Deterministic Parallel Hypergraph Partitioning

Lars Gottesbüren
lars.gottesbueren@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Michael Hamann
michael.hamann@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

Balanced hypergraph partitioning is a classical NP-hard optimization problem with applications in various domains such as VLSI design, simulating quantum circuits, optimizing data placement in distributed databases or minimizing communication volume in high-performance computing. Engineering parallel heuristics for this problem is a topic of recent research. Most of them are non-deterministic though. In this work, we design and implement a highly scalable deterministic algorithm in the state-of-the-art parallel partitioning framework Mt-KaHyPar. On our extensive set of benchmark instances, it achieves similar partition quality and performance as a comparable but non-deterministic configuration of Mt-KaHyPar and outperforms the only other existing parallel deterministic algorithm BiPart regarding partition quality, running time and parallel speedups.

1 Introduction

Balanced k-way hypergraph partitioning (HGP) is a classical optimization problem. Its goal is to divide the vertices of a hypergraph into k blocks of bounded size while minimizing an objective function on hyperedges that connect more than one block. Hypergraphs are a generalization of graphs where hyperedges can contain more than two vertices. A commonly used objective function is the *connectivity metric* where we aim to minimize the sum of the number of blocks connected by each hyperedge. This problem is NP-hard [31] and even hard to approximate [4] within constant factors, which is why heuristic algorithms are used in practice. Balanced hypergraph partitioning has numerous applications in domains such as VLSI design [2, 3], logic synthesis for integrated circuits [35], simulating quantum circuits [24], scientific computing [10], SAT solving [33], as well as storage sharding in distributed databases and data centers [13, 28].

There has been a huge amount of research on graph and hypergraph partitioning, but especially in recent years, the interest in parallel algorithms has surged [16, 20, 22, 23, 28, 32] due to ever growing problem sizes. With the exception

of BiPart [32], these algorithms are non-deterministic. Researchers have advocated the benefits of deterministic parallel algorithms for several decades [7, 9, 27, 30], including ease of debugging, reasoning about performance, and reproduciblity. While some strive for deterministic programming models, we want to leverage randomized scheduling for better performance and thus pursue deterministic algorithms. The above reasons are desirable properties, yet the BiPart authors [32] argue that for VLSI circuit design, deterministic partitioning results are even *necessary*, since some manual post-processing is involved that should not be repeated.

Previous results showed that the quality of partitions computed by BiPart does not compare favorably with current state-of-the-art parallel algorithms such as Mt-KaHyPar [23] and our experiments show that BiPart exhibits poor scalability. The goal of this work is to design, implement and evaluate a scalable and deterministically parallel hypergraph partitioning algorithm with state-of-the-art solution quality.

1.1 Multilevel Partitioning

More formally, the hypergraph partitioning problem is defined as follows. Given a hypergraph $H = (V, E)$, *imbalance parameter* $\varepsilon \in (0, 1)$, and number of blocks $k \in \mathbb{N}$, find a k-way partition $V_1 \cup \cdots \cup V_k = V$ of the vertices V that is ε-balanced $|V_i| \leq (1 + \varepsilon)\lceil |V|/k \rceil$. The objective function to minimize is the *connectivity metric* $\sum_{e \in E}(\lambda(e) - 1)$, where $\lambda(e) := |\{V_i \mid V_i \cap e \neq \emptyset\}|$ is the number of blocks connected by hyperedge e. The most successful approach for partitioning is the multilevel framework. Starting with the input hypergraph, a sequence of successively smaller (coarser) but structurally similar hypergraphs is constructed by repeatedly contracting groups of vertices (clusters or matchings). This is called the *coarsening phase*. Once a sufficiently small hypergraph is reached, expensive algorithms compute an *initial partition*. In the following *refinement phase*, this partition is then projected back up through the hierarchy by assigning vertices to the same block as their representative in the next coarser hypergraph. This yields a partition of the larger hypergraph with the same imbalance and objective function as on the coarse hypergraph. On each level of the hierarchy, local search algorithms such as Fiduccia-Mattheyses (FM) [18] or label propagation [36] move vertices to improve the current partition.

Mt-KaHyPar [22] adds a preprocessing phase based on community detection to the coarsening phase. Community

---

Algorithm 1: Local Moving Round

---

for $v \in V$ *in random order* **do in parallel**
 | compute and perform best move for v
 | update data structures

Algorithm 2: Synchronous Local Moving Round

---

randomly split vertices into sub-rounds
for $r = 0$ **to** *number of sub-rounds* **do**
 | **for** $v \in V$ *in sub-round r* **do in parallel**
 | | compute and save best move for v
 | approve saved moves and update data structures

---

detection, also called clustering, aims to group vertices together that are internally densely but externally sparsely connected. Contractions during the coarsening phase are then restricted to vertices in the same community to avoid destroying small cuts, as first proposed in [26]. To avoid confusion, we use the terms *community detection* and *communities* in the preprocessing phase, while we use *clustering* and *clusters* in the coarsening phase.

