recently, probabilistic methods were proposed that preserve the balance in expectation [95, 123]. The Social Hash Partitioner [95] aggregates the number of vertices $S_{i,j}$ that want to move from block V_i to V_j after each computation phase at a dedicated master process. Then, a vertex part of block V_i is moved to its desired target block V_j with probability $\frac{\min(S_{i,j}, S_{j,i})}{S_{i,j}}$. This ensures that the expected number of vertices that move from block V_i to V_j and vice versa is the same and, thus, preserves the balance of the partition in expectation. However, each PE moves its highest ranked

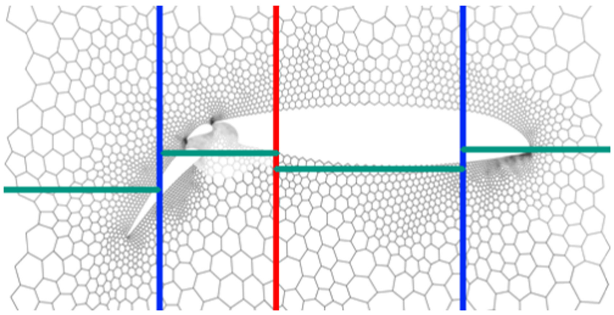

Fig. 7. A visualization of an airfoil. The graph has coordinates for each vertex. Geometric partitioning algorithms use this type of information for partitioning. In the example above, recursive coordinated bisection (always split the graph along the shorter axis) has been applied to derive a partition.

vertices with probability one and all remaining probabilistically. Martella et al. [123] moves a vertex u to its desired target block V_j with probability $\frac{L_{\max}-c(V_j)}{M_j}$ where M_j are the number of vertices that want to move to block V_j. The advantage of the probabilistic method is that only the number of vertices preferring a different block need to be communicated instead of all moves.

Multilevel Algorithms. Although it is widely known that multilevel algorithms produce better partitions than flat partitioning schemes, the systems used in industry, e.g., at Google [16] or Facebook [95, 123], are primarily non-multilevel algorithms. The main reason for this is that the scalability of multilevel algorithms is often limited to a few hundred processors [92, 106]. Furthermore, most parallel multilevel systems implement matching-based coarsening algorithms [40, 46, 92, 106, 170, 181] that are not capable to efficiently reduce the size of today's complex networks (power-law node degree distribution). The most prominent distributed multilevel algorithms are Jostle [181], ParMetis [97], PT-Scotch [40], KaPPa [92], ParHIP [128] and ScalaPart [106] for graph, and Parkway [170] and Zoltan [46] for hypergraph partitioning.

Meyerhenke et al. [128] build the parallel multilevel partitioner ParHIP that uses a parallel version of the size-constraint label propagation algorithm [126]. The algorithm is used to compute a clustering in the coarsening phase and as a local search algorithm in the refinement phase. To obtain an initial partition of the coarsest graph, it uses the distributed evolutionary graph partitioner KaFFPaE [149]. On complex networks, ParHIP computes edge cuts 38% smaller than those of ParMetis [97] on average, while it is also more than a factor of two faster.

Wang et al. [182] use a similar approach that also utilizes the label propagation algorithm to compute a clustering in the coarsening phase. The algorithm is implemented on top of Microsoft's Trinity graph engine [161]. The partitioner additionally uses external memory techniques to partition large graphs on a small number of machines. However, it does not perform multilevel refinement (the initial partition is projected to the input graph).

Geometric Partitioners. Many graphs are derived from geometric applications and are enriched with coordinate information (e.g., each vertex is associated with a d-dimensional point). A mesh with coordinate information and a partition based on these coordinates is shown in Figure 7. Geometric partitioning techniques use this information to partition the corresponding point set into k equally sized clusters while minimizing an objective function defined on the clusters. The objective function should be chosen such that it implicitly optimizes the desired graph partitioning metric (e.g., the sum of the lengths of all bounding boxes approximates the total communication volume [45]). Since geometric methods ignore the underlying structure of the (hyper)graph, the quality of the partitions is often inferior compared to traditional multilevel algorithms. However, these algorithms are often simpler leading to faster and more scalable algorithms. Prominent techniques use space-filling curves [19, 138] that map a set of d-dimensional points to a one-dimensional line. A fundamental property of this curve is that points that are close on the line are also close in the original space. Other approaches recursively divide the space via cutting planes such as Octree-based partitioning [129], recursive coordinate bisection [22, 164], and recursive inertial bisection [169, 184]. The MultiJagged algorithm of Deveci et al. [45] uses multisection rather than bisection to reduce the depth of the recursion and speed up computation relative to recursive coordinate bisection; its hybrid implementation uses MPI and Kokkos [171] to support both distributed-memory message passing between PEs and multithreading or GPU computation within PEs.

Recently, Von Looz et al. [177] presented a scalable balanced k-means algorithm to partition geometric graphs. The k-means problem asks for a partition of a point set P into k roughly equally sized clusters such that the squared distances of each point to the mean of its cluster is minimized (in the following also referred to as the center of a cluster). Clusters obtained with this problem definition tend to have better shapes than computed with previous methods and also produce better partitions when measured with graph metrics [125]. They present a parallel implementation of Lloyd's greedy algorithm [118] that repeats the following steps until convergence. First, each point $p \in P$ is assigned to the cluster that minimizes the distance of p to its center. Afterwards, the center of each cluster is updated by calculating the arithmetic mean of all points assigned to it. To achieve balanced cluster sizes, an influence factor γ_c is introduced individually for each cluster c and $\text{eff\_dist}(p, \text{center}(c)) := \text{dist}(p, \text{center}(c))/\gamma_c$ is used as the distance of point p to the center of a cluster c. If a cluster c becomes overloaded, then the influence factor γ_c is decreased; otherwise, it is increased. Thus, underloaded clusters become more attractive. The implementation replicates the cluster centers and influence factors globally and after each computation phase, it updates the values via a parallel sum operation. To obtain an initial solution, it sorts the points according to the index on a space-filling curve and splits the order into k equally sized clusters. Furthermore, it establishes a lower bound for the distances of each point to the second-closest cluster, which allows us to skip expensive distance computations for most of the points. Additionally, each processor sorts the cluster centers according to their distances to a bounding box around the process-local points. Evaluating the target clusters in increasing distance order allows us to abort early when the minimum distance of the remaining clusters is above the distance of already found candidates.

Scalable Edge Partitioning. Schlag et al. [156] present a distributed algorithm to solve the edge partitioning problem. The edge partitioning problem asks for a partition $\Pi = \{E_1, \ldots, E_k\}$ of the edge set into k blocks each containing roughly the same number of edges, while minimizing the vertex cut $\sum_{v \in V} \rho(v) - 1$ where $\rho(v) = |\{E_i \mid E_i \in \Pi : I(v) \cap E_i \neq \emptyset\}|$. They evaluated two methods to solve the problem. The first transforms the graph into its dual hypergraph representation (edges of the graph become vertices of the hypergraph and each vertex of the graph induces a net spanning its incident edges). Using a hypergraph partitioner that optimizes the connectivity metric to partition the vertex set directly optimizes the vertex cut of the underlying edge partitioning problem. The second method uses a distributed construction algorithm of the so-called ***split-and-connect*** **(SPAC)** graph. For each vertex u, it inserts $d(u)$ auxiliary vertices into the SPAC graph and connects them to a cycle using auxiliary edges each with weight one. Each auxiliary vertex is a representative for exactly one incident edge of u. For each edge $(u, v) \in E$, it adds an infinite weight edge between the two representatives of the corresponding edge. Thus, a partition of the vertex set of the SPAC graph cannot cut an edge connecting two representatives. Therefore, such a partition can be transformed into an edge partition by assigning each edge to the block of its representatives. In the evaluation, they compare both representations while using different graph and hypergraph partitioners. The results showed that parallel graph partitioners outperform distributed hypergraph partitioners. However, the sequential hypergraph partitioner KaHyPar [154] produces significantly better vertex cuts than all other approaches (more than 20% better than the best graph-based approach), but is an order of magnitude slower than the evaluated distributed algorithms.

