Scalable Edge Partitioning*

Sebastian Schlag† Christian Schulz‡ Daniel Seemaier§ Darren Strash¶

Abstract

Edge-centric distributed computations have appeared as a recent technique to improve the shortcomings of think-like-a-vertex algorithms on large scale-free networks. In order to increase parallelism on this model, *edge partitioning*—partitioning edges into roughly equally sized blocks—has emerged as an alternative to traditional (node-based) graph partitioning. In this work, we develop a fast parallel split-and-connect graph construction algorithm in the distributed setting and show that combining our parallel construction with advanced parallel node partitioning algorithms yields high-quality edge partitions in a scalable way. Our technique scales to networks with billions of edges, and runs efficiently on thousands of PEs. Our extensive experiments show that our algorithm computes solutions of high quality on large real-world networks and large hyperbolic random graphs—which have a power law degree distribution and are therefore specifically targeted by edge partitioning.

1 Introduction

With the recent stagnation of Moore's law, the primary method for gaining computing power is to increase the number of available cores, processors, or networked machines (all of which are generally referred to as *processing elements* (PEs)) and exploit parallel computation. One increasingly useful method to take advantage of parallelism is found in *graph partitioning* [6, 10, 56], which attempts to partition the vertices of a graph into roughly equal disjoint sets (called *blocks*), while minimizing some objective function—for example minimizing the number of edges crossing between blocks. Graph partitioning is highly effective, for instance, for distributing data to PEs in order to minimize communication volume [24], and to minimize the overall running time of jobs with dependencies between computation steps [57].

This traditional (node-based) graph partitioning has also been essential for making efficient distributed graph algorithms in the Think Like a Vertex (TLAV) model of computation [43]. In this model, node-centric operations are performed in parallel, by mapping nodes to PEs and executing node computations in parallel. Nearly all algorithms in this model require information to be communicated between neighbors — which results in network communication if stored on different PEs — and therefore high-quality graph partitioning directly translates into less communication and faster overall running time. As a result, graph partitioning techniques are included in popular TLAV platforms such as Pregel [40] and GraphLab [39].

However, node-centric computations have serious shortcomings on power law graphs — which have a skewed degree distribution. In such networks, the overall running time is negatively affected by very high-degree nodes, which can result in more communication steps. To combat these effects, Gonzalez et al. [23] introduced edge-centric computations, which duplicates node-centric computations across edges to reduce communication overhead. In this model, *edge partitioning*—partitioning edges into roughly equally sized blocks—must be used to reduce the overall running time. However, like node-based partitioning, edge partitioning is NP-hard [9].

Similar to (node-based) graph partitioning, the quality of the edge partitioning can have a dramatic effect on parallelization [38, 47]. Noting that edge partitioning can be solved directly with hypergraph partitioners, such as hMETIS [32, 33] and PaToH [13], Li et al. [38] showed that these techniques give the highest quality partitionings; however, they are also slow. Therefore, a balance of solution quality and speed must be taken into consideration. This balance is struck well for the split-and-connect (SPAC) method introduced by Li et al. [38]. In the SPAC method, vertices are duplicated and weighted so that a (typically fast) standard node-based graph partitioner can be used to compute an edge partitioning; however, this method was only studied in the sequential setting. While this is a great initial step, the graphs that benefit the most

*The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506. This work was partially supported by DFG grants SA 933/10-2.

†Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany.

‡Faculty of Computer Science, University of Vienna, Vienna, Austria.

§Karlsruhe Institute of Technology, Karlsruhe, Germany.

¶Department of Computer Science, Hamilton College, Clinton, NY, USA.

from edge partitioning are massive—and therefore do not fit on a single machine [28].

However, distributed algorithms for the problem fare far worse [9, 23]. While adding much computational power with many processing elements (PEs), edge partitioners such as PowerGraph [23] and Ja-Be-Ja-VC [50], produce partitionings of significantly worse quality than those produced with hypergraph partitioners or SPAC. Thus, there is no clear winning algorithm that gives high quality while executing quickly in a distributed setting.

1.1 Our Results. In this paper, we give the first *high-quality* distributed memory parallel edge partitioner. Our algorithm scales to networks with billions of edges, and runs efficiently on thousands of PEs. Our technique is based on a fast parallelization of split-and-connect graph construction and a use of advanced node partitioning algorithms. Our experiments show that while hypergraph partitioners outperform SPAC-based graph partitioners in the sequential setting regarding both solution quality and running time, our new algorithms compute significantly *better* solutions than the distributed memory hypergraph partitioner Zoltan [20] in *shorter* time. For large random hyperbolic graphs, which have a power law degree distribution and are therefore specifically targeted by edge partitioning, our algorithms compute solutions that are more than a factor of two better. Moreover, our techniques scale well to 2 560 PEs, allowing for efficient partitioning of graphs with billions of edges within seconds.

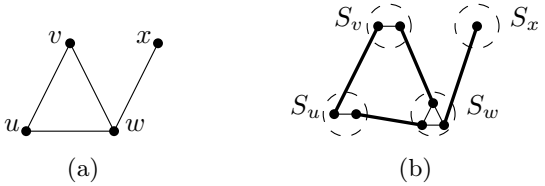
2 Preliminaries

2.1 Basic Concepts. Let $G = (V = \{0, \dots, n-1\}, E, c, \omega)$ be an undirected graph with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $N(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v and $E(v) := \{e : v \in e\}$ denotes the edges incident to v. Given a positive integer k, we are looking for *blocks* of nodes $V_1, \dots, V_k$ that partition V, i.e., $V_1 \cup \dots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. A node $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a *boundary node*. The *balance constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter ε. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. Similar to the node partitioning problem, the *edge partitioning problem* asks for blocks of edges $E_1, \dots, E_k$ that partition E, i.e. $E_1 \cup \dots \cup E_k = E$ and $E_i \cap E_j = \emptyset$ for $i \neq j$. The *balance constraint* demands that $\forall i \in \{1..k\} : \omega(E_i) \leq (1+\varepsilon)\left\lceil \frac{\omega(E)}{k} \right\rceil$. The objective is to minimize the *vertex cut* $\sum_{v \in V} |I(v)| - 1$ where $I(v) := \{i : E(v) \cap E_i \neq \emptyset\}$. Intuitively, the objective expresses the number of required *replicas* of nodes: if a node v has to be copied to each block that has edges incident to v, the number of required replicas of that node is $|I(v)| - 1$.

2.2 Hypergraphs. An *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 is a subset of the vertex set V (i.e., $e \subseteq V$). The vertices of a net are called *pins*. As before, c and ω are extended to work on sets. A vertex v is *incident* to a net e if $v \in e$. The *size* $|e|$ of a net e is the number of its pins. A *k-way partition* of a hypergraph 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 $i \neq j$. A k-way partition Π is called *ε-balanced* if each block $V_i \in \Pi$ satisfies the *balance constraint*: $c(V_i) \leq L_{\max} := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter ε. Given a k-way partition Π, the number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. For each net e, $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$ denotes the *connectivity set* of e. The *connectivity* of a net e is the cardinality of its connectivity set: $\lambda(e) := |\Lambda(e)|$. A net is called *cut net* if $\lambda(e) > 1$. The *k-way hypergraph partitioning problem* is to find an ε-balanced k-way partition Π of a hypergraph H that minimizes an objective function over the cut nets for some ε. The most commonly used cost functions are the *cut-net* metric $\text{cut}(\Pi) := \sum_{e \in E'} \omega(e)$ and the *connectivity* metric $(\lambda - 1)(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\, \omega(e)$, where E' is the set of all cut nets [19, 21]. Optimizing each of these objective functions is NP-hard [36].

2.3 The Split-and-Connect (SPAC) Method. One technique to compute an edge partitioning is to first form a hypergraph with a node for each edge in E, and a hyperedge for each node, consisting of the edges to which it is incident. Thus, hypergraph partitioners optimizing the connectivity metric can be applied to this problem [38]; however, they are more powerful than necessary, as this conversion has mostly small hyperedges.