There are two fundamental approaches to the multilevel framework. In direct k-way partitioning, the initial partition is directly k-way and refinement algorithms operate on a k-way partition. Recursive bipartitioning instead obtains a k-way partition by recursively splitting the blocks in two parts. This is simpler as it only requires refinement algorithms that work on two-way partitions. Most partitioners [10, 16, 29, 32], including BiPart use recursive bipartitioning. However, recursive bipartitioning often achieves substantially worse solution quality than direct k-way [38], which is why we focus on direct k-way in this paper.

1.2 Non-Determinism in Local Moving

Typical community detection, clustering coarsening and label propagation refinement algorithms are so-called *local moving algorithms*, which follow the structure outlined in Algorithm 1: given an initial assignment of vertices to groups, visit vertices in random order in parallel, and improve the solution by greedily moving vertices when they are visited. Since vertices are moved right away, the local optimization decisions depend on non-deterministic scheduling decisions. Our approach to incorporate determinism is illustrated in Algorithm 2. It is based on the *synchronous local moving* approach of Hamann et al. [25] to parallelize the Louvain community detection algorithm [8] on distributed memory.

Instead of performing moves asynchronously, vertices are split into sub-rounds – using deterministically reproducible randomness. In each sub-round, the best move for each vertex in the current sub-round is computed with respect to the unmodified groups. In a second step, some of the calculated moves are approved and performed, and some are denied, for example due to the balance constraint.

1.3 Contributions

We propose three deterministic parallel local moving algorithms: for the preprocessing, coarsening and refinement phases of Mt-KaHyPar. An algorithmic novelty is the incorporation of vertex weights in the approval step of the refinement phase via a merge-style parallelization. Extensive

experiments demonstrate that our new algorithm achieves good speedups (28.7 geometric mean, 48.9 max on 64 threads) and achieves similar solution quality and performance as its non-deterministic counterpart (as expected slightly worse overall though). We investigate potential causes for this cost of determinism, finding that the coarsening phase is the most affected. Our algorithm outperforms BiPart regarding solution quality, running time and parallel speedups on 98% of the instances.

Our shared-memory implementation is available as open-source software as part of the Mt-KaHyPar framework. We describe engineering details, and also describe how to incorporate determinism in the rest of the framework.

The paper is organized as follows. In Section 2, we introduce concepts and notation. Subsequently, we describe our algorithmic components and the implementation in Section 3. In Section 4, we analyze the algorithm experimentally via a parameter study and comparison with existing algorithms, before concluding the paper in Section 5.

2 Preliminaries

By $[m]$ we denote the set $\{0, 1, \ldots, m-1\}$ for a positive integer m. We use Python-style slicing notation $A[i : j]$ to denote sub-arrays from index i up to (excluding) index j.

Hypergraphs. A *weighted hypergraph* $H = (V, E, c, \omega)$ is a set of vertices V and a set of hyperedges E with vertex weights $c : V \rightarrow \mathbb{N}$ and hyperedge weights $\omega : E \rightarrow \mathbb{N}$, where each hyperedge e is a subset of the vertex set V. The vertices of a hyperedge are called its *pins*. A vertex v is *incident* to a hyperedge e if $v \in e$. $I(v)$ denotes the set of all incident hyperedges of v. The *degree* of a vertex v is $d(v) := |I(v)|$. The *size* $|e|$ of a hyperedge e is the number of its pins. By $N(v) := \{u \in V \mid I(u) \cap I(v) \neq \emptyset\}$ we denote the *neighbors* of v. We extend c and ω to sets in the natural way $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$.

A graph is a hypergraph where each edge has size 2. We use the terms nodes and edges when referring to graphs, and vertices, hyperedges and pins when referring to hypergraphs.

Partitions. A *k-way partition* of a hypergraph H is a function $\Pi : V \rightarrow [k]$ that assigns each vertex to a *block* (or block identifier) $i \in [k]$. In addition to the identifiers, we also use the term block for their corresponding vertex sets $V_i := \Pi^{-1}(i)$. We call Π *ε-balanced* if each block V_i satisfies

the *balance constraint*: $c(V_i) \leq L_{\max} := (1 + \varepsilon)\lceil c(V)/k \rceil$ for some parameter $\varepsilon \in (0, 1)$.

For each hyperedge e, $\Lambda(e) := \{V_i \mid V_i \cap e \neq \emptyset\}$ denotes the *connectivity set* of e. The *connectivity* $\lambda(e)$ of a hyperedge e is $\lambda(e) := |\Lambda(e)|$. A hyperedge is called a *cut hyperedge* if $\lambda(e) > 1$. Given parameters ε, and k, and a hypergraph H, the *hypergraph partitioning problem* is to find an ε-balanced k-way partition Π that minimizes the *connectivity metric* $(\lambda - 1)(\Pi) := \sum_{e \in E}(\lambda(e) - 1)\,\omega(e)$. We use the term solution quality for connectivity metric in the experiments. In order to avoid confusion later on, note that, while these concepts are defined as functions, the pseudocodes in this paper treat them as data that are modified as the algorithms iteratively change the partition.