6.3 GPU

Due to their high computational power, modern GPUs have become an important tool for accelerating data-parallel applications. However, due to the highly irregular structure of graphs, it remains challenging to design graph algorithms that efficiently utilize the SIMD architecture of modern GPUs.

Multilevel Graph Partitioning. Goodarzi et al. [67, 68] present two algorithms for GPU-based multilevel graph partitioning. Their earlier approach [67] uses heavy-edge matching for coarsening and transfers the coarsest graph onto CPU for initial partitioning (using Mt-Metis [96]). During refinement, vertices are distributed among threads and each thread finds the blocks maximizing the gain values of its assigned vertices. To prevent conflicting moves that worsen the edge cut in combination, refinement alternates between rounds in which only moves to blocks with increasing (respectively, decreasing) block IDs are considered. For each block, potential moves to the block are collected in a global buffer, which is then sorted, and the highest rated moves are executed.

Their later approach [68] brings several improvements. First, the authors use Warp Segmentation [103] to improve the efficiency of the heavy-matching computation during coarsening. Initial partitioning is then performed on the GPU using a greedy growing technique. During refinement, vertices are once more divided among threads, and each thread finds the blocks maximizing the gain of its assigned boundary vertices and collects the potential moves in a global buffer. Then, the algorithm finds the highest rated ℓ moves in the global buffer, for some small input constant ℓ. Since moves might conflict with each other, their algorithm distributes all 2^ℓ move combinations across thread groups and finds the best combination, which is then applied to the graph partition. This process is repeated until the global buffer is empty. On average, their GPU-based approach is approximately 1.9 times faster than Mt-Metis while computing slightly worse edge cuts across their benchmark set of 16 graphs.

In his PhD thesis, Fagginer Auer [15] develops two multilevel algorithms running on a GPU. One that uses spectral refinement and one that uses greedy refinement. Later, Fagginger Auer and Bisseling [53] present a fine-grained shared-memory parallel algorithm for graph coarsening and apply this algorithm in the context of graph clustering to obtain a fast greedy heuristic for maximising modularity in weighted undirected graphs. The algorithm is suitable for both multi-core CPUs and GPUs. Later, Gilbert et al. [63] present performance-portable graph coarsening algorithms. In particular, the authors study a GPU parallelization of the heavy edge coarsening method. The authors evaluate their coarsening method using a multilevel spectral graph partitioning algorithm as primary use case.

Spectral Graph Partitioning. The availability of efficient eigensolvers on GPUs has led to a recent re-emergance of spectral techniques for graph partitioning on GPU systems [1, 2, 133]. These techniques were first developed by Donath and Hoffman [50, 51] and Fiedler [56] to compute graph bisections in the 1970s. Subsequently, these techniques have been improved [20, 27, 83, 140, 164] and extended to partition a graph into more than two blocks using multiple eigenvectors [8, 83].

Naumov and Moon [133] present an implementation of spectral graph partitioning for single GPU systems as part of the nvGRAPH library, whereas Acer et al. [1, 2] propose the multi-GPU implementation Sphynx. Both partitioners precondition the matrix and use the LOBPCG [107] eigenvalue solver. The eigenvectors are then used to embed the graph into a multidimensional coordinate space, which is then used to derive a partition of the graph. In nvGRAPH, this is done using a k-means clustering algorithm on the embedded graph, whereas Sphynx uses the geometric graph partitioner Multi-Jagged [45], which supports multi-GPU systems. Since the approach by Acer et al. outperforms nvGRAPH in terms of partition balance, cut size and running time (when run on a single GPU system), we focus on their experimental evaluation. To this end, when Sphynx is compared against ParMETIS [100], ParMETIS generally obtains significantly better cuts than Sphynx—approximately 20% (respectively, 70%) lower cuts on regular (respectively, irregular) graph instances). On irregular graphs, the authors report a significant speedup of approximately 19 using Sphynx on 24 GPUs compared to ParMETIS with 168 MPI processes across four compute nodes, although ParMETIS is approximately three times faster than Sphynx on regular graphs even when using a single CPU core for each GPU used by Sphynx. Additionally, Acer et al. report the influence of several matrix preconditioners on different classes of graphs.

6.4 Approaches for Other Types of Hardware

Recently, Ushijima-Mwesigwa et al. [175] explore graph partitioning using quantum annealing on the D-Wave 2X machine. The main idea is to formulate the graph partitioning problem as a quadratic unconstraint binary optimization problem via a spectral approach. These problems can be mapped to minimizing Ising objective functions, which the D-wave system can minimize. The method could run directly on the D-Wave system for small graphs and used a hybrid approach for large graphs.

Lio et al. [117] experiment with solving the graph partitioning problem on the Fujitsu Digital Annealer, which is a special-purpose hardware designed for solving combinatorial optimization problems. In particular, the authors also model the problem as a quadratic unconstrained binary optimization problem via a spectral approach.. The Fujitsu Digital Annealer then utilizes application-specific integrated circuit hardware for solving fully connected quadratic unconstraint binary optimization problems. The authors identify a dense network for which their approach significantly outperforms KaffpaE. Then a range of similar instances is created using the MUSKETEER [79] framework. On most of those instances, their approach outperforms KaffpaE and Gurobi.

7 EXPERIMENTAL METHODOLOGY

As more applications of (hyper)graph partitioning have emerged in recent years, it becomes increasingly important to evaluate algorithms on many (hyper)graph instances to demonstrate their effectiveness for practical applications. The presentation of results using tables comparing the running time and solution quality of different algorithms can quickly become difficult to interpret, while evaluations based on aggregated numbers can lead to misleading conclusions. This section discusses several alternative methods helpful for presenting and interpretting experimental results.

Performance Profile. Performance profiles can be used to compare the solution quality of different algorithms [48]. Let $\mathcal{A}$ be the set of all algorithms, $\mathcal{I}$ the set of instances, and $q_A(I)$ the value of a quality metric (e.g., edge-cut) of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, performance profiles show the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \text{Best}(I)$, where τ is on the x-axis and $\text{Best}(I) := \min_{A' \in \mathcal{A}} q_{A'}(I)$ is the best solution produced by an algorithm $A \in \mathcal{A}$ on an instance $I \in \mathcal{I}$. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm $A \in \mathcal{A}$ performs best. Achieving higher fractions at smaller τ values is considered better.

Figure 8 (left) compares a quality metric of four different algorithms using a performance profile. Algorithms B and C compute on roughly 40% of the instances the best solutions while Algorithms

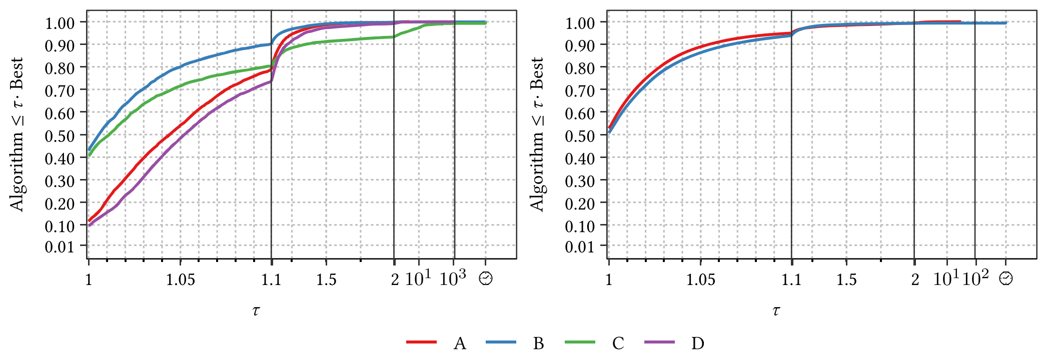

Fig. 8. Performance profiles comparing four different algorithms (left) and the result of a effectiveness test between two algorithms (right). The ☹ tick marks instances for which the corresponding algorithm has computed an infeasible solution or has ran into a time limit.