The problem can also be solved with node-based graph partitioning by creating a new graph G' with the *split-and-connect transformation* (SPAC) of Li et al. [38]. More precisely, given an undirected, unweighted graph $G = (V, E)$, they construct the split-and-connect graph $G' = (V', E', c', \omega')$ as follows: for each node $v \in V$, create a set of *split nodes* $S_v := \{v'_1, \dots, v'_{d(v)}\}$ that are connected to a cycle by *auxiliary edges* with edge-weight one, i.e. edges $\{v'_i, v'_{i+1}\}$ for $i = 1, \dots, d(v) - 1$ and $\{v'_{d(v)}, v'_1\}$. In the connect



(a) (b)

Figure 1: (a) The input graph. (b) The resulting split-and-connect graph. Each node v is replaced by a set S_v of split nodes that form a cycle (if $|S_v| \geq 3$). Auxiliary edges are drawn thin, dominant edges are drawn thick.

phase, split nodes are connected by edges, i.e. for each edge $e = \{u, v\}$ in G, a corresponding *dominant edge* $\{u', v'\}$ in G' is created. This is done such that overall both $u' \in S_u$ and $v' \in S_v$ are connected to one and only one dominant edge. Those dominant edges get assigned edge weight infinity. Figure 1 gives an example.

Note that the original version of the SPAC method connects the split nodes of the same split set to an induced path rather than a cycle. Exchanging them for cycles simplifies implementation and does not change the theoretical approximation bound [38].

To partition the edges of G, a node-based partitioning algorithm is run on G'. Since dominant edges have edge weight infinity, it is infeasible to cut dominant edges for the node-based partitioning algorithm. Thus, both endpoints of a dominant edge are put into the same partition. To obtain an edge partition of the input graph, one transfers the block numbers of those endpoints to the edge in G that induced the dominant edge.

Each distinct node partition occurring in a split node set cuts at least one auxiliary edge, unless the set is fully contained in a single partition. Hence, the number of node replicas is at most the number of edge cuts.

Since the vertex cut is always smaller than or equal to the edge cut, a good node partition of the split-and-connect graph intuitively leads to a good edge partition of the input graph. Note that the node-based partitioning algorithm is a parameter of the algorithm. Overall, their approach is shown to be up to orders of magnitude faster than the hypergraph partitioning approaches using hMetis [30] and PaToH [14] and considered competitive in terms of the number of replicas [38].

3 Related Work

There has been a *huge* amount of research on graph and hypergraph partitioning so that we refer the reader to existing literature [6, 10, 46, 56] for most of the material. Here, we focus on issues closely related to our main contributions. Since both graph partitioning algorithms using the method by Li et al. [38] and hypergraph partitioning algorithms are useful to solve the edge partitioning problem, we start this section with reviewing literature for those problems and then finish with edge partitioning algorithms.

3.1 Node Partitioning.

One of the most prominent methods for computing high-quality node partitions on large real-world graphs is the multilevel scheme. In the *multilevel graph partitioning* (MGP) method, the input graph is recursively *contracted* to achieve smaller graphs which should reflect the same basic structure as the input graph. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level. Well-known software packages based on this approach include KaHIP [54], Jostle [64], METIS [31] and Scotch [15].

Most probably the fastest available parallel code is the parallel version of METIS, ParMETIS [29]. This parallelization has difficulty maintaining the balance of the partitions since at any particular time, it is difficult to say how many nodes are assigned to a particular block. PT-Scotch [15], the parallel version of Scotch, is based on recursive bipartitioning.

Within this work, we use sequential and distributed memory parallel algorithms of the open source multilevel graph partitioning framework KaHIP [54] (Karlsruhe High Quality Partitioning). This framework tackles the node partitioning problem using the edge cut as objective. ParHIP [44] is a distributed memory parallel node partitioning algorithm. The algorithm is based on parallelizing and adapting the *label propagation* technique originally developed for graph clustering [49]. By introducing size constraints, label propagation becomes applicable for both the coarsening and the refinement phase of multilevel graph partitioning. The resulting system is more scalable and achieves higher quality than commonly-used partitioners like ParMETIS and PT-Scotch.

3.2 Hypergraph Partitioning.

Driven by applications in VLSI design and parallel scientific computing, hypergraph partitioning (HGP) has evolved into a broad research area since the 1990s and lead to the development of multiple HGP software packages with certain distinguishing characteristics. Well-known *sequential* systems—all of which use the multilevel paradigm—include PaToH [13] (originating from scientific computing), hMetis [32, 33] (originating from VLSI design), KaHyPar [1, 25, 55] (n-level, general purpose), Mondriaan [62] (targeted at partitioning sparse rectangular matrices), MLPart [3] (targeted at circuit partitioning), Zoltan-AlgD [58] (coarsening inspired by algebraic

multigrid methods), UMPa [61] (directed hypergraph model, multi-objective partitioning), and kPaToH [4] (multiple constraints, fixed vertices). *Distributed* HGP systems include Zoltan [20] and Parkway [60] (multilevel), and SHP [28] (non-multilevel).

3.3 Edge Partitioning. While hypergraph partitioning and the SPAC method are effective for computing an edge partitioning of small graphs, different techniques are used for graphs that do not fit in the memory of a single computer. In the Chaos graph processing system [52], edges are partitioned randomly. Gonzalez et al. [23] study the streaming case, where edges are assigned to partitions in a single pass over the graph. They investigate randomly assigning edges to partitions, as well as a greedy strategy (which they call Oblivious). Using their greedy method, the average number of replicas is around 5 for their collection of power-law graphs. Bourse et al. [9] later improved the replication factor by performing a similar process, but weighting each vertex by its degree. The degree-based hashing (DBH) method of Xie et al. [66] and the high-degree replicated first (HDRF) approach of Petroni et al. [48] are streaming edge partitioning algorithms that exploit power-law degree distributions by using information about the vertex degrees in the partitioning process. Sheep [41] is a distributed algorithm that partitions an elimination tree of the input graph and then uses the tree partition to derive an edge partition. The neighborhood expansion (NE) algorithm of Zhang et al. [67] computes each block of the edge partition separately and bases edge assignment decisions on the neighborhood structure of already assigned vertices. It has been shown to perform better than DBH [66], Oblivious [23], HDRF [48], and Sheep [41]. ADWISE [42] is a streaming edge partitioning algorithm that is positioned between single-edge streaming algorithms and traditional graph partitioning approaches that consider the entire input graph, because it operates on a sliding window of edges at a time. With Ja-Be-Ja-VC, Rahimian et al. [50] present a distributed parallel algorithm for edge-based partitioning of large graphs which is essentially a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. Mykhailenko et al. [45] propose a distributed edge partitioning framework based on simulated annealing, which is not yet able to consistently perform better than Ja-Be-Ja-VC.

4 Engineering a Parallel Edge Partitioner

We now present our algorithm to quickly compute high quality edge partitions in the distributed memory setting. Roughly speaking, we engineer a distributed version of the SPAC algorithm (dSPAC) and then use a distributed memory parallel node-based graph partitioning to partition the model. We start this section by giving a description of the data structures that we use and then explain the parallelization of the SPAC algorithm.

4.1 Graph Data Structure. We start with details of the parallel graph data structure and the implementation of the methods that handle communication. First of all, each PE gets a subgraph, i.e. a contiguous range of nodes $a..b$, of the whole graph as its input, such that the subgraphs combined correspond to the input graph. Each subgraph consists of the nodes with IDs from the interval $I := a..b$ and the edges incident to at least one node in this interval, and includes vertices not in I that are adjacent to vertices in I. These vertices are referred to as *ghost* nodes (also referred to elsewhere in the literature as *halo* nodes). Note that every non-local node can induce at most one ghost node. Note that each PE may have edges it shares with another PE and the number of edges assigned to the PEs may therefore vary significantly. The subgraphs are stored using a standard adjacency array representation—we have one array to store edges and one array for nodes, which stores head pointers into the edge array. However, for the parallel setting, the node array is divided into two parts: the first part stores local nodes and the second part stores ghost nodes. The method used to keep local node IDs and ghost node IDs consistent is explained next.