Contraction. A *clustering* (or set of *communities*) C is a partition without a restriction on the number of clusters (vertex blocks). *Contracting* a clustering yields a smaller, *coarser* hypergraph, by merging vertices assigned to the same cluster into a super-vertex whose weight is the sum of the constituents. Correspondingly, the hyperedges of the coarse hypergraph become smaller and duplicate hyperedges may arise. Duplicates are removed and their weight is aggregated at the sole non-removed representative.

Given a partition Π_c of a hypergraph H_c that was obtained from another hypergraph H by contracting a clustering C, the partition Π_c can be projected to H by assigning vertices to the block of their super-vertex $\Pi(v) = \Pi_c(C(v))$. The partition of H then has the same imbalance and objective function value as that of H_c. This is the fundamental property of the multilevel framework.

Parallel Primitives. We use parallel primitives such as prefix sums, reduce and sorting from Intel's tbb library. Whenever possible, we use our own counting sort (integer sorting) implementation [12, Section 8.2] with $O(\log(n) + K)$ depth and $O(n + K)$ work, where n is the size of the input, and K is the maximum key value.

A fundamental primitive in our algorithms is deterministic parallel random shuffling, or more often, randomly splitting elements into sub-rounds. Our algorithm is based on the RandSort and RandDist algorithms [11]. We divide the input range into a fixed number of equal-size chunks (256 regardless of number of threads used for reproducibility), which are handled independently in parallel. For each chunk, a random number generator is seeded with an input seed and the index of the first element, which is then used to generate 8-bit tags for each element in the chunk. The input range is then sorted by the tags using parallel counting sort. For splitting into sub-rounds, the algorithm stops here – counting sort returns an array of offsets where each tag starts. Multiple tags are further grouped together to form sub-rounds as needed. For random shuffling, the elements with the same tag are shuffled sequentially, but each tag independently in parallel. The

order within the same tag is deterministic because counting sort is stable.

3 Deterministic Parallel Multilevel Partitioning

In this section, we describe our algorithms. The structure follows the order of the multilevel framework. We introduce the community detection preprocessing in Section 3.1, the coarsening in Section 3.2, the initial partitioning in Section 3.3, and the direct k-way refinement in Section 3.4.

3.1 Preprocessing

The preprocessing phase detects communities in the hypergraph that are used to guide the coarsening process by restricting contractions to vertices in the same community, as proposed in [26]. We perform community detection on the bipartite graph representation of the input hypergraph with some modified edge weights to handle large hyperedges. The bipartite graph representation $G = (V_G, E_G)$ of a hypergraph $H = (V, E)$ has the vertices and hyperedges as nodes, and an edge for every pin connecting the vertex and hyperedge. Since this also assigns hyperedges to communities, we subsequently restrict the communities to the vertices. The edge weights are set to $w'(v, e) := \frac{\omega(e)|\mathrm{I}(v)|}{|e|}$ or $w'(v, e) := \omega(e)$ for pin $v \in e$ and hyperedge $e \in E$, depending on the density of the hypergraph as described in [26].

We heuristically maximize the well-known modularity objective using a synchronous parallel version of the popular Louvain algorithm [8]. The modularity of given communities C is $Q(C) := \mathrm{cov}(C) - \sum_{C \in \mathcal{C}} \mathrm{vol}(C)^2/\mathrm{vol}(V_G)^2$. Here, the coverage $\mathrm{cov}(C) := \sum_{C \in \mathcal{C}} \sum_{u \in C} \sum_{v \in C \cap N(u)} w'(u, v)/\mathrm{vol}(V_G)$ is the fraction of edge weights inside communities. The volume $\mathrm{vol}(u) := \sum_{v \in N(u)} w'(u, v)$ is the sum of incident edge weights of a node (counting self-loops twice), which is extended to node-sets $\mathrm{vol}(X) := \sum_{u \in X} \mathrm{vol}(u)$.

The Louvain algorithm starts with each node in its own community. In a round, it visits each node in a random order and greedily maximizes modularity by possibly moving the node to the community of a neighbor. After a fixed number of rounds or if no node has been moved in the last round, the communities are contracted and the algorithm is applied recursively to the contracted graph. This continues until no node has been moved on a level, at which point the community assignment is projected to the input graph.

The gain (modularity difference) of moving a node from its current community to a neighboring community can be computed purely from the weight of incident edges to the target or current community, as well as their volumes. Therefore, it suffices to store and update the volume for each community, and compute the weights to the communities by iterating once over the neighbors of the node.

We randomize the visit order by dividing nodes into random sub-rounds. For each sub-round we calculate the best

move for each node in parallel, but only apply volume and community assignment updates after synchronizing, to make the algorithm deterministic.

Volume Updates. One intricacy with updating the community volumes is that adding floating point numbers is not commutative, and thus the previous approach [22, 39] of applying all updates in parallel with compare-and-swap instructions is non-deterministic. Instead, we have to establish an order in which the volume updates of each community are aggregated. For this, we collect all necessary updates in a global vector, which we lexicographically sort by community (primary key) and node ID (secondary key). Applying the updates is done in parallel for different communities. To reduce the sorting overhead we split the updates into two vectors (addition and subtraction) which are sorted independently in parallel, but applied one after another.