A and D only on 10% (see $\tau = 1$). The solutions produced by Algorithm A are worse than the best solutions by $\approx$ 4% in the median (intersection of $y = 0.5$ with the red line is at $\tau \approx 1.04$). If we compare Algorithms C and D based on their geometric mean of the quality metric, then we would observe that Algorithm C is only 0.2% better than Algorithm D. Hence, we would probably conclude that there is no significant difference between both. If we look at the performance profile, then we see that Algorithm C is on most of the instances closer to the best solution than Algorithm D. However, on $\approx$ 10% of the instances the solutions are worse than the best one by more than a factor of two, which has a large influence on its geometric mean (see green line at $\tau = 2$).

Effectiveness Tests. The performance profile in Figure 8 (left) suggests that Algorithm B produces solutions that are better than those of Algorithm A. However, if we look at their running times, we observe that Algorithm A is more than an order of magnitude faster than Algorithm B on average. Thus, Algorithm B may have an unfair advantage due to its longer running time. Therefore, Akhremtsev et al. [4] introduces *effectiveness tests* to compare solution quality when two algorithms are given a similar running time, by performing additional repetitions with the faster algorithm. Consider Algorithms A and B and an instance I. Effectiveness tests first sample one run of both algorithms. Let t_A^1 and t_B^1 be their running times and assume that $t_A^1 \geq t_B^1$. Then, additional runs of Algorithm B are sampled until the accumulated time exceeds t_A^1. Let $t_B^2, \ldots, t_B^l$ denote their running times. The last run is accepted with probability $(t_A^1 - \sum_{i=1}^{l-1} t_B^i)/t_B^l$ so that the expected time for the runs of B equals t_A^1. The quality metric taken is the minimum out of all runs.

Figure 8 (right) shows the result of the effectiveness test for Algorithms A and B using a performance profile. The plot shows that there is no significant difference between the two algorithms.

Significance Tests. It is not immediately obvious from the performance profile in Figure 8 (left) whether Algorithm B performs better than Algorithm C or vice versa, since both lines are close to each other. Here, statistical tests can give further information that helps to decide whether the difference between two algorithms is statistically significant.

A widely used technique is to formulate a *null hypothesis* assuming that there is no difference between the observed distributions. A significance test then calculates the probability (p-value) that the observed distribution occurs under the assumption that the null hypothesis is true [191]. The null hypothesis is rejected if the probability is below a predefined significance level α (e.g., $\alpha = 5\%$ as suggested by Fisher [59]). To compute the p-value, the Wilcoxon signed rank test [183] is often used to compare two algorithms while the Friedman test [60] makes pairwise comparisons between multiple algorithms.

Significance tests based on a null hypothesis decide if the difference between two measurements is statistically significant but do not reveal any information whether or not the difference is relevant in practice. For example, a significance test comparing the running times Algorithms A and B may conclude that Algorithm A is faster than Algorithm B. However, Algorithm A might be only 1% faster on average and, thus, the relevance of the improvement is questionable. Therefore, Angriman et al. [12] recommend using *parameter estimation* instead of hypothesis testing. To apply parameter estimation to our example, we could compute the running time ratios $t_A(I)/t_B(I)$ for each instance I and determine the *confidence interval* that contains 95% of the measurements. The confidence interval gives additional information on the *effect size* of the improvement.

8 FUTURE CHALLENGES

While there has been a considerable progress in the field in the last decade, there is a wide range of challenges that remain open. It is an interesting question to what extent the multitude of results sketched above have reached a state of maturity where future improvements become less and less likely. Algorithms like multi-threaded graph partitioning for balanced graph partitioning using a small number of threads and standard inputs may be difficult to improve. Long time open problems like large gaps between theory and practice may remain open for a long time. However, other important issues have considerable potential. We try to identify some of them below.

Parallelism and Other Hardware Issues. Scalability of high-quality parallel (hyper)graph partitioning remains an active area of research. In particular, achieving good scalability and quality on large distributed-memory machines is still a challenge, but even on shared-memory machines, scalability to a large number of threads seems difficult. Even more difficult is aligning the inherent complexity and irregularity of state-of-the-art algorithms with the restrictions of GPUs or SIMD-instructions. Another conundrum is that, for good memory access locality during partitioning, (hyper)graphs need to already be partitioned reasonably well.

Hierarchies of supercomputers have to be taken into account during partitioning. This can be done by using multi-recursive approaches taking the system hierarchy into account or by adapting the deep multilevel partitioning approach sketched above to the distributed-memory case. When arriving at a compute-node level, additional techniques are necessary to employ the full capabilities of a parallel supercomputer. For example, many of those machines have GPU's on a node level. Recently, researchers started to develop partitioning algorithms that run on GPUs and while of independent interest, partitioning algorithms developed for this type of hardware can help in that regard. Hence, future parallel algorithms have to compute partitions on and for heterogeneous machines. However, algorithms should be energy-efficient and performance per watt has to be considered. Lastly, future hardware platforms have to be taken into consideration when developing such algorithms. One way to achieve this will be to use performance portable programming ecosystems like the Kokkos library [171].

Problem Variations. Current partitioning algorithms work well on a wide range of instances. Most of these instances are either unweighted and translate into well behaved weighted problems if a multilevel algorithm is used or instances have a fairly even distribution of node weights. For instances with vastly different values of node weights, partitioning problems discussed in this article get close to bin packing problems. Currently solvers are not able to handle this case well.

Algorithms for the well established objective functions like edge cut (in the graph case) or connectivity/cut (in the hypergraph case) are able to compute very high-quality partitions. Many open problems remain when optimizing for other or multiple objectives and when the problem has other/multiple constraints. For example, in parallel computing, bottleneck objectives such as minimizing the maximum edge cut of a block should perform better, but few partitioners optimize for such objectives. The same is true for high-quality solvers for problems with multiple constraints in the case of repartitioning. Streaming algorithms currently compute much worse quality partitions than internal memory partitioning algorithms. This is partially due to the fact that such algorithms do not have a global view of the optimization problem. Improving quality of streaming algorithms is an open problem. Last, we believe that directed variants of the problems will become more important to model different applications.

Multilevel Partitioning. The multilevel technique has been incredibly successful in the field of decomposition. Yet, multilevel algorithms in the area still consist in practice of a very limited number of multilevel techniques. Development and understanding components of more sophisticated coarsening schemes, edge ratings, and metrics of nodes' similarity that can be propagated throughout the hierarchies are among the future challenges for (hyper)graph partitioning. Additional challenges occur in attempt of their rigorous analysis. For example, coarsening at the moment is typically stopped based on simple formulas. However, in practice it may be much more fruitful to derive stopping rules for coarsening that take the given instance into account. This also translates to different parameters of sub-algorithms that should be chosen based on the type of input that is provided to the algorithm.

Modeling and Experiments. Traditionally instances for partitioning algorithms came from scientific simulations and thus were fairly well structured. For these networks, edge cut and communication volume of the application have been highly correlated. However, for instances having much more complex structures like social networks that recently became important in practice, the correlation is not that high. It remains an open question how different models of (hyper)graph partitioning translate into concrete performance of applications.

ACKNOWLEDGMENTS

We thank Ilya Safro for feedback and additional references on an early version of the manuscript.