Instead of using the node IDs provided by the input graph (i.e., the *global* IDs), each PE p maps those IDs to the range $0\,..\,n_p - 1$, where n_p is the number of distinct nodes of the subgraph on that PE. Note that this number includes the number of ghost nodes stored on PE p. The number of local nodes is denoted by ℓ_p. Each global ID $i \in a\,..\,b$ is mapped to a local node ID $i - a$. The IDs of the ghost nodes are mapped to the remaining $n_p - (b - a + 1)$ local IDs in the order in which they appeared during the construction of the graph data structure. Transforming a local node ID to a global ID or vice versa, can be done by adding or subtracting a. We store the global ID of the ghost nodes in an extra array and use a (local) hash table to transform global IDs of ghost nodes to their corresponding local IDs. Additionally, we store for each ghost node the ID of its corresponding PE, using an array `PE` for $\mathcal{O}(1)$ lookups. We call a node an *interface node* if it is adjacent to at least one ghost node. The PE associated with the ghost node is called an *adjacent PE*.

Our data structure stores undirected edges as two directed edges, a forward and a backward edge. Similar to node IDs, directed edges are mapped to the range $0\,..\,m_p - 1$ of local edge IDs, where m_p denotes the

number of local directed edges of the subgraph on PE p. Edges that start from node $v \in V(G)$ have consecutive edge IDs $e_v \,..\, e_{v+d(v)-1}$. A roughly equal distribution of nodes to PEs is suboptimal for the SPAC algorithm since the size of the split-and-connect graph mostly depends on the number of edges. Hence, we improve load imbalance by distributing the graph such that the *number of edges* is roughly the same on each PE. We do this by assigning consecutive nodes to a PE until the number of edges incident to those nodes exceeds the average number of edges per processor m/P.

4.2 Distributed Split-And-Connect Graph Construction. We use G to denote the input graph with n_p nodes and m_p edges on PE p and construct the split-and-connect graph G' with n'_p nodes and m'_p edges on the same PE. A pseudocode description of our distributed SPAC algorithm (dSPAC) is given in Algorithm 1.

First, recall that the split-and-connect graph contains a set S_v of $d(v)$ split nodes for each node $v \in V(G)$. For a node v, we create the nodes of the split-and-connect graph on the PE that owns v. Thus, the number of local split-and-connect graph nodes on PE p is equal to the number of local edges of the input graph on PE p, i.e. $n'_p = m_p$. Since the edges incident to the same node have consecutive IDs, we can use edge IDs from G as local node IDs in the split-and-connect graph G'. To transform those local node IDs to global ones, we need the overall number of edges on PEs that have a smaller PE ID than p. This can be easily computed in parallel by computing a prefix sum over the number of local edges m_p of each PE p. This takes $\mathcal{O}(\log p)$ time and linear work [34].

Afterwards, the SPAC transformation requires us to connect split nodes by auxiliary edges and dominant edges. Auxiliary edges with edge-weight one are inserted between split nodes $v \in S_v$ to connect them to an induced cycle. Since S_v is fully contained on a single PE, we can create the auxiliary edges independently on each PE.

Recall that dominant edges are induced by undirected edges $\{u, v\}$ in G. These edges connect split nodes, i.e. nodes from the sets S_u and S_v, such that each split node is incident to precisely one (undirected) dominant edge. To construct dominant edges, global coordination is required: say that u has neighbors on two different PEs. Both neighbors do not know about each other, since u is only available as ghost node on those PEs, i.e. without information about its adjacency. Yet, both neighbors must choose a unique split node from the set S_u.

Algorithm 1: Our dSPAC algorithm on PE p

Input: Graph G with n_p nodes, m_p edges on PE p
Input: Empty split-and-connect graph G'
$n'_p := m_p$ *// one split node per local edge*
$m_p^{\text{global}} := \texttt{prefixSum}(m_p)$ *// i.e., $\sum_{p'=0}^{p-1} m_p$*
for $i := 0;\ i < n'_p;\ ++i$ **do** *// create split nodes ...*
  $G'.\texttt{insertNode}(m_p^{\text{global}} + i)$ *// ... with global IDs*
// compute $\mathcal{E}'_p$ values for all interface nodes
foreach $u \in V(G)$ **do** *// i.e., $u \in 0\,..\,l_p - 1$*
  $p' := -1$
  foreach $e \in E(u)$ **do** *// i.e., $e \in e_u \,..\, e_{u+d(u)-1}$*
    $v := \texttt{edgeTarget}(e)$
    if $\texttt{PE}[v] \neq p'$ **then**
      $p' := \texttt{PE}[v]$
      $\mathcal{E}_{p'}(a + u) := m_p^{\text{global}} + e$
// exchange $\mathcal{E}'_p$ values with adjacent PEs
foreach *PE* $p' \neq p$ **do**
  send $\mathcal{E}_{p'}$ values to PE p'
  receive $\mathcal{E}_p$ values from PE p'
// insert auxiliary and dominant edges
foreach $u \in V(G)$ **do** *// i.e., $u \in 0\,..\,l_p - 1$*
  foreach $e \in E(u)$ **do** *// i.e., $e \in e_u \,..\, e_{u+d(u)-1}$*
    $v := \texttt{edgeTarget}(e)$
    $v' := \texttt{globalID}(v)$
    $u' := e + m_p^{\text{global}}$ *// global ID of split node u'*
    // insert dominant edge with edge-weight ∞
    $G'.\texttt{insertEdge}(u', \mathcal{E}_p(v'), \infty)$
    $\mathcal{E}_p(v') := \mathcal{E}_p(v') + 1$
    if $d(u) > 1$ **then** *// insert auxiliary edges*
      // compute target nodes for aux. edge
      $u_{\text{prev}} := e - e_u - 1 \mod d(u)$
      $u_{\text{next}} := e - e_u + 1 \mod d(u)$
      // local $\rightarrow$ global ID for target node
      $v' := e_u + m_p^{\text{global}} + u_{\text{next}}$
      $G'.\texttt{insertEdge}(u', v', 1)$
      if $u_{prev} \neq u_{next}$ **then**
        // local $\rightarrow$ global ID for target node
        $v' := e_u + m_p^{\text{global}} + u_{\text{prev}}$
        $G'.\texttt{insertEdge}(u', v', 1)$
Output: distributed split-and-connect graph G'

Our algorithm solves this problem as follows. First, the adjacency lists of each node are ordered by their global node ID. We assume that this is already the case for the input network, otherwise one can simply run a sorting algorithm on the neighborhood of each vertex. Since nodes are assigned consecutively among the processors, this implies that the adjacency list of each interface node in the input graph is ordered by the

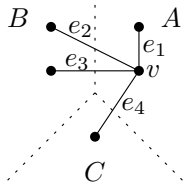


Figure 2: A node $v \in V(G)$ on PE A with neighbors on PEs A, B and C. Neighbors are traversed in the order of the edges. PEs are ordered $A < B < C$.

target processor, i.e. the processor that owns the target of the edge. Figure 2 gives an example.

For an interface node $u \in V(G)$ on PE p, let $\mathcal{E}_{p'}(u)$ be the global ID of the first edge $\{u, v\}$ with $v \in V(G)$ on PE p'. In the split-and-connect graph G' this ID corresponds to the global ID of the first split node in S_u that will be adjacent to a split node of S_v on PE p'. Due to the order of the vertices, these values can be easily computed for all adjacent PEs by scanning the neighborhood of that vertex. We send the corresponding value to PE p'. Using this information, we will be able to construct dominant edges in the desired way. To avoid startup overheads, we first compute all $\mathcal{E}_{p'}(v_I)$ values for all adjacent interface nodes v_I and then send a *single* message from p to p' that contains all values. Hence, the total message size for PE p is $\mathcal{O}(n_I^p P_A^p)$ where n_I^p is the number of interface nodes on PE p and P_A^p is the number of PEs adjacent to PE p.