To analyze the work and depth, let V'_G denote the nodes in a sub-round. The work is $\sum_{u \in V'_G} \deg(u) + |V'_G| \log(|V'_G|)$. The depth is linear in the maximum number of moves in or out of a community (sequential volume updates) and the maximum degree $\max_{u \in V'_G}(\deg(u))$ for calculating modularity gains, plus the depth of the sorting algorithm. This is usually poly-logarithmic in $|V'_G|$, but tbb's quick-sort implementation uses sequential partitioning, and thus is linear. We tested a supposedly better sorting algorithm but did not achieve an improvement. The number of moves and degree linear terms in the depth may be reduced to poly-logarithmic by parallelizing the per-vertex gain calculation (parallel for loop over neighbors, atomic fetch-add for weight to neighbor clusters) and aggregating updates within a community in parallel with a deterministic reduce. However, in practice the outer level of parallelism is sufficient.

Contraction. To contract the communities, we first remap the community IDs to a consecutive range. This is done by computing the prefix sum over an array with 1's in the locations of used community IDs. In a second pass, we remap the node-to-community assignment by looking up the old community ID in the prefix sum array.

Subsequently, we sort the nodes by community ID via counting sort, which also gives us an array of offsets into the sorted range where each community begins. We parallelize the generation of the coarse edges on a per-community level. For each community, one thread generates all of its outgoing coarse edges by iterating sequentially over the neighbors of the vertices in the community, and aggregating the edge weights in a vector indexed by community ID of the neighbor. Since the sorting algorithm is stable, the orders in which node volumes are aggregated and edges are generated are deterministic. The work is $O(|E_G|)$, and the depth is the largest degree sum over vertices in a community.

Algorithm 3: Compute Heavy-Edge Rating

Input: vertex $u \in V$
candidates $\leftarrow \emptyset$
for $e \in I(u)$ **do**
    for $v \in e$ **do**
        if $community[u] = community[v]$
            if $rating[C[v]] = 0$
                add $C[v]$ to candidates
            $rating[C[v]]$ += $\omega(e)/|e|$
for $C \in candidates$ **do**
    if $weight[C] + c(u) \le CW_{\max}$ *and* $rating[C] > best$
    rating
        store C as best candidate
    $rating[C] \leftarrow 0$
return best candidate

3.2 Coarsening

After performing community detection on the bipartite graph representation, we proceed to contracting the actual hypergraph. In the coarsening phase we keep performing coarsening passes over the vertices (Algorithm 4) until only a few vertices remain, at which point we can run initial partitioning. We choose this contraction limit $CL = 160 \cdot k$ dependent on k, as in [1, 22]. In each coarsening pass, we perform one round of local moving and then contract the resulting clusters. The objective function for the clustering is the commonly used [1, 10, 22] heavy-edge rating function $r(u, C) := \sum_{e \in I(u) \cap I(C)} \frac{\omega(e)}{|e|-1}$ which rewards heavy hyperedges between a vertex u and a potential target cluster C, but penalizes large hyperedges.

Initially, the clustering C is a singleton clustering, i.e., each vertex is in its own cluster. For each vertex u in a sub-round, we store the best target cluster according to the rating function in an array of propositions $\mathcal{P}$. Algorithm 3 shows pseudocode for calculating the ratings and Algorithm 4 shows pseudocode for one coarsening pass over the vertices. First, we aggregate the ratings in a sparse array indexed by cluster ID and store the potential candidates in a dense vector, before we select the highest-rated candidate and reset the ratings. To save running time, and since their contribution to the rating function is small, we skip hyperedges with size > 1000. This ensures that at most $O(|I(v)|)$ time is spent for vertex v (though the constant is large) instead of $O(\sum_{e \in I(v)} |e|)$, which leads to work linear in the number of pins per coarsening pass to compute target clusters. If there are multiple candidates with the same rating, we pick one uniformly at random – to achieve deterministic selection we use a hash-and-combine function seeded with u as a random number generator. In Algorithm 3, we could theoretically parallelize the iteration over neighbors by using atomic fetch-and-add instructions for aggregating the ratings. The check $rating[C[v]] = 0$ can be faithfully implemented because the