REFERENCES

[1] Seher Acer, Erik G. Boman, Christian A. Glusa, and Sivasankaran Rajamanickam. 2021. 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] Seher Acer, Erik G. Boman, and Sivasankaran Rajamanickam. 2020. SPHYNX: Spectral partitioning for hybrid and axelerator-enabled systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'20).* 440–449. https://doi.org/10.1109/IPDPSW50202.2020.00082

[3] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a direct k-way hypergraph partitioning algorithm. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX'17).* SIAM, 28–42. https://doi.org/10.1137/1.9781611974768.3

[4] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2020. High-quality shared-memory graph partitioning. *IEEE Trans. Parallel Distrib. Syst.* 31, 11 (2020), 2710–2722. https://doi.org/10.1109/TPDS.2020.3001645

[5] Charles J. Alpert. 1998. The ISPD98 circuit benchmark suite. In *Proceedings of the International Symposium on Physical Design (ISPD'98)*, Majid Sarrafzadeh (Ed.). ACM, 80–85. https://doi.org/10.1145/274535.274546

[6] C. J. Alpert, J.-H. Huang, and A. B. Kahng. 1998. Multilevel circuit partitioning. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 17, 8 (1998), 655–667. https://doi.org/10.1109/43.712098

[7] C. J. Alpert and A. B. Kahng. 1995. Recent directions in netlist partitioning: A survey. *Integr. VLSI J.* 19, 1-2 (1995), 1–81.

[8] Charles J. Alpert and So-Zen Yao. 1995. Spectral partitioning: The more eigenvectors, the better. In *Proceedings of the 32nd Conference on Design Automation.* ACM Press, 195–200. https://doi.org/10.1145/217474.217529

[9] Zhao An, Qilong Feng, Iyad Kanj, and Ge Xia. 2020. The complexity of tree partitioning. *Algorithmica* 82, 9 (2020), 2606–2643.

[10] Robin Andre, Sebastian Schlag, and Christian Schulz. 2018. Memetic multilevel hypergraph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'18).* ACM, 347–354. https://doi.org/10.1145/3205455.3205475

[11] K. Andreev and H. Räcke. 2006. Balanced graph partitioning. *Theory Comput. Syst.* 39, 6 (2006), 929–939. https://doi.org/10.1007/s00224-006-1350-7

[12] Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. 2019. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms* 12, 7 (2019), 127. https://doi.org/10.3390/a12070127

[13] Aaron Archer, Kevin Aydin, MohammadHossein Bateni, Vahab S. Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. 2019. Cache-aware load balancing of data center applications. *Proc. VLDB Endow.* 12, 6 (2019), 709–723. https://doi.org/10.14778/3311880.3311887

[14] Ankita Atrey, Gregory Van Seghbroeck, Higinio Mora, Bruno Volckaert, and Filip De Turck. 2020. UnifyDR: A generic framework for unifying data and replica placement. *IEEE Access* 8 (2020), 216894–216910. https://doi.org/10.1109/ACCESS.2020.3041670

[15] B. O. Fagginger Auer. 2013. *GPU Acceleration of Graph Matching, Clustering, and Partitioning*. Ph.D. Dissertation. Utrecht University, Netherlands. Retrieved from http://dspace.library.uu.nl/handle/1874/278892.

[16] Kevin Aydin, MohammadHossein Bateni, and Vahab S. Mirrokni. 2019. Distributed balanced partitioning via linear embedding. *Algorithms* 12, 8 (2019), 162. https://doi.org/10.3390/a12080162

[17] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. 2008. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.* 68, 5 (2008), 609–625. https://doi.org/10.1016/j.jpdc.2007.09.006

[18] Thomas Bäck. 1996. *Evolutionary Algorithms in Theory and Practice—Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press.

[19] Michael Bader. 2013. *Space-Filling Curves—An Introduction with Applications in Scientific Computing*. Texts in Computational Science and Engineering, Vol. 9. Springer. https://doi.org/10.1007/978-3-642-31046-1

[20] S. T. Barnard and H. D. Simon. 1993. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*. 711–718.

[21] Una Benlic and Jin-Kao Hao. 2010. An effective multilevel memetic algorithm for balanced graph partitioning. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10)*. IEEE Computer Society, 121–128. https://doi.org/10.1109/ICTAI.2010.25

[22] M. J. Berger and S. H. Bokhari. 1987. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.* 100, 5 (1987), 570–580.

[23] C. Bichot and P. Siarry (Eds.). 2011. *Graph Partitioning*. Wiley. https://doi.org/10.1002/9781118601181

[24] Rob H. Bisseling and Wouter Meesen. 2005. Communication balancing in parallel sparse matrix-vector multiplication. *Electr. Trans. Numer. Anal.* 21 (2005), 47–65. http://eudml.org/doc/128024.

[25] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. https://doi.org/10.1145/2145816.2145840

[26] Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *J. Stat. Mech.: Theory Exper.* 2008, 10 (2008). https://doi.org/10.1088/1742-5468/2008/10/p10008

[27] R. B. Boppana. 1987. Eigenvalues and graph bisection: An average-case analysis (extended abstract). In *Proceedings of the 28th Symposium on Foundations of Computer Science*. 280–285.

[28] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, 1456–1465. https://doi.org/10.1145/2623330.2623660

[29] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent Advances in Graph Partitioning*. Springer International Publishing, Cham, 117–158. https://doi.org/10.1007/978-3-319-49487-6_4

[30] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. 2000. Improved algorithms for hypergraph bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'00)*. ACM, 661–666. https://doi.org/10.1145/368434.368864

[31] Andrew. E. Caldwell, Andrew B. Kahng, and Igor L. Markov. 2000. Optimal partitioners and end-case placers for standard-cell layout. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* 19, 11 (2000), 1304–1313. https://doi.org/10.1109/43.892854

[32] Ü. V. Catalyürek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.* 10, 7 (July 1999), 673–693. https://doi.org/10.1109/71.780863

[33] Ü. V. Çatalyürek and C. Aykanat. 2011. PaToH: Partitioning Tool for Hypergraphs. Retrieved from https://www.cc.gatech.edu/~umit/PaToH/manual.pdf.

[34] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. 2010. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.* 32, 2 (2010), 656–683. https://doi.org/10.1137/080737770

[35] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy, and Lee Ann Riesen. 2009. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.* 69, 8 (2009), 711–724.

[36] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. 2012. Multithreaded clustering for multi-level hypergraph partitioning. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12).* IEEE Computer Society, 848–859. https://doi.org/10.1109/IPDPS.2012.81

[37] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2022. More recent advances in (hyper)graph partitioning. Retrieved from https://arXiv:2205.13202.

[38] Ümit V. Çatalyürek and Cevdet Aykanat. 2001. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'01).* https://doi.org/10.1109/SC.2001.10035

[39] J. Chen and I. Safro. 2011. Algebraic distance on graphs. *SIAM J. Sci. Comput.* 33, 6 (2011), 3468–3490. https://doi.org/10.1137/090775087

[40] C. Chevalier and F. Pellegrini. 2008. PT-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.* 34, 6 (2008), 318–331.

[41] C. Curino, E. Jones, Y. Zhang, and S. Madden. 2010. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.* 3, 1-2 (2010), 48–57. https://doi.org/10.14778/1920841.1920853

[42] Timothy A. Davis, William W. Hager, Scott P. Kolodziej, and S. Nuri Yeralan. 2020. Algorithm 1003: Mongoose, a graph coarsening and partitioning library. *ACM Trans. Math. Softw.* 46, 1 (2020), 1–18. https://doi.org/10.1145/3337792

[43] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. 2011. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (LCNS)*, Vol. 6630. Springer, 376–387.