We now create dominant edges by using the just computed values as follows. First of all, each processor traverses its nodes in the order of their IDs. Let $u \in V(G)$ be a node on PE p with its ordered neighbors being $v_1, \ldots, v_{d(v)}$. For each edge $\{u, v_i\}$, we create a dominant edge from u's i-th split node to $\mathcal{E}_p(v_i)$, i.e. we create the dominant edge $\{u'_i, \mathcal{E}_p(v_i)\}$. Note that the value $\mathcal{E}$ was initially sent from the PE that contains v_i. Afterwards, we increment $\mathcal{E}_p(v_i)$ by one, so that the next neighbor of v_i that is on PE p connects a dominant edge to v_i's next split node.

LEMMA 4.1. *Our parallel SPAC algorithm creates a valid split-and-connect graph.*

Proof. First note that, by the process above, we create precisely one dominant edge for each split node. Hence, it is sufficient to show that the resulting split-and-connect graph is undirected. Consider a pair of adjacent nodes $u, v \in V(G)$ where u is owned by PE p and v is owned by PE p'. We show that both vertices pick the correct split node — hence, forming an undirected (dominant) edge.

Roughly speaking, by the order in which the nodes and their incident edges are traversed it is ensured that the values $\mathcal{E}$ used for the creation of the edges point to the correct split node. More precisely, let $u \in V(G)$ be a node of the input graph and its ordered neighbors be $v_1, \ldots, v_{d(u)}$. We consider $\{u, v_i\}$ and argue that its induced dominant edge is indeed undirected. For this purpose, let u be owned by PE p and v_i be owned by PE p'. On PE p, we create a directed edge (u'_i, v'), where u'_i is u's i-th split node and v' is *some* split node of v defined by the process above. We argue that PE p', creates an edge (v', u'_i) which makes the graph undirected. It is sufficient to argue that both edges include u'_i, since we can use the same argument with u and v_i reversed to imply that the other endpoint is correct too. Node u chooses u'_i for the dominant edge from S_u to S_{v_i}, because it traverses its neighbors in order and v_i is its i-th neighbor. In the other direction, v_i chooses the split node of u based on $\mathcal{E}_{p'}(u)$. We claim that it chooses u'_i. Let v_j be the first neighbor of u on PE p'. Since the neighborhood is ordered as described above, $v_j, \ldots, v_i$ are all on PE p' and moreover, they are traversed in the same order on PE p' (thus construct their dominant edges in the same order). Thus, v_i connects the dominant edge to $\mathcal{E}_{p'}(u) + (i - j)$, since that is the total increment of $\mathcal{E}_{p'}(u)$ at the time when v_i constructs its dominant edges. But by the definition of $\mathcal{E}_{p'}(u)$, we have that $\mathcal{E}_{p'}(u)$ is the global split node ID of u'_j. Thus, v_i connects to u's $j + (i - j) = i$-th split node.

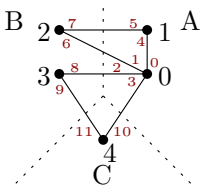
Assuming that the adjacency list of the nodes are already sorted by global ID, our algorithm performs a linear amount of work. Thus split-and-connect graph construction takes $\mathcal{O}(m/p + \log p)$ time, if edges are distributed evenly.

After computing the split-and-connect graph, we use the distributed parallel node-partitioning algorithms ParHIP [44] and ParMETIS [29] to partition it. To obtain an edge partition of the input graph, we transfer the block numbers of those endpoints to the edge in G that induced the dominant edge.

5 Experimental Evaluation

In this section we evaluate the performance of the proposed algorithm. We start by presenting our methodology and setup, the system used for the evaluation and the benchmark set that we used. We then look at solution quality, running time, and scalability of (d)SPAC-based GP as well as HGP, and compare our algorithm to those systems.

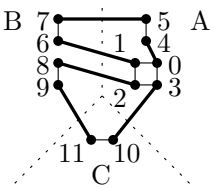
5.1 Methodology and Setup. We implemented the distributed split-and-connect graph construction algo-



(a)

↱	A	B	C
A		$\mathcal{E}_B(0) = 1$ $\mathcal{E}_B(1) = 5$	$\mathcal{E}_C(0) = 3$
B	$\mathcal{E}_A(2) = 5$ $\mathcal{E}_A(3) = 7$		$\mathcal{E}_C(3) = 8$
C	$\mathcal{E}_A(4) = 9$	$\mathcal{E}_B(4) = 10$	

(b)



(c)

Figure 3: (a) The input graph, distributed across three PEs (indicated by the dashed delimiters). IDs of directed edges are drawn next to their sources. (b) The $\mathcal{E}_p(\cdot)$ messages each PE sends to other PEs: the messages in the i-th row and j-th column are sent from PE i to PE j. (c) The constructed split-and-connect graph. Auxiliary edges are drawn thin, dominant edges are drawn thick.

rithm described in Section 4 in the ParHIP graph partitioning framework [44]. In the following, we use SPAC when referring to sequential split-and-connect graph construction and use dSPAC to denote our algorithm in the distributed setting. The code is written in C++, compiled with g++ 7.3.0, and uses OpenMPI 1.10 as well as KaHIP v2.0[1].

In order to establish the state-of-the-art regarding edge partitioning, we perform a large number of experiments using several partitioning tools including sequential and distributed graph and hypergraph partitioners.

For *graph partitioning*, our experimental comparisons use the sequential systems KaHIP [53] and METIS [31] as well as their respective distributed versions ParHIP [44] and ParMETIS [29]. For *hypergraph partitioning*, we use the sequential tools PaToH [13], Zoltan-AlgD [58], the k-way (hMetis-K) and the recursive bisection variant (hMetis-R) of hMetis 2.0 (p1) [32, 33], and KaHyPar-MF [26]. To evaluate distributed hypergraph partitioning approaches, we include Zoltan [20]. We also tried to use Par$k$way [59], but were not able work with the current version provided online[2], because the code has deadlocks and hangs on many instances. Note that although Zoltan-AlgD is implemented in the Zoltan framework, it does not work in parallel mode. We chose these HGP systems because of the following reasons: KaHyPar performs better than hMetis and PaToH [26], but has not been compared to Zoltan-AlgD yet. PaToH in turn produces better quality than Zoltan's native parallel hypergraph partitioner (PHG) in serial mode [8, 20] and has been shown to shown to compute better solutions than Mondriaan [7, 51] and MLPart [12]. Additionally, MLPart is restricted to bisections [11, 16]. The performance of SHP is comparable to the performance of Zoltan and Mondriaan [28]. UMPa does not improve on PaToH when optimizing single objective functions that do not benefit from the directed hypergraph model [19]. Furthermore kPaToH [4] does not perform better than PaToH for standard hypergraph partitioning [1].

For *edge partitioning*, we include the distributed Ja-Be-Ja-VC algorithm and the sequential NE algorithm [67]. Since there is no implementation of the Ja-Be-Ja-VC algorithm [50] publicly available, we include our own implementation. Judging from the results presented in [50], both implementations provide comparable solution quality. However, since Ja-Be-Ja-VC performed significantly worse than all other partitioning approaches in our experiments, we only consider it in a sequential setting. Furthermore we do not report running times for Ja-Be-Ja-VC, because all other systems are highly engineered, while our Ja-Be-Ja-VC implementation is a prototype. For partitioning, we use $\varepsilon = 0.03$ as the imbalance factor for all tools except h-METIS-R, which treats the imbalance parameter differently. We therefore use an adjusted imbalance value as described in [55].

For each algorithm, we perform five repetitions with different seeds and use the arithmetic mean to average solution quality and running time of the different runs. When averaging over different instances, we use the *geometric mean* in order to give every instance a comparable influence on the final result.

We furthermore use *performance plots* [55] to compare the best solutions of competing algorithms on a per-instance basis. For each algorithm, these plots relate the smallest minimum vertex cut of *all* algorithms to the corresponding vertex cut produced by each algorithm on a per-instance basis. These ratios are then sorted in decreasing order for each algorithm. The plots show $1 - (\text{best}/\text{algorithm})$ on the y-axis to highlight the instances were each partitioner performs badly and use a cube root scale on the y-axis to reduce right skewness [17]. A point close to one indicates that the

[1] The source code and detailed per-instance results of our experiments are available from: `https://algo2.iti.kit.edu/edge_partitioning/`.