---

Algorithm 4: Coarsening Pass

---

randomly split vertices into sub-rounds
$C[u] \leftarrow u, \mathcal{P}[u] \leftarrow u \; : \; \forall u \in V$
opportunistic-weight$[u] \leftarrow c(u) \; : \; \forall u \in V$
for $r = 0$ **to** *number of sub-rounds* **do**
    for $u \in V$ *in sub-round* r **do in parallel**
        $\mathcal{P}[u] \leftarrow \text{ComputeHeavyEdgeRating}(u)$
        opportunistic-weight$[\mathcal{P}[u]] \mathrel{+}= c(u)$ // *atomic*
    $M \leftarrow \emptyset$ // *moves*
    for $u \in V$ *in sub-round* r **do in parallel**
        if *opportunistic-weight*$[\mathcal{P}[u]] \leq CW_{\max}$
            $C[u] \leftarrow \mathcal{P}[u]$
        else
            add u to M
    sort M lexicographically by $(\mathcal{P}[u], c(u), u)$
    for $i = 0$ **to** $|M|$ **do in parallel**
        if $i = 0$ **or** $\mathcal{P}[M[i-1]] \neq \mathcal{P}[M[i]]$
            for $j = i$ *until* $CW_{\max}$ *exceeded* **do**
                $C[M[j]] \leftarrow \mathcal{P}[M[j]]$
            set opportunistic-weight of $C[M[i-1]]$
contract clustering C

---

atomic instruction returns the value immediately prior to its execution. However, in practice the outer level of parallelism over the vertices is again sufficient.

Approving Moves. Since the initial partitioning step must be able to compute a feasible partition, we enforce a maximum weight on the clusters $CW_{\max} := \min\left(L_{\max}, c(V)/CL\right)$. To respect this constraint, we filter the target cluster candidates further during the selection. Additionally, some of the calculated moves must be rejected. Therefore, we sort the moves lexicographically by cluster, vertex weight, and lastly vertex ID (for determinism). For each target cluster, we then approve the vertices one by one (in order of ascending weight), and reject all of the remaining moves into this cluster once $CW_{\max}$ would be exceeded. Our implementation iterates over the moves in parallel, and the iteration of the first vertex in the sub-range of a cluster is responsible for performing the moves into the cluster. To drastically reduce the number of moves we have to sort, we employ an optimization, where we already sum up the cluster weights during the target-cluster calculation step using atomic fetch-and-add instructions, and simply approve all moves into a target cluster whose weight will not exceed $CW_{\max}$. Due to this optimization, calculating the target clusters is by far the more expensive step in practice, even though approving the moves requires sorting.

Contraction. The hypergraph contraction algorithm consists of several steps: remapping cluster IDs to a consecutive range (as for graph contraction), generating pin lists of the hyperedges of the contracted hypergraph, removing duplicate hyperedges, and finally assembling the data structure.

We generate the coarse pin list of each hyperedge in parallel, by replacing the vertex ID with the remapped cluster ID and removing duplicate entries. Our version uses a bit-set for de-duplication, but sorting is not much slower. At this stage we already discard hyperedges of size one.

To remove duplicate hyperedges, we use a parallel version of the INRSort algorithm [5, 15]. The INRSort algorithm works as follows. Comparing all hyperedge-pairs for equality is too expensive, so a hash function is used to restrict comparisons to hyperedges with equal hash value and size. For parallelism, hyperedges are distributed across threads using the hash value of their pins [22]. Each thread sorts its hyperedges by their hash value, their size, as well as ID for determinism. In each sub-range with equal hash value and size (consecutive in memory due to sorting), pair-wise comparisons of their pins are performed. Again, we use a bit-set to check for equality, as this was slightly faster than sorting the pins. The running time of the de-duplication algorithm is difficult to analyze, since it depends on the collision rate of the hash function, the number of duplicate hyperedges and their sizes. However, in practice, it is faster than constructing the pin lists and the incident hyperedge lists.

At this point, we have obtained the pin lists of the coarse hypergraph, and now need to construct the list of incident hyperedges at each vertex. For this, we first count the number of incident hyperedges at each vertex, and compute a prefix sum over these values. In a second pass, we write the incident hyperedges into the sub-ranges of the corresponding pins, using an atomic fetch-and-add instruction on the starting position of the sub-range. Finally, we sort the incident hyperedges of each vertex for determinism.

3.3 Initial Partitioning

After the coarsening phase, we compute an initial k-way partition on the coarsest hypergraph. We perform recursive bipartitioning with the multilevel algorithm and thus only need to provide *flat* algorithms for computing initial 2-way partitions. Since the coarsest hypergraphs are *small*, a portfolio of 9 different simple, sequential algorithms [37] is used. Combined with 20 repetitions each for diversity, there is ample parallelism. Each run is followed up with 3 rounds of sequential FM local search [18]. These algorithms are inherently deterministic, however care must be taken when selecting which partition to use for refinement. The primary criteria are connectivity followed by imbalance. To achieve deterministic selection, we assign sequentially generated tags to the initial bipartitions, which are used as a tie breaking mechanism. In combination with deterministic coarsening and refinement, the overall initial partitioning phase is deterministic.