[44] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. 2015. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *J. Parallel Distrib. Comput.* 77 (2015), 69–83. https://doi.org/10.1016/j.jpdc.2014.12.002

[45] Mehmet Deveci, Sivasankaran Rajamanickam, Karen D. Devine, and Ümit V. Çatalyürek. 2016. Multi-jagged: A scalable parallel spatial partitioning algorithm. *IEEE Trans. Parallel Distrib. Syst.* 27, 3 (2016), 803–817. https://doi.org/10.1109/TPDS.2015.2412545

[46] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek. 2006. Parallel hypergraph partitioning for scientific computing. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06).* IEEE, 124–124.

[47] Hristo N. Djidjev, Georg Hahn, Susan M. Mniszewski, Christian F. A. Negre, and Anders M. N. Niklasson. 2019. Using graph partitioning for scalable distributed quantum molecular dynamics. *Algorithms* 12, 9 (2019), 187. https://doi.org/10.3390/a12090187

[48] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Program.* 91, 2 (2002), 201–213. https://doi.org/10.1007/s101070100263

[49] W. E. Donath. 1988. Logic partitioning. *Phys. Design Autom. VLSI Syst.* (1988), 65–86.

[50] W. E. Donath and A. J. Hoffman. 1972. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Tech. Disclosure Bull.* 15, 3 (1972), 938–944.

[51] W. E. Donath and A. J. Hoffman. 1973. Lower bounds for the partitioning of graphs. *IBM J. Res. Dev.* 17, 5 (1973), 420–425.

[52] S. Dutt and H. Theny. 1997. Partitioning around roadblocks: Tackling constraints with intermediate relaxations. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'97).* IEEE Computer Society/ACM, 350–355. https://doi.org/10.1109/ICCAD.1997.643546

[53] B. O. Fagginger Auer and Rob H. Bisseling. 2012. Graph coarsening and clustering on the GPU. In *Graph Partitioning and Graph Clustering.*

[54] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proc. VLDB Endow.* 13, 10 (2020), 1261–1274. https://doi.org/10.14778/3389133.3389142

[55] C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Conference on Design Automation.* 175–181.

[56] M. Fiedler. 1975. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czech. Math. J.* 25, 4 (1975), 619–633.

[57] J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. 2012. Optimized hybrid parallel lattice Boltzmann fluid flow simulations on complex geometries. In *Proceedings of the Euro-Par Parallel Processing (LNCS)*, Vol. 7484. Springer, 818–829.

[58] Jonas Fietz, Mathias J. Krause, Christian Schulz, Peter Sanders, and Vincent Heuveline. 2012. Optimized hybrid parallel lattice boltzmann fluid flow simulations on complex geometries. In *Proceedings of the 18th International Conference on Parallel Processing (Euro-Par'12) (Lecture Notes in Computer Science)*, Vol. 7484. Springer, 818–829. https://doi.org/10.1007/978-3-642-32820-6_81

[59] Ronald Aylmer Fisher. 1992. Statistical methods for research workers. In *Breakthroughs in Statistics*. Springer, 66–70.

[60] M. Friedman. 1940. A comparison of alternative tests of significance for the problem of m rankings. *Ann. Math. Stat.* 11, 1 (1940), 86–92.

[61] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Vol. 174. W.H. Freeman, San Francisco.

[62] M. R. Garey, D. S. Johnson, and L. Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing (STOC'74)*. ACM, 47–63.

[63] Michael S. Gilbert, Seher Acer, Erik G. Boman, Kamesh Madduri, and Sivasankaran Rajamanickam. 2021. Performance-portable graph coarsening for efficient multilevel graph analysis. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*. 213–222. https://doi.org/10.1109/IPDPS49936.2021.00030

[64] Roland Glantz, Henning Meyerhenke, and Christian Schulz. 2016. Tree-based coarsening and partitioning of complex networks. *ACM J. Exp. Algor.* 21, 1 (2016), 1.6:1–1.6:20. https://doi.org/10.1145/2851496

[65] Christian Godenschwager, Florian Schornbaum, Martin Bauer, Harald Köstler, and Ulrich Rüde. 2013. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 35:1–35:12. https://doi.org/10.1145/2503210.2503273

[66] D. E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

[67] Bahareh Goodarzi, Martin Burtscher, and Dhrubajyoti Goswami. 2016. Parallel graph partitioning on a CPU-GPU architecture. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS'16)*. 58–66. https://doi.org/10.1109/IPDPSW.2016.16

[68] Bahareh Goodarzi, Farzad Khorasani, Vivek Sarkar, and Dhrubajyoti Goswami. 2019. High performance multilevel graph partitioning on GPU. In *Proceedings of the 17th International Conference on High Performance Computing and Simulation (HPCS'19)*. 769–778. https://doi.org/10.1109/HPCS48598.2019.9188120

[69] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. 2020. Advanced flow-based multilevel hypergraph partitioning. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. 11:1–11:15. https://doi.org/10.4230/LIPIcs.SEA.2020.11

[70] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. 2019. Evaluation of a flow-based hypergraph bipartitioning algorithm. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*. 52:1–52:17. https://doi.org/10.4230/LIPIcs.ESA.2019.52

[71] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Scalable shared-memory hypergraph partitioning. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX'21)*. 16–30. https://doi.org/10.1137/1.9781611976472.2

[72] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2022. Shared-memory n-level hypergraph partitioning. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX'22)*. SIAM. https://arxiv.org/abs/2104.08107. to appear.

[73] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2022. Shared-memory n-level hypergraph partitioning. In *Proceedings of the 24th Workshop on Algorithm Engineering and Experiments (ALENEX'22)*. SIAM. https://doi.org/10.1137/1.9781611977042.11

[74] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep multilevel graph partitioning. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA'21) (LIPIcs)*, Vol. 204. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 48:1–48:17. https://doi.org/10.4230/LIPIcs.ESA.2021.48

[75] Lars Gottesbüren and Michael Hamann. 2021. Deterministic parallel hypergraph partitioning. Retrieved from https://arxiv.org/abs/2112.12704.

[76] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. 2022. Parallel flow-based hypergraph partitioning. In *Proceedings of the 20th International Symposium on Experimental Algorithms*, Vol. 233. LIPICS. Retrieved from https://arxiv.org/abs/2201.01556.

[77] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (2021), 410. https://doi.org/10.22331/q-2021-03-15-410

[78] Michelangelo Grigni and Fredrik Manne. 1996. On the complexity of the generalized block distribution. In *Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96) (Lecture Notes in Computer Science)*, Afonso Ferreira, José D. P. Rolim, Yousef Saad, and Tao Yang (Eds.), Vol. 1117. Springer, 319–326. https://doi.org/10.1007/BFb0030123

[79] Alexander Gutfraind, Ilya Safro, and Lauren Ancel Meyers. 2015. Multiscale network generation. In *Proceedings of the 18th International Conference on Information Fusion (FUSION'15)*. IEEE, 158–165. https://ieeexplore.ieee.org/document/7266557/.

[80] Michael Hamann and Ben Strasser. 2018. Graph bisection with pareto optimization. *ACM J. Exper. Algor.* 23 (2018). https://doi.org/10.1145/3173045

[81] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. 2018. Distributed graph clustering using modularity and map equation. In *Proceedings of the 24th International Conference on Parallel and Distributed Computing (EuroPar'18) (Lecture Notes in Computer Science)*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.), Vol. 11014. Springer, 688–702. https://doi.org/10.1007/978-3-319-96983-1_49

[82] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in Pregel-like graph processing systems. *Proc. VLDB Endow.* 8, 9 (2015), 950–961. https://doi.org/10.14778/2777598.2777604

[83] B. Hendrickson and R. Leland. 1995. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.* 16, 2 (1995), 452–469.