[2] `https://github.com/parkway-partitioner/parkway`

partition produced by the corresponding algorithm was considerably worse than the partition produced by the best algorithm. A value of zero therefore indicates that the corresponding algorithm produced the best solution. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are below those of the other algorithm. Points above one correspond to infeasible solutions that violate the balance constraint.

5.2 System and Instances. We use the ForHLR II cluster (Forschungshochleistungsrechner) for our experimental evaluation. This cluster has 1 152 compute nodes, each of which is equipped with 64 GB main memory and two Intel Xeon E5-2660 Deca-Core v3 processors (Haswell) clocked at 2.6 GHz. A single Deca-Core processor has 25 MB L3-Cache, and every core has 256 KB L2-Cache and 64 KB L1-Cache. All cluster nodes are connected by an InfiniBand 4X EDR interconnect.

We evaluate the algorithms on the graphs listed in Table 1, which include standard partitioning benchmark instances (Walshaw + SPMV), larger real-world instances, and randomly generated instances. Random geometric rggX graphs have 2^X nodes and were generated using code from [27]. Random hyperbolic rhgX graphs are generated using [22] with power law exponent 2.2 and average degree 8. rhgX graphs were chosen since their degree distributions follow a power law (and are thus targeted by edge partitioning techniques). SPMV graphs are bipartite locality graphs for sparse matrix vector multiplication (SPMV), which were also used to evaluate the sequential SPAC algorithm in [38]. Given a $n \times n$ matrix M (in our case the adjacency matrix of the corresponding graph), an SPMV graph corresponding to an SPMV computation $Mx = y$ consists of $2n$ vertices representing the x_i and y_i vector entries and contains an edge (x_i, y_j) if x_i contributes to the computation of y_j, i.e. if $M_{ij} \neq 0$. To evaluate the hypergraph approaches, we transform the graphs into hypergraphs as described in Section 2.3. A hypergraph contains one hypernode for each undirected edge in the graph and a hyperedge for each graph node that contains the hypernodes corresponding to its incident edges.

5.3 Solution Quality of SPAC+X and HGP. We start by exploring the solution quality provided by the different *sequential* algorithmic approaches to the edge partitioning problem, i.e., we consider the results of Ja-Be-Ja-VC and the NE algorithm, as well as the vertex cut that is obtained by applying the partition of the SPAC or hypergraph model to the input graph. We restrict the benchmark set to the Walshaw graphs,

Graph	n	m	Type	Ref.
Walshaw Graph Archive				
add20	≈ 2.3K	≈ 7.4K	M	[63]
data	≈ 2.8K	≈ 15K	M	[63]
3elt	≈ 4.7K	≈ 13.7K	M	[63]
uk	≈ 4.8K	≈ 6.8K	M	[63]
add32	≈ 4.9K	≈ 9.4K	M	[63]
bcsstk33	≈ 8.7K	≈ 291K	M	[63]
whitaker3	≈ 9.8K	≈ 289K	M	[63]
crack	≈ 10K	≈ 30K	M	[63]
wing_nodal	≈ 10K	≈ 75K	M	[63]
fe_4elt2	≈ 11K	≈ 32K	M	[63]
vibrobox	≈ 12K	≈ 165K	M	[63]
bcsstk29	≈ 13K	≈ 302K	M	[63]
4elt	≈ 15K	≈ 45K	M	[63]
fe_sphere	≈ 16K	≈ 49K	M	[63]
cti	≈ 16K	≈ 48K	M	[63]
memplus	≈ 17K	≈ 54K	M	[63]
cs4	≈ 22K	≈ 43K	M	[63]
bcsstk30	≈ 28K	≈ 1M	M	[63]
bcsstk31	≈ 35K	≈ 572K	M	[63]
fe_pwt	≈ 36K	≈ 144K	M	[63]
bcsstk32	≈ 44K	≈ 985K	M	[63]
fe_body	≈ 45K	≈ 163K	M	[63]
t60k	≈ 60K	≈ 89K	M	[63]
wing	≈ 62K	≈ 121K	M	[63]
brack2	≈ 62K	≈ 366K	M	[63]
finan512	≈ 74K	≈ 261K	M	[63]
fe_tooth	≈ 78K	≈ 452K	M	[63]
fe_rotor	≈ 99K	≈ 662K	M	[63]
598a	≈ 110K	≈ 741K	M	[63]
fe_ocean	≈ 143K	≈ 409K	M	[63]
144	≈ 144K	≈ 1M	M	[63]
wave	≈ 156K	≈ 1M	M	[63]
m14b	≈ 214K	≈ 1.6M	M	[63]
auto	≈ 448K	≈ 3.3M	M	[63]
rhgX	2^{10} – 2^{18}	≈ $3.6K$ – 976K	S	[22]
SPMV Graphs				
cant_spmv	≈ 125K	≈ 2M	SP	[65]
scircuit_spmv	≈ 350K	≈ 100K	SP	[18]
mc2depi_spmv	≈ 1M	≈ 2.1M	SP	[65]
in-2004_spmv	≈ 2.5M	≈ 17M	SP	[35]
circuit5M_spmv	≈ 11M	≈ 60M	SP	[18]
Large Graphs				
amazon	≈ 407K	≈ 2.3M	S	[37]
eu-2005	≈ 862K	≈ 16.1M	S	[5]
youtube	≈ 1.1M	≈ 2.9M	S	[37]
in-2004	≈ 1.4M	≈ 27M	S	[5]
packing	≈ 2.1M	≈ 17.4M	M	[5]
channel	≈ 4.8M	≈ 42.6M	M	[5]
road_central	≈ 14M	≈ 34M	R	[5]
hugebubble-10	≈ 18.3M	≈ 27.5M	M	[5]
uk-2002	≈ 18.5M	≈ 262M	S	[35]
nlpkkt240	≈ 27.9M	≈ 373M	M	[18]
europe_osm	≈ 51M	≈ 108M	R	[5]
rhgX	2^{20} – 2^{26}	≈ 4M – 280M	S	[22]
Huge Graphs				
rggX	2^{25} – 2^{28}	≈ 550M – 5G	M	[27]

Table 1: Our benchmark set. Type 'S' stands for social or web graphs, 'M' is used for mesh type networks, 'R' is used for road networks, SP(MV) is used for graphs for sparse matrix-vector multiplication.

SPMV graphs with up to 1M nodes[3] and `rhg10` – `rhg18`, since the running times for hypergraph partitioners were too high for larger instances. We run all partitioners on *one* PE, i.e., one core of a single node. Each instance is partitioned into k blocks for $k \in \{2, 4, 8, 16, 32, 64, 128\}$.

The results are presented in Figure 4 and Figure 5. Both Ja-Be-Ja-VC and NE can not compete with hypergraph partitioning and SPAC-based graph partitioning approaches in terms of solution quality. The inferior performance of these flat (i.e., non-multilevel) edge partitioning algorithms echoes the intuition that by providing a more global view of the partitioning problem on the coarser levels, multilevel schemes enable local search algorithms, which are known to easily get stuck in local optima [32], to explore local solution spaces very effectively. Furthermore note that NE computes imbalanced solutions in 42 cases. Most of these correspond to partitions of rhgX graphs with large values of k. Since the algorithm does not explicitly check the current size of a block E_i (i.e., number of edges already assigned to E_i) while allocating incident edges of core vertices to block E_i, it is likely to produce imbalanced solutions for graphs having a power-law degree distribution.

In the experiments of Li et al. [38], SPAC+METIS was significantly faster than the hypergraph partitioners hMetis and PaToH, while achieving comparable solution quality. However, this comparison was restricted to five graphs. In our experiments using a larger benchmark set, all hypergraph partitioners except Zoltan perform better than SPAC+METIS for most instances, with PaToH being roughly a factor of 1.6 slower on average. This result could be explained by the choice of k—the number of blocks used for partitioning. While we use standard values for (node-based) graph partitioning benchmarks [63], Li et al. [38] choose k such that each block contains *approximately* 10 240 edges. Thus some instances are partitioned into up to 1 692 and 5 952 blocks, which might be too large for current partitioning tools.