Algorithm 5: Compute Max Gain Move

Input: vertex $v \in V$

gains$[i] \leftarrow 0 \; \forall i \in [k]$

internal $\leftarrow 0$

for $e \in I(v)$ **do**

   if $\Phi(e, \Pi[v]) > 1$

      | internal $+= \omega(e)$

   for *block* $i \in \Lambda(e)$ **do**

      | gains$[i] \mathrel{+}= \omega(e)$

$j \leftarrow \arg\max_{i \in [k]}(\text{gains}[i])$

return j, gains$[j]$ − internal

3.4 Refinement

In the refinement phase, we take an existing k-way partition (from the previous level or initial partitioning) and try to improve it by moving vertices to different parts, depending on their gain values. The gain of moving vertex $u \in V$ from its current block s to block t is $\text{gain}(u, t) := \sum_{e \in I(u): \Phi(e,s)=1} \omega(e) - \sum_{e \in I(u): \Phi(e,t)=0} \omega(e)$. The first term accounts for the hyperedges e for which s will be removed from their connectivity set $\Lambda(e)$, the second term accounts for those where t will be newly added.

Finding Moves. Our refinement algorithm is a synchronous version of label propagation refinement [22, 34]. The vertices are randomly split into sub-rounds. For each vertex in the current sub-round, we compute the highest gain move, and store it if the gain is positive. Algorithm 5 shows pseudocode for computing the gains of a vertex v to all k blocks, and selecting the highest gain move. As an optimization it uses the connectivity sets $\Lambda(e)$ instead of checking the pin counts $\Phi(e, i)$ for each block $i \in [k]$ directly. The gain-calculation phase takes $O(k|V| + \sum_{u \in V} \sum_{e \in I(u)} \lambda(e))$ work (across all sub-rounds) and $O(k + \max_{u \in V}(\sum_{e \in I(u)} \lambda(e)))$ depth for each sub-round. The $O(k)$ term per vertex for initializing the gains array and selecting the highest gain can be eliminated by tracking occupied slots and resetting only these, though this is not useful in practice if k is small.

In a second step we approve some of the stored moves, and subsequently apply them in parallel, before proceeding to the next sub-round. This is the interesting part, as just applying all moves does not guarantee a balanced partition.

Maintaining Balance By Vertex Swaps. In this step we perform a sequence of balance-preserving vertex swaps on each block-pair, prioritized by gain. This approach was first introduced in SHP [28], though their work only considers unweighted vertices as SHP is not a multilevel algorithm. For each block-pair $(s, t) \in \binom{[k]}{2}$, we collect the vertices M_{st} that want to move from s to t and M_{ts} from t to s, and sort both sequences descendingly by gain (with vertex ID as tie breaker for determinism). SHP now moves the first $\min(|M_{st}|, |M_{ts}|)$ vertices from each sequence. If each vertex

has unit weight, this does not change the balance of the partition. However, we have to handle weights, so we are interested in the longest prefixes of M_{st}, M_{ts}, represented by indices i, j, whose cumulative vertex weights $c(M_{st}[0 : i]), c(M_{ts}[0 : j])$ are equal. This is similar to merging two sorted arrays, as we describe in the next paragraph. We do not have to swap exactly equal weight, as long as the resulting partition is still balanced. For each block, we have a certain additional weight B_t it can take before becoming overloaded. If block-pairs are handled sequentially one after another, we can set $B_t = L_{\max} - c(V_t)$. If they are handled in parallel, we divide this budget (equally) among the different block-pairs that have moves into t.

Again, we denote the prefixes as indices i and j into M_{st} and M_{ts}, respectively. The prefixes i, j are called *feasible* if they satisfy the condition $-B_s \leq c(M_{st}[0 : i]) - c(M_{ts}[0 : j]) \leq B_t$, i.e., swapping the first i, j moves yields a balanced partition. To compute the two longest feasible prefixes of M_{st}, M_{ts}, we can iterate simultaneously through both sequences and keep track of the so far exchanged vertex weight $c(M_{st}[0 : i]) - c(M_{ts}[0 : j])$. If $c(M_{st}[0 : i]) - c(M_{ts}[0 : j]) < 0$ and M_{st} has moves left, we approve the next move from M_{st} by incrementing i. Otherwise we approve the next move from M_{ts}. In each iteration we check whether the current prefixes are feasible.

We parallelize this similar to a parallel merge algorithm. In a first step, we compute cumulative vertex weights via parallel prefix sums. Then the following algorithm is applied recursively. We search for the cumulative weight of the middle of the longer sequence in the shorter sequence using binary search. The left and right parts of the sequences can be searched independently. If the right parts contain feasible prefixes, we return them, otherwise we return the result from the left parts. The top-level recursive call on the left parts is guaranteed to find at least $i = j = 0$ (no move applied). If n denotes the length of the longer sequence, this algorithm has $O(\log(n)^2)$ depth and $O(n)$ work.

Since we are interested in the longest prefixes, we can omit the recursive call on the left parts if the prefixes at the splitting points are feasible. Depending on the available budgets B_s, B_t this is fairly likely, since the cumulative weights are as close as possible. Further, we can omit the recursive call on the right parts if the cumulative weight at the middle of the longer sequence exceeds the cumulative weight at the end of the shorter sequence plus the appropriate budget. Note that in this case the binary search finds the end and thus the left recursion takes the entirety of the shorter sequence.