[84] Alexandra Henzinger, Alexander Noe, and Christian Schulz. 2020. ILP-based local search for graph partitioning. *ACM J. Exp. Algor.* 25 (2020), 1–26. https://doi.org/10.1145/3398634

[85] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2017. Acyclic partitioning of large directed acyclic graphs. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'17)*. IEEE Computer Society/ACM, 371–380. https://doi.org/10.1109/CCGRID.2017.101

[86] J. Herrmann, M. Yusuf Özkaya, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. 2019. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM J. Sci. Comput.* 41, 4 (2019), A2117–A2145. https://doi.org/10.1137/18M1176865

[87] Tobias Heuer. 2015. *Engineering Initial Partitioning Algorithms for direct k-way Hypergraph Partitioning*. Bachelor Thesis. Karlsruhe Institute of Technology.

[88] Tobias Heuer, Nikolai Maas, and Sebastian Schlag. 2021. Multilevel hypergraph partitioning with vertex weights revisited. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21)*, Vol. 190. 8:1–8:20. https://doi.org/10.4230/LIPIcs.SEA.2021.8

[89] Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2019. Network flow-based refinement for multilevel hypergraph partitioning. *ACM J. Exper. Algor.* 24, 1 (2019), 2.3:1–2.3:36. https://doi.org/10.1145/3329872

[90] T. Heuer and S. Schlag. 2017. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. 21:1–21:19. https://doi.org/10.4230/LIPIcs.SEA.2017.21

[91] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. IEEE, 439–450. https://doi.org/10.1109/IPDPS.2019.00054

[92] M. Holtgrewe, P. Sanders, and C. Schulz. 2010. Engineering a scalable high-quality graph partitioner. *Proceedings of the 24th IEEE International Parallal and Distributed Processing Symposium.* 1–12.

[93] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, et al. 2020. Classical simulation of quantum supremacy circuits. Retrieved from https://arXiv:2005.06787.

[94] Jiewen Huang and Daniel J. Abadi. 2016. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.* 9, 7 (2016). https://doi.org/10.14778/2904483.2904486

[95] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. 2017. Social hash partitioner: A scalable distributed hypergraph partitioner. *Proc. VLDB Endow.* 10, 11 (2017), 1418–1429. https://doi.org/10.14778/3137628.3137650

[96] George Karypis and Vipin Kumar. 1995. Multilevel graph partitioning schemes. In *Proceedings of the International Conference on Parallel Processing, Vol. III: Algorithms and Applications*. CRC Press, 113–122.

[97] George Karypis and Vipin Kumar. 1996. Parallel multilevel K-way partitioning scheme for irregular graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'96)*. IEEE Computer Society. https://doi.org/10.1145/369028.369103

[98] G. Karypis and V. Kumar. 1998. A fast and high-quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[99] George Karypis and Vipin Kumar. 1998. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 28. https://doi.org/10.1109/SC.1998.10018

[100] George Karypis and Vipin Kumar. 1998. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.* 48, 1 (1998), 71–95. https://doi.org/10.1006/jpdc.1997.1403

[101] George Karypis and Vipin Kumar. 2000. Multilevel k-way hypergraph partitioning. *VLSI Design* 2000, 3 (2000), 285–300. https://doi.org/10.1155/2000/19436

[102] Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. 2013. Analysis of partitioning models and metrics in parallel sparse matrix-vector multiplication. In *Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics (PPAM'13), Revised Selected Papers, Part II (LNCS)*, Vol. 8385. Springer, 174–184. https://doi.org/10.1007/978-3-642-55195-6_16

[103] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-efficient graph processing on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT'15)*. IEEE Computer Society, 39–50. https://doi.org/10.1109/PACT.2015.15

[104] Tim Kiefer, Dirk Habich, and Wolfgang Lehner. 2016. Penalized graph partitioning for static and dynamic load balancing. In *Proceedings of the European Conference on Parallel Processing*. Springer, 146–158. https://doi.org/10.1007/978-3-319-43659-3_11

[105] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. 2011. Genetic approaches for graph partitioning: A survey. In *Proceedings of the 13th Genetic and Evolutionary Computation (GECCO'11)*. ACM, 473–480. https://doi.org/10.1145/2001576.2001642

[106] S. Kirmani and P. Raghavan. 2013. Scalable parallel graph partitioning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, 51.

[107] Andrew V. Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.* 23, 2 (2001), 517–541. https://doi.org/10.1137/S1064827500366124

[108] Mathias J. Krause, Adrian Kummerländer, Samuel J. Avis, Halim Kusumaatmaja, Davide Dapelo, Fabian Klemens, Maximilian Gaedtke, Nicolas Hafen, Albert Mink, Robin Trunk, Jan E. Marquardt, Marie-Luise Maier, Marc Haussmann, and Stephan Simonis. 2021. OpenLB—Open source lattice Boltzmann code. *Comput. Math. Appl.* 81 (2021), 258–288. https://doi.org/10.1016/j.camwa.2020.04.033

[109] K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. 2014. SWORD: Workload-aware data placement and replica selection for cloud data management systems. *VLDB J.* 23, 6 (2014), 845–870. https://doi.org/10.1007/s00778-014-0362-1

[110] Ratnesh Kumar, Guillaume Charpiat, and Monique Thonnat. 2014. Multiple object tracking by efficient graph partitioning. In *Proceedings of the 12th Asian Conference on Computer Vision, Revised Selected Papers, Part IV (LNCS)*, Vol. 9006. Springer, 445–460. https://doi.org/10.1007/978-3-319-16817-3_29

[111] Sebastian Lamm, Peter Sanders, and Christian Schulz. 2015. Graph partitioning for independent sets. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15) (LNCS)*, Vol. 9125. Springer, 68–81. https://doi.org/10.1007/978-3-319-20086-6_6

[112] Jesper Larsson Träff. 2006. Direct graph k-partitioning with a Kernighan–Lin like heuristic. *Oper. Res. Lett.* 34, 6 (Nov. 2006), 621–629. https://doi.org/10.1016/j.orl.2005.10.003

[113] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *IEEE Trans. Parallel Distrib. Syst.* IEEE, 225–236. https://doi.org/10.1109/IPDPS.2013.50

[114] Dominique LaSalle and George Karypis. 2016. A parallel hill-climbing refinement algorithm for graph partitioning. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP'16)*. IEEE, 236–241. https://doi.org/10.1109/ICPP.2016.34

[115] Dominique LaSalle, Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Pradeep Dubey, and George Karypis. 2015. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications—Architectures and Algorithms (IA3'15)*. 14:1–14:4. https://doi.org/10.1145/2833179.2833188

[116] Yifan Li, Xiaohui Yu, and Nick Koudas. 2021. LES3: Learning-based exact set similarity search. Retrieved from https://arXiv:2107.10417.

[117] Xiaoyuan Liu, Hayato Ushijima-Mwesigwa, Indradeep Ghosh, and Ilya Safro. 2022. Partitioning dense graphs with hardware accelerators. In *Proceedings of the International Conference on Computational Science (ICCS'22)*, Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer International Publishing, Cham, 476–483.

[118] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Trans. Info. Theory* 28, 2 (1982), 129–137.