Looking at the solution quality of different SPAC+X approaches, we see that KaHIP performs better than METIS when using its *strong* configuration at the cost of an increased running time. However, partitioning the hypergraph model with KaHyPar-MF, which dominates all other hypergraph partitioners in terms of solution quality, overall results in the lowest vertex cuts, while also being slightly faster than SPAC+KaHIP on average. In a sequential edge partitioning setting with a reasonable number k of blocks, we therefore conclude that both the existing flat edge partitioning and the SPAC-based graph partitioning approaches are outperformed by hypergraph partitioning regarding solution quality (using KaHyPar-MF).

[3] `scircuit_spmv`, `cant_spmv` and `mc2depi_spmv`

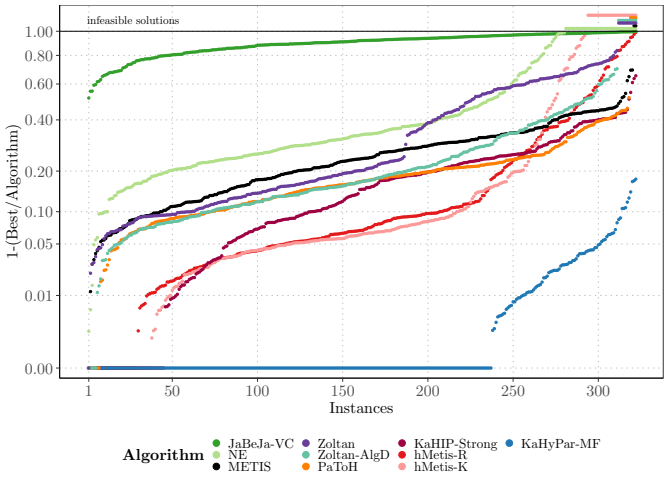


Figure 4: Comparing solution quality of SPAC+X, several HGP tools, JaBeJa-VC, and NE on small graphs.

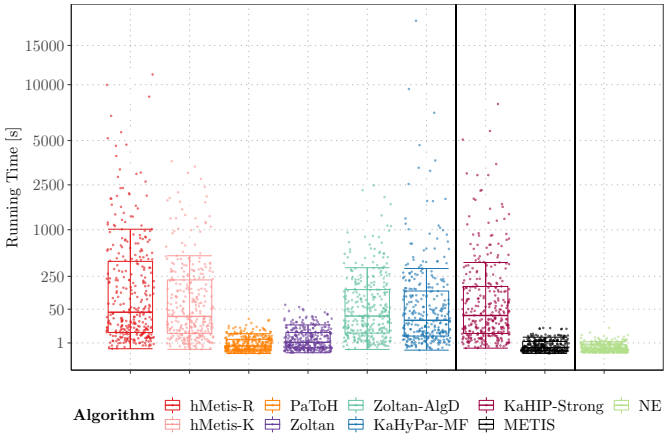


Figure 5: Running time comparison for SPAC+X, several HGP systems, and NE on small graphs. Note the cube root scale on the y-axis.

5.4 Solution Quality of dSPAC+X and dHGP.

We now investigate state-of-the-art methods for computing edge partitions in the distributed memory setting. Here, we use the large graphs from Table 1 including `rhg20` – `rhg26`, as well as the two large SPMV graphs `in-2004_spmv` and `circuit5M_smpv`. Since Ja-Be-Ja-VC [50] already produced low quality solutions on small graphs, we restrict the following comparison to distributed memory hypergraph partitioning (dHGP) with Zoltan and distributed graph partitioning using our distributed split-and-connect graph construction (dSPAC) in combination with both ParMETIS and ParHIP. All instances are again partitioned into

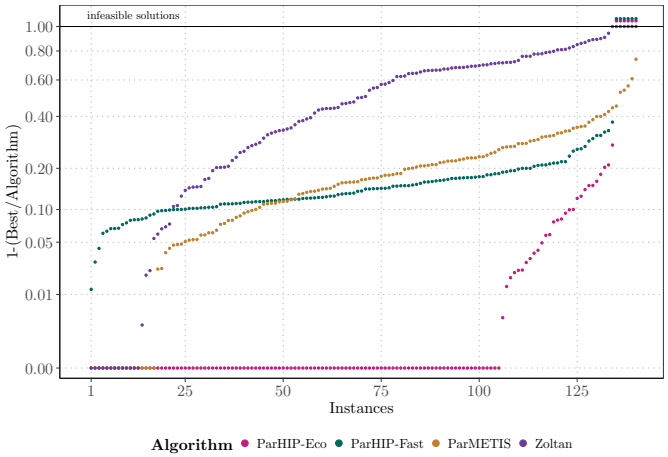


Figure 6: Solution quality for distributed SPAC+X and HGP approaches on large graphs. Instances[4] for which Zoltan ran out of memory are set to 1.

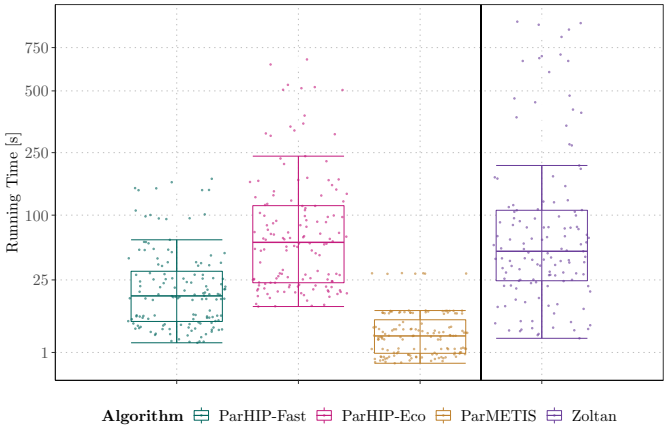


Figure 7: Running times of distributed approaches on large social graphs, meshes, road networks, and SPMV graphs. Instances[5] for which Zoltan ran out of memory are excluded. Note the cube root scale on the y-axis.

$k \in \{2, 4, 8, 16, 32, 64, 128\}$ blocks. This time, we run all algorithms on 32 cluster nodes (i.e., with 640 PEs in total). As can be seen in Figure 6 and Figure 7, the results differ from the sequential setting. The dSPAC-based graph partitioning approach outperforms the hypergraph partitioning approach using Zoltan in *both* solution quality and running time. While dSPAC+ParMETIS is the fastest configuration, dSPAC+ParHIP-Eco provides the best solution quality. ParMETIS furthermore performs better than ParHIP-Fast by computing solutions of comparable quality in less time. In a distributed setting, we thus conclude that combining the SPAC approach with a high quality distributed graph partitioner is currently the best approach for computing edge partitions of large graphs, in particular if sequential partitioning is not an option.

5.5 Scalability and Solution Quality of dSPAC+X. Finally, we look at the scaling behavior of distributed SPAC graph construction and partitioning using ParMETIS and ParHIP-Fast. To simplify the evaluation, we restrict the experiments in this section to partitioning the eight largest graphs (including `rgg25` - `rgg28`) into $k = 2$ blocks on an increasing number of PEs. We start with a single PE on a single node and then go up to all 20 PEs of a single node. From there on we double the number of nodes in each step, until we arrive at 128 nodes with a total of 2 560 PEs. The results are shown in Figure 9. Results for the remaining large graphs can be found in Figure 10 in Appendix A. The running times of dSPAC+X are dominated by the running times of the distributed graph partitioners (see Figure 8). While dSPAC+ParMETIS is faster than dSPAC+ParHIP-

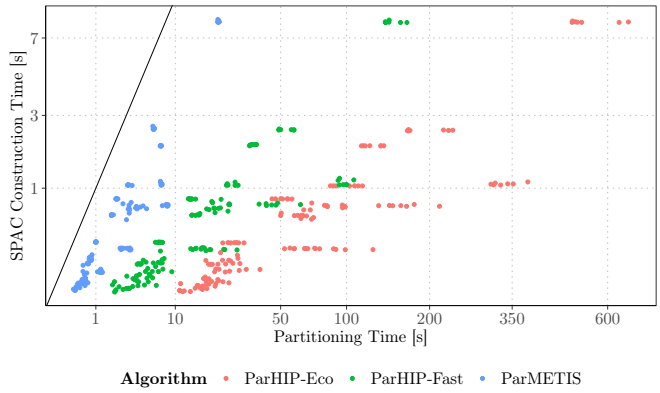


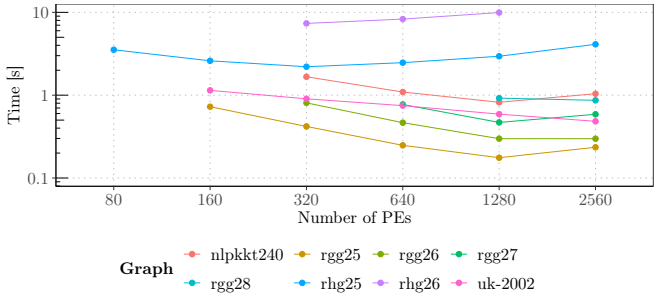
Figure 8: Comparing the running times of distributed split-and-connect graph construction and partitioning.