We stop the recursion and run the sequential algorithm if both sequences have less than 2000 elements. This value worked well in preliminary experiments. As we already computed cumulative weights, we instead perform the simultaneous traversal from the ends of the sequences. Since we expect to approve the majority of the saved moves, this is likely faster.

Algorithm 6: Perform Move

Input: vertex $v \in V$ to be moved from block s to t
attributed $\leftarrow 0$
for $e \in I(v)$ **do**
    lock(e)
    if $(\Phi(e, s) \mathrel{-}= 1) = 0$
        attributed $\mathrel{+}= \omega(e)$
        remove s from $\Lambda(e)$
    if $(\Phi(e, t) \mathrel{+}= 1) = 1$
        attributed $\mathrel{-}= \omega(e)$
        add t to $\Lambda(e)$
    unlock(e)
$\Pi[v] \leftarrow t$
part-weight$[s] \mathrel{-}= c(v)$ // *atomic*
part-weight$[t] \mathrel{+}= c(v)$ // *atomic*
return attributed

Further Implementation Details. If we move vertices in parallel, the computed gains are not correct, as moving a vertex impacts the gains of its neighbors. However, we can still obtain the sum of the exact gains using a technique dubbed attributed gains [22] which is outlined in Algorithm 6, where we also describe the data structure updates incurred by a vertex move. For each incident hyperedge e of the moved vertex, we increment the pin count $\Phi(e, t)$ in the target block t and decrement $\Phi(e, s)$ in the source block s. If $\Phi(e, s)$ becomes zero, we attribute an $\omega(e)$ connectivity improvement, similarly if $\Phi(e, t)$ becomes one we attribute an $\omega(e)$ loss. The sum of these attributed gains is equal to the overall improvement, though each individual attributed gain may not be correct. If the overall attributed gain is negative, we made the solution worse. Since we cannot single out particular bad moves, we instead revert to the partition before the sub-round. A good strategy to avoid this in future iterations is to increase the number of sub-rounds for the current level. However, this happens predominantly in the last round on the level, which is why we observed little impact.

To speed up the gain-calculation phase, we employ an optimization that is commonly known as *active sets*. We perform multiple rounds (consisting of sub-rounds) of the refinement algorithm. Starting from the second round, only neighbors of vertices that were moved in the previous round are considered. We implement this by activating neighbors of moved vertices, using compare-and-swap instructions on an array storing the last round in which a vertex was activated to avoid duplicate insertion. To achieve linear work in the number of pins, we also use this mechanism to scan each hyperedge at most once per round. At the beginning of the next round, the collected neighbors are sorted for determinism, before being split into sub-rounds.

3.5 Differences to BiPart

We now discuss differences between our algorithm and `BiPart`. As already mentioned, `BiPart` uses recursive bipartitioning, whereas we use direct k-way, which is superior regarding solution quality [38]. Other than 2-way versus k-way and using sub-rounds and active sets, the refinement algorithms are largely the same, inspired by label propagation [34] and SHP [28]. However, their refinement ignores vertex weights (apply all moves of the shorter sequence and the same number from the longer), which leads to imbalanced partitions that must be repaired by explicit rebalancing. This can be slow and offers little control by how much solution quality degrades. Our refinement guarantees balanced partitions at all stages without rebalancing. Furthermore, `BiPart` uses no mechanism to track actual improvements, whereas we use attributed gains to detect and prevent quality-degrading moves. Their coarsening scheme assigns each vertex to its smallest incident hyperedge (ties broken by ID) and merges all vertices assigned to the same hyperedge. This offers no control over vertex weights and does not rank higher hyperedge weights as more important to not cut. Preventing large vertex weights is important so that initial partitioning can find balanced partitions and there is more leeway for optimization in initial partitioning and refinement. `BiPart` uses a parallel version of greedy graph growing [10] for initial partitioning, even though the coarsest hypergraphs are small, where it is feasible to afford many diversified, parallel repetitions of sequential algorithms.

4 Experiments

Our code is integrated in the `Mt-KaHyPar` hypergraph partitioning framework. It is written in `C++17`, uses Intel's `tbb` library for parallelization and is compiled with `g++` version 9.2 with optimization level `-O3` and native architecture optimizations. The experiments are run on a 128-core (2 sockets, 64 cores each) AMD EPYC Zen 2 7742 CPU clocked at 2.25GHz (3.4GHz turbo boost) with 1TB DDR4 RAM, and 256MB L3 cache.