[119] Foad Lotfifar and Matthew Johnson. 2015. A multi-level hypergraph partitioning algorithm using rough set clustering. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing (EuroPar'15) (LNCS)*, Vol. 9233. Springer, 159–170. https://doi.org/10.1007/978-3-662-48096-0_13

[120] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. BiPart: A parallel and deterministic hypergraph partitioner. In *Proceedings of the26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21)*. 161–174. https://doi.org/10.1145/3437801.3441611

[121] Zoltán Ádám Mann and Pál András Papp. 2017. Guiding SAT solving by formula partitioning. *Int. J. Artific. Intell. Tools* 26, 4 (2017), 1750011:1–1750011:37. https://doi.org/10.1142/S0218213017500117

[122] Fredrik Manne and Tor Sørevik. 1996. Partitioning an array onto a mesh of processors. In *Proceedings of the3rd International Workshop on Applied Parallel Computing, Industrial Computation, and Optimization (PARA'96) (LNCS)*, Vol. 1184. Springer, 467–477. https://doi.org/10.1007/3-540-62095-8_50

[123] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable graph partitioning in the cloud. In *Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE'17)*. IEEE Computer Society, 1083–1094. https://doi.org/10.1109/ICDE.2017.153

[124] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Survey* 48, 2, Article 25 (2015), 39 pages. https://doi.org/10.1145/2818185

[125] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger. 2009. Graph partitioning and disturbed diffusion. *Parallel Comput.* 35, 10-11 (2009), 544–569. https://doi.org/10.1016/j.parco.2009.09.006

[126] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2014. Partitioning complex networks via size-constrained clustering. In *Proceedings of the 13th International Symposium on Experimental Algorithms (LNCS)*, Vol. 8504. Springer, 351–363. https://doi.org/10.1007/978-3-319-07959-2_30

[127] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering. *J. Heurist.* 22, 5 (2016), 759–782. https://doi.org/10.1007/s10732-016-9315-8

[128] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2625–2638. https://doi.org/10.1109/TPDS.2017.2671868

[129] T. Minyard and Y. Kallinderis. 1998. Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations. *Int. J. Numer. Methods Fluids* 26, 1 (1998), 57–78.

[130] Burkhard Monien, Robert Preis, and Stefan Schamberger. 2007. Approximation algorithms for multilevel graph partitioning. In *Handbook of Approximation Algorithms and Metaheuristics.* Taylor & Francis, Chapter 60, 60–1–60–15. https://doi.org/10.1201/9781420010749.ch60

[131] O. Moreira, M. Popp, and C. Schulz. 2018. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'18).* 332–339. https://doi.org/10.1145/3205455.3205464

[132] Orlando Moreira, Merten Popp, and Christian Schulz. 2020. Evolutionary multi-level acyclic graph partitioning. *J. Heurist.* 26, 5 (2020), 771–799. https://doi.org/10.1007/s10732-020-09448-8

[133] Maxim Naumov and Timothy Moon. 2016. Parallel spectral graph partitioning. NVIDIA Technical Report.

[134] Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. 2015. Hermes: Dynamic partitioning for distributed social network graph databases. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT'15).* 25–36. https://doi.org/10.5441/002/edbt.2015.04

[135] V. Osipov and P. Sanders. 2010. n-level graph partitioning. In *Proceedings of the 18th European Conference on Algorithms: Part I (LNCS)*, Vol. 6346. Springer, 278–289. https://doi.org/10.1007/978-3-642-15775-2_24

[136] Feng Pan and Pan Zhang. 2021. Simulating the sycamore quantum supremacy circuits. Retrieved from https://arXiv:2103.03074.

[137] D. A. Papa and I. L. Markov. 2007. Hypergraph partitioning and clustering. In *Handbook of Approximation Algorithms and Metaheuristics.* Chapman and Hall/CRC. https://doi.org/10.1201/9781420010749.ch61

[138] John R. Pilkington and Scott B. Baden. 1996. Dynamic partitioning of non-uniform structured workloads with space-filling curves. *IEEE Trans. Parallel Distrib. Syst.* 7, 3 (1996), 288–300. https://doi.org/10.1109/71.491582

[139] Merten Popp, Sebastian Schlag, Christian Schulz, and Daniel Seemaier. 2021. Multilevel acyclic hypergraph partitioning. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX'21).* SIAM, 1–15. https://doi.org/10.1137/1.9781611976472.1

[140] A. Pothen, H. D. Simon, and K. P. Liou. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11, 3 (1990), 430–452.

[141] Richard John Preen and Jim Smith. 2019. Evolutionary n-level hypergraph partitioning with adaptive coarsening. *IEEE Trans. Evol. Comput.* 23, 6 (2019), 962–971. https://doi.org/10.1109/TEVC.2019.2896951

[142] U. N. Raghavan, R. Albert, and S. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76, 3 (2007), 11. https://doi.org/10.1103/PhysRevE.76.036106

[143] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Márk Jelasity, and Seif Haridi. 2013. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems.* IEEE Computer Society, 51–60. https://doi.org/10.1109/SASO.2013.13

[144] Diego Recalde, Ramiro Torres, and Polo Vaca. 2020. An exact approach for the multi-constraint graph partitioning problem. *EURO J. Comput. Optim.* 8, 3 (2020), 289–308. https://doi.org/10.1007/s13675-020-00126-9

[145] D. Ron, I. Safro, and A. Brandt. 2011. Relaxation-based coarsening and multiscale graph organization. *Multiscale Model. Simul.* 9, 1 (2011), 407–423. https://doi.org/10.1137/100791142

[146] Emmanuel Romero Ruiz and Carlos Segura. 2018. Memetic algorithm with hungarian matching based crossover and diversity preservation. *Computación y Sistemas* 22, 2 (2018). Retrieved from http://www.cys.cic.ipn.mx/ojs/index.php/CyS/article/view/2951.

[147] P. Sanders and C. Schulz. 2010. *Engineering Multilevel Graph Partitioning Algorithms.* Technical Report. Karlsruhe Institute of Technology. Retrieved from https://arXiv:1012.0006v3.

[148] P. Sanders and C. Schulz. 2011. Engineering multilevel graph partitioning algorithms. In *Proceedings of the 19th European Symposium on Algorithms (LNCS)*, Vol. 6942. Springer, 469–480. https://doi.org/10.1007/978-3-642-23719-5_40

[149] P. Sanders and C. Schulz. 2012. Distributed evolutionary graph partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*. 16–29. https://doi.org/10.1137/1.9781611972924.2

[150] P. Sanders and C. Schulz. 2012. *Think Locally, Act Globally: Perfectly Balanced Graph Partitioning*. Technical Report. Karlsruhe Institute of Technology. Retrieved from https://arXiv:1210.0477.

[151] P. Sanders and C. Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning, In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'12)*. Retrieved from https://arXiv:1210.0477. https://doi.org/10.1007/978-3-642-38527-8_16

[152] Peter Sanders, Christian Schulz, Darren Strash, and Robert Williger. 2017. Distributed evolutionary k-way node separators. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'17)*. ACM, 345–352. https://doi.org/10.1145/3071178.3071204

[153] John E. Savage and Markus G. Wloka. 1991. Parallelism in graph-partitioning. *J. Parallel Distrib. Comput.* 13, 3 (1991), 257–272. https://doi.org/10.1016/0743-7315(91)90074-J

[154] Sebastian Schlag. 2020. *High-quality Hypergraph Partitioning*. Ph.D. Dissertation. Karlsruhe Institute of Technology, Germany. Retrieved from https://nbn-resolving.org/urn:nbn:de:101:1-2020030403581620165765.

[155] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*. SIAM, 53–67. https://doi.org/10.1137/1.9781611974317.5

[156] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. 2019. Scalable edge partitioning. In *Proceedings of the 21st Workshop on Algorithm Engineering and Experiments (ALENEX'19)*. SIAM, 211–225. https://doi.org/10.1137/1.9781611975499.17

[157] Kirk Schloegel, George Karypis, and Vipin Kumar. 1999. A new algorithm for multi-objective graph partitioning. In *Proceedings of the 5th International Euro-Par Conference (EuroPar'99) (Lecture Notes in Computer Science)*, Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz (Eds.), Vol. 1685. Springer, 322–331. https://doi.org/10.1007/3-540-48311-X_42

[158] Kirk Schloegel, George Karypis, and Vipin Kumar. 2001. Graph partitioning for dynamic, adaptive and multi-phase scientific simulations. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'01)*. IEEE Computer Society, 271–273. https://doi.org/10.1109/CLUSTR.2001.959987

[159] K. Schloegel, G. Karypis, and V. Kumar. 2003. Graph partitioning for high-performance scientific simulations. In *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers, 491–541.