Fast, the latter scales slightly better than the former. Regarding solution quality, For large numbers of PEs, ParHIP-Fast computes better solutions than ParMETIS (see detailed per-instance results). By combining our distributed split-and-connect graph construction algorithm with high quality distributed graph partitioning algorithm, we are able to compute edge partitions of huge graphs that were previously not solvable on a single PE, or even a small number of PEs.

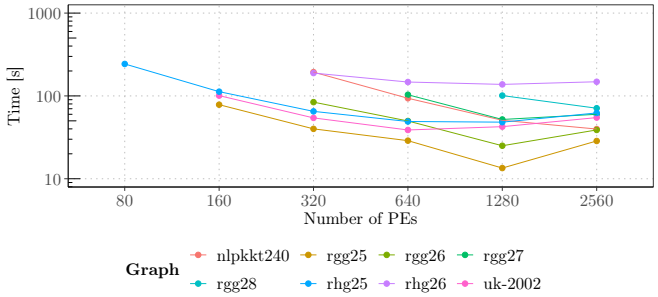
6 Conclusion and Future Work

We presented an efficient distributed memory parallel edge partitioning algorithm that computes solutions of very high quality. By efficiently parallelizing the split-and-connect graph construction, our dSPAC+X algo-

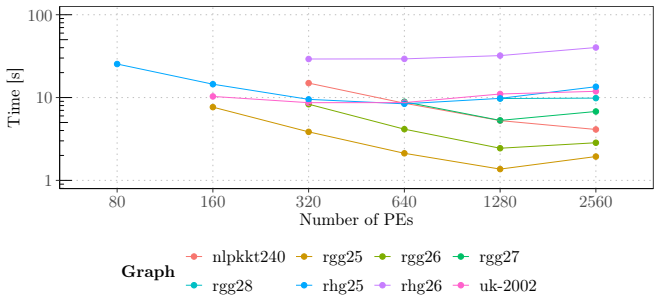
[5] `nlpkkt240` with $k \in \{2, 4, 8, 16, 32, 64, 128\}$.



(a) Running time for distributed split-and-connect graph construction.



(b) Running time for dSPAC+ParHIP-Fast.



(c) Running time for dSPAC+ParMETIS.

Figure 9: Comparing the running times of distributed split-and-connect graph construction (a), dSPAC+ParHIP-Fast (b), and dSPAC+ParMETIS (c) for the eight largest graphs of our benchmark set on an increasing number of PEs.

rithm scales to graphs with billions of edges and runs efficiently on up to 2 560 PEs. Our extensive experiments furthermore show that in a *sequential* setting hypergraph partitioners still outperform node-based graph partitioning methods based on the SPAC approach regarding solution quality while maintaining similar running time. Hence, further research into hypergraph partitioning must be done in order to narrow the gap between dSPAC+X and dHGP. In the future, we would like to run a working implementation of Par$k$way in order to get a complete overview regarding the state-of-the-art in distributed HGP. Furthermore it would be interesting to combine ParHIP with the the shared memory parallel MT-KaHIP [2] partitioner in order to get a partitioner that uses shared memory parallelism within a cluster node, while cluster nodes themselves still work in a distributed memory fashion.

Acknowledgments. This work was performed on the computational resource ForHLR II funded by the Ministry of Science, Research and the Arts Baden-Württemberg.

References

[1] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *19th Workshop on Algorithm Engineering and Experiments, (ALENEX)*, pages 28–42, 2017.

[2] Y. Akhremtsev, P. Sanders, and C. Schulz. High-quality shared-memory graph partitioning. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018: Parallel Processing*, volume 11014 of *LNCS*, pages 659–671. Springer, 2018. doi:10.1007/978-3-319-96983-1_47.

[3] C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel Circuit Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, 1998.

[4] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68 (5):609–625, 2008. ISSN 0743-7315.

[5] D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

[6] C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011.

[7] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Catalyürek. Two-dimensional Approaches to Sparse Matrix Partitioning. In *Combinatorial Scientific Computing*, pages 321 – 349. CRC Press, New York, 2012.

[8] E. Boman, K. Devine, V. Leung, S. Rajamanickam, L. A. Riesen, and Ü. V. Catalyürek. Zoltan User's Guide. `http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_patoh.html`, 2012.

[9] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced Graph Edge Partition. In *Proc. 20th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465. ACM, 2014.

[10] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*, pages 117–158. Springer, 2016.

[11] A. Caldwell, A. Kahng, and I. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000.*, pages 661–666, June 2000.

[12] Ü. V. Catalyürek. ISPD98 Benchmark. `http://bmi.osu.edu/umit/PaToH/ispd98.html`.

[13] Ü. V. Catalyürek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7): 673–693, Jul 1999. ISSN 1045-9219.

[14] Ü. V. Catalyürek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. `http://bmi.osu.edu/umit/PaToH/manual.pdf`, 1999.

[15] C. Chevalier and F. Pellegrini. PT-Scotch. *Parallel Computing*, 34(6-8):318–331, 2008.

[16] J. Cong, M. Romesis, and M. Xie. Optimality, scalability and stability study of partitioning and placement algorithms. In *International Symposium on Physical Design*, ISPD, pages 88–94. ACM, 2003.

[17] N. J. Cox. Stata tip 96: Cube roots. *Stata Journal*, 11(1):149–154(6), 2011. URL `http://www.stata-journal.com/article.html?article=st0223`.

[18] T. Davis. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`, 2008.

[19] M. Deveci, K. Kaya, B. Uçar, and Ü. V. Çatalyürek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77: 69–83, 2015.

[20] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Conference on Parallel and Distributed Processing (IPDPS)*, pages 124–124. IEEE, 2006.

[21] W. Donath. Logic partitioning. *Physical Design Automation of VLSI Systems*, pages 65–86, 1988.

[22] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free massively distributed graph generation. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2018.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30. USENIX, 2012.

[24] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[25] T. Heuer and S. Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms, (SEA)*, pages 21:1–21:19, 2017.

[26] T. Heuer, P. Sanders, and S. Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. In *17th International Symposium on Experimental Algorithms (SEA)*, volume 103, pages 1:1–1:19, 2018.

[27] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *Proceedings of the 24th IEEE International Parallal and Distributed Processing Symposium*, pages 1–12, 2010.

[28] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, A. Presta, and Y. Akhremtsev. Social Hash Partitioner: A scalable distributed hypergraph partitioner. *Proceedings VLDB Endow.*, 10(11):1418–1429, August 2017.

[29] G. Karypis and V. Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing'96*, 1996.

[30] G. Karypis and V. Kumar. hMETIS: A hypergraph partitioning package, version 1.5.3. *user manual*, 23, 1998.

[31] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[32] G. Karypis and V. Kumar. Multilevel K-way Hypergraph Partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 343–348. ACM, 1999.

[33] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration VLSI Systems*, 7 (1):69–79, 1999.

[34] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[35] Laboratory of Web Algorithms, University of Milano. Datasets, `http://law.dsi.unimi.it/datasets.php`.