4.1 Benchmark Set

We use the established benchmark set of 94 large hypergraphs that was assembled to evaluate `Mt-KaHyPar`, set B in [22]. It contains hypergraphs from three different sources: VLSI instances from the DAC 2012 Routability-Driven Placement Contest [40], various large sparse matrices from the SuiteSparse Matrix Collection [14], and three different representations (literal, primal, dual) of SAT formulas from the 2014 SAT Competition [6]. We use $k \in \{2, 4, 8, 16, 32, 64\}$, $\varepsilon = 0.03$, and partition each instance five times with different random seeds. Since we have 94 instances, we cannot report instance sizes for each of them, however these statistics are available online[1] in the supplementary material of [22]. The
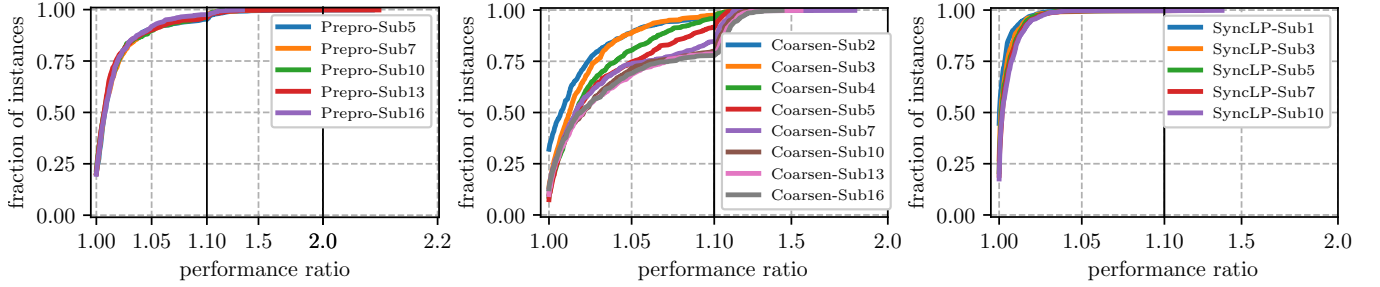
---

[1] https://algo2.iti.kit.edu/heuer/alenex21/

Figure 1. Performance profiles comparing the impact of the number of sub-rounds parameter on the different phases.
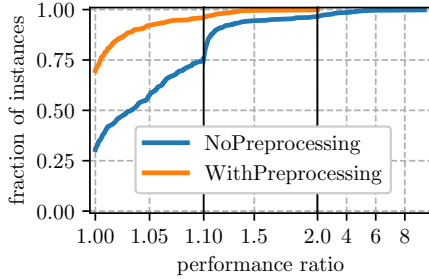

Figure 2. Solution quality with and without preprocessing.

largest instances have between 10^7 and 10^8 vertices and hyperedges, as well as 10^8 to $2 \cdot 10^9$ pins. Dual SAT instances are known for large hyperedges (with up to millions of pins), and hence their corresponding primal counterparts are known for large vertex degrees. Some sparse matrices are even more skewed with a maximum degree of 10^7.

4.2 Configurations

We perform 5 rounds of local moving on each level during refinement, 5 rounds before contracting during preprocessing, and one round before contracting during coarsening. We call the algorithm and configuration proposed in this work `Mt-KaHyPar-SDet`, and the equivalent configuration that uses the existing non-deterministic local moving algorithms `Mt-KaHyPar-S`, where `Det` stands for determinism, and `S` for speed. Additionally, we consider the configuration `Mt-KaHyPar-D` (for default) from [22], which has a different initial partitioning configuration that is non-deterministic [23] and additionally uses parallel localized FM – a more advanced refinement algorithm that is difficult to make deterministic. The parallel n-level version from [23] is excluded here since it represents a vastly different time-quality trade-off.

4.3 Performance Profiles

To compare the solution quality of different algorithms, we use *performance profiles* [17]. Let $\mathcal{A}$ be the set of all algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where τ is on the x-axis.

Achieving higher fractions at equal τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best. To interpret these plots, we either look at how quickly the curve converges towards $y = 1$ (higher is better), or we look at the maximum ratio for certain instance fraction quantiles. To calculate the ratios, we take the average connectivity across different seeds for each instance. In addition to the plots, we report the τ ratios (performance ratios) at certain quantiles, as well as their geometric mean.

4.4 Parameter Tuning

The supposedly most important parameter is the number of sub-rounds used, as it offers a trade-off between scalability (synchronization after each sub-round) and solution quality (more up-to-date information). In the following, we show that this is actually not a trade-off, as the number of sub-rounds either does not affect solution quality, or using fewer sub-rounds even leads to better quality.

Sub-rounds. We made an initial guess of 5 sub-rounds for refinement, and 16 sub-rounds for coarsening and pre-processing, which we use as a baseline configuration when varying each parameter. Figure 1 shows the performance profiles. The largest impact is on the coarsening phase, where 2 sub-rounds performs the best. Such a small value is surprising, yet one possible explanation is that high-degree vertices attract low-degree vertices too quickly if synchronization happens too frequently. Using only 1 sub-round is excluded here, since the clustering oscillates, which leads to coarsening converging long before the contraction limit is reached and thus initial partitioning takes very long. Furthermore, using 2 sub-rounds is about 12% slower than using 3 sub-rounds in the geometric mean, again due to the same effect. Since it gives only slightly worse solution quality, we choose 3 sub-rounds for coarsening in the main experiments. For preprocessing, there is little impact on solution quality. Here we stick with our original choice of 16 sub-rounds since the floating-point-aggregation handling becomes substantially slower if more vertices are in a sub-round due to the sorting overhead. For refinement, there is again little difference, where 1 sub-round narrowly emerges as the best choice. This