[160] C. Schulz. 2013. *High-quality Graph Partitioning*. Ph.D. Dissertation. Karlsruhe Institute of Technology. https://doi.org/10.5445/IR/1000035713

[161] Bin Shao, Haixun Wang, and Yatao Li. 2012. *The Trinity Graph Engine*. Technical Report. Microsoft Research Asia. 4 pages.

[162] Ruslan Shaydulin, Jie Chen, and Ilya Safro. 2019. Relaxation-based coarsening for multilevel hypergraph partitioning. *Multiscale Model. Simul.* 17, 1 (2019), 482–506. https://doi.org/10.1137/17M1152735

[163] Ruslan Shaydulin and Ilya Safro. 2018. Aggregative coarsening for multilevel hypergraph partitioning. In *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA'18) (LIPIcs)*, Gianlorenzo D'Angelo (Ed.), Vol. 103. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2:1–2:15. https://doi.org/10.4230/LIPIcs.SEA.2018.2

[164] H. D. Simon. 1991. Partitioning of unstructured problems for parallel processing. *Comput. Syst. Eng.* 2, 2 (1991), 135–148.

[165] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. 2016. Complex network partitioning using label propagation. *SIAM J. Sci. Comput.* 38, 5 (2016). https://doi.org/10.1137/15M1026183

[166] George M. Slota, Sivasankaran Rajamanickam, Karen D. Devine, and Kamesh Madduri. 2017. Partitioning trillion-edge graphs in minutes. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 646–655. https://doi.org/10.1109/IPDPS.2017.95

[167] George M. Slota, Cameron Root, Karen D. Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. 2020. Scalable, multi-constraint, complex-objective graph partitioning. *IEEE Trans. Parallel Distrib. Syst.* 31, 12 (2020), 2789–2801. https://doi.org/10.1109/TPDS.2020.3002150

[168] Guy L. Steele. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of the Annual Symposium on Principles of Programming Languages (POPL'90)*. ACM Press, 218–231. https://doi.org/10.1145/96709.96731

[169] Valerie E. Taylor and Bahram Nour-Omid. 1994. A study of the factorization fill-in for a parallel implementation of the finite element method. *Int. J. Numer. Methods Eng.* 37, 22 (1994), 3809–3823.

[170] Aleksandar Trifunovic and William J. Knottenbelt. 2004. Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium.* IEEE Computer Society. https://doi.org/10.1109/IPDPS.2004.1303286

[171] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy

Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming model extensions for the exascale era. *IEEE Trans. Parallel Distrib. Syst.* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[172] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. 333–342. https://doi.org/10.1145/2556195.2556213

[173] Bora Uçar and Cevdet Aykanat. 2004. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Sci. Comput.* 25, 6 (2004), 1837–1859. https://doi.org/10.1137/S1064827502410463

[174] Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining (WSDM'13)*. ACM, 507–516. https://doi.org/10.1145/2433396.2433461

[175] Hayato Ushijima-Mwesigwa, Christian F. A. Negre, and Susan M. Mniszewski. 2017. Graph partitioning using quantum annealing on the D-wave system. In *Proceedings of the 2nd International Workshop on Post Moores Era Supercomputing (PMES'17)*. Association for Computing Machinery, New York, NY, 22–29. https://doi.org/10.1145/3149526.3149531

[176] Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2014. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 144–153. https://doi.org/10.1109/ICDCS.2014.23

[177] Moritz von Looz, Charilaos Tzovas, and Henning Meyerhenke. 2018. Balanced k-means for parallel geometric partitioning. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP'18)*. ACM, 52:1–52:10. https://doi.org/10.1145/3225058.3225148

[178] Moritz von Looz, Mario Wolter, Christoph R. Jacob, and Henning Meyerhenke. 2016. Better partitions of protein graphs for subsystem quantum chemistry. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16) (LNCS)*, Vol. 9685. Springer, 353–368. https://doi.org/10.1007/978-3-319-38851-9_24

[179] Chris Walshaw. 2003. *An Exploration of Multilevel Combinatorial Optimisation*. Springer U.S., 71–123. https://doi.org/10.1007/978-1-4757-3748-6_2

[180] C. Walshaw, M. Cross, and M. G. Everett. 1997. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.* 47, 2 (1997), 102–108.

[181] Chris Walshaw, Mark Cross, and Martin G. Everett. 1997. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.* 47, 2 (1997), 102–108. https://doi.org/10.1006/jpdc.1997.1407

[182] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. 2014. How to partition a billion-node graph. In *Proceedings of the IEEE 30th International Conference on Data Engineering*. IEEE Computer Society, 568–579. https://doi.org/10.1109/ICDE.2014.6816682

[183] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

[184] Roy D. Williams. 1991. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurr. Pract. Exp.* 3, 5 (1991), 457–481. https://doi.org/10.1002/cpe.4330030502

[185] Ning Xu, Lei Chen, and Bin Cui. 2014. LogGP: A log-based dynamic graph partitioning method. *Proc. VLDB Endow.* 7, 14 (2014), 1917–1928. https://doi.org/10.14778/2733085.2733097

[186] Honghua Yang and D. F. Wong. 1994. Efficient network flow based min-cut balanced partitioning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'94)*. IEEE Computer Society/ACM, 50–55. https://doi.org/10.1109/ICCAD.1994.629743

[187] Wenyin Yang, Guojun Wang, Kim-Kwang Raymond Choo, and Shuhong Chen. 2018. HEPart: A balanced hypergraph partitioning algorithm for big data applications. *Future Gener. Comput. Syst.* 83 (2018), 250–268. https://doi.org/10.1016/j.future.2018.01.009

[188] Long Yao, Peilin Hong, Wen Zhang, Jianfei Li, and Dan Ni. 2015. Controller placement and flow based dynamic management problem towards SDN. In *Proceedings of the IEEE International Conference on Communication (ICC'15)*. IEEE, 363–368. https://doi.org/10.1109/ICCW.2015.7247206

[189] Abdurrahman Yasar, Muhammed Fatih Balin, Xiaojing An, Kaan Sancak, and Ümit V. Çatalyürek. 2020. On symmetric rectilinear matrix partitioning. Retrieved from https://arxiv.org/abs/2009.07735.

[190] Abdurrahman Yasar and Ümit V. Çatalyürek. 2019. Heuristics for symmetric rectilinear matrix partitioning. Retrieved from http://arxiv.org/abs/1909.12209.

[191] G. Alastair Young, Thomas A. Severini, George Albert Young, R. L. Smith, et al. 2005. *Essentials of Statistical Inference*. Vol. 16. Cambridge University Press.

[192] Xiaosong Yu, Huihui Ma, Zhengyu Qu, Jianbin Fang, and Weifeng Liu. 2020. NUMA-aware optimization of sparse matrix-vector multiplication on armv8-based many-core architectures. In *Proceedings of the IFIP International*

Conference on Network and Parallel Computing (LNCS), Vol. 12639. Springer, 231–242. https://doi.org/10.1007/978-3-030-79478-1_20

[193] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. 2021. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale graphs. Retrieved from https://arxiv.org/abs/2112.15345.

[194] Xiaojin Zhu and Zoubin Ghahramani. 2002. Learning from labeled and unlabeled data with label propagation. Technical Report.

Received 30 June 2022; revised 7 November 2022; accepted 14 November 2022