[36] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[37] J. Leskovec. Stanford Network Analysis Package (SNAP). `http://snap.stanford.edu/index.html`.

[38] L. Li, R. Geda, A. B. Hayes, Y. Chen, P. Chaudhari, E. Z. Zhang, and M. Szegedy. A simple yet effective balanced edge partition model for parallel computing. *SIGMETRICS Perform. Eval. Rev.*, 45 (1):6–6, June 2017.

[39] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In P. Grünwald and P. Spirtes, editors, *Proc. 26th Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349. AUAI Press, 2010.

[40] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. 2010 ACM SIGMOD International Conf. on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[41] D. W. Margo and M. I. Seltzer. A scalable distributed graph partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[42] C. Mayer, R. Mayer, M. A. Tariq, H. Heiko Geppert, L. Laich, L. Rieger, and K. Rothermel. ADWISE: adaptive window-based streaming edge partitioning for high-speed graph processing. In *38th International Conference on Distributed Computing Systems (ICDCS) 2018*, pages 685–695, 2018.

[43] R. R. McCune, T. Weninger, and G. Madey. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[44] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1055–1064, May 2015.

[45] H. Mykhailenko, G. Neglia, and F. Huet. Simulated annealing for edge partitioning. In *Conference on Computer Communications Workshops (INFOCOM)*, pages 54–59, 2017.

[46] D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In T. F. Gonzalez, editor, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

[47] M. M. A. Patwary, R. H. Bisseling, and F. Manne. Parallel greedy graph matching using an edge partitioning approach. In *Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 45–54, 2010.

[48] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: stream-based partitioning for power-law graphs. In *24th International Conference on Information and Knowledge Management, (CIKM)*, pages 243–252, 2015.

[49] U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3), 2007.

[50] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. Distributed vertex-cut partitioning. In K. Magoutis and P. Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 186–200, Berlin, Heidelberg, 2014.

[51] S. Riyavong. Experiments on Sparse Matrix Partitioning. Technical Report CERFACS Working Note WN/PA/03/32, CERFACS, 2003.

[52] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *25th Symposium on Operating Systems Principles (SOSP)*, pages 410–424, 2015.

[53] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

[54] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'12)*, LNCS. Springer, 2013.

[55] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way Hypergraph Partitioning via n-Level Recursive Bisection. In *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67, 2016.

[56] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

[57] C. Schulz and D. Strash. Graph partitioning: Formulations and applications to big data. In S. Sakr and A. Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–7. Springer, 2018.

[58] R. Shaydulin and I. Safro. Aggregative Coarsening for Multilevel Hypergraph Partitioning. In *17th International Symposium on Experimental Algorithms (SEA)*, volume 103, pages 2:1–2:15, 2018.

[59] A. Trifunovic and W. J. Knottenbelt. Parkway 2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *Computer and Information Sciences - ISCIS 2004*, volume 3280, pages 789–800. Springer, 2004.

[60] A. Trifunovic and W. J. Knottenbelt. Parallel Multilevel Algorithms for Hypergraph Partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[61] Ü. V. Çatalyürek and M. Deveci and K. Kaya and B. Uçar. UMPa: A multi-objective, multilevel partitioner for communication minimization. In D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors, *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*, pages 53–66. AMS, 2012.

[62] B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.

[63] C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

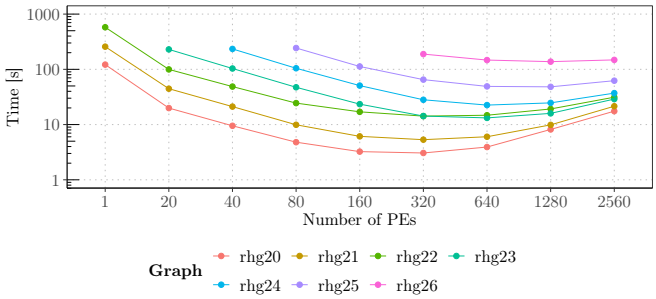
[64] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[65] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

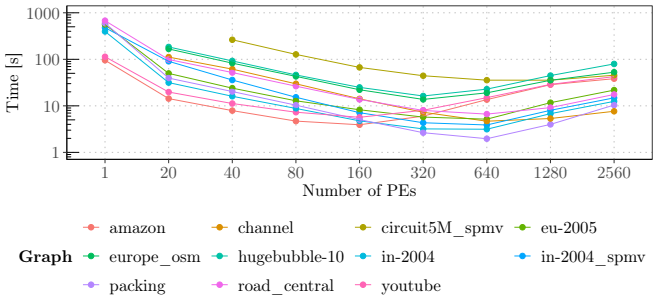
[66] C. Xie, L. Yan, W. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *Neural Information Processing Systems (NIPS)*, pages 1673–1681, 2014.

[67] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and L. Z. Graph edge partitioning via neighborhood heuristic. In *23rd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 605–614, 2017.

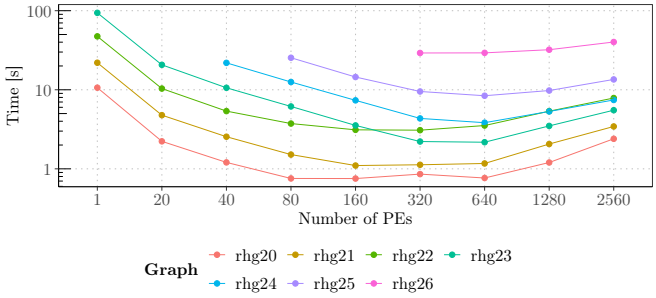
A Additional Experimental Results for dSPAC+X



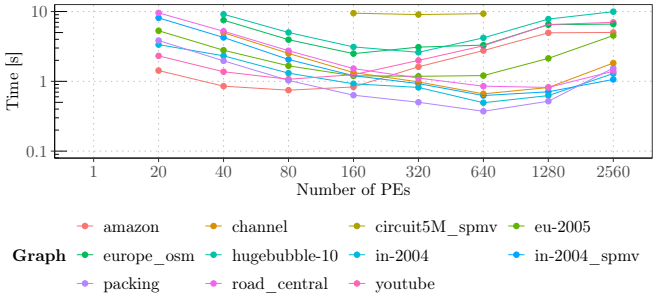
(a) Running time for dSPAC+ParHIP-Fast.



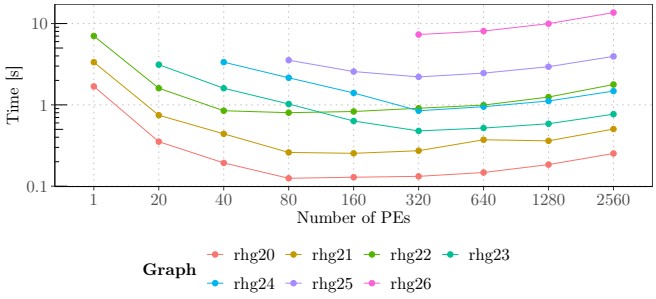
(b) Running time for dSPAC+ParHIP-Fast.



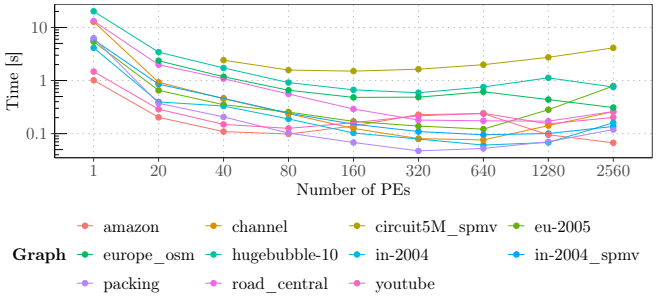
(c) Running time for dSPAC+ParMETIS.



(d) Running time for dSPAC+ParMETIS.



(e) Running time for distributed split-and-connect graph construction.



(f) Running time for distributed split-and-connect graph construction.

Figure 10: Running times for dSPAC+ParHIP-Fast (a, b), dSPAC+ParMETIS (c, d) and the time it takes to compute the distributed split-and-connect graph (e, f) on increasing numbers of PE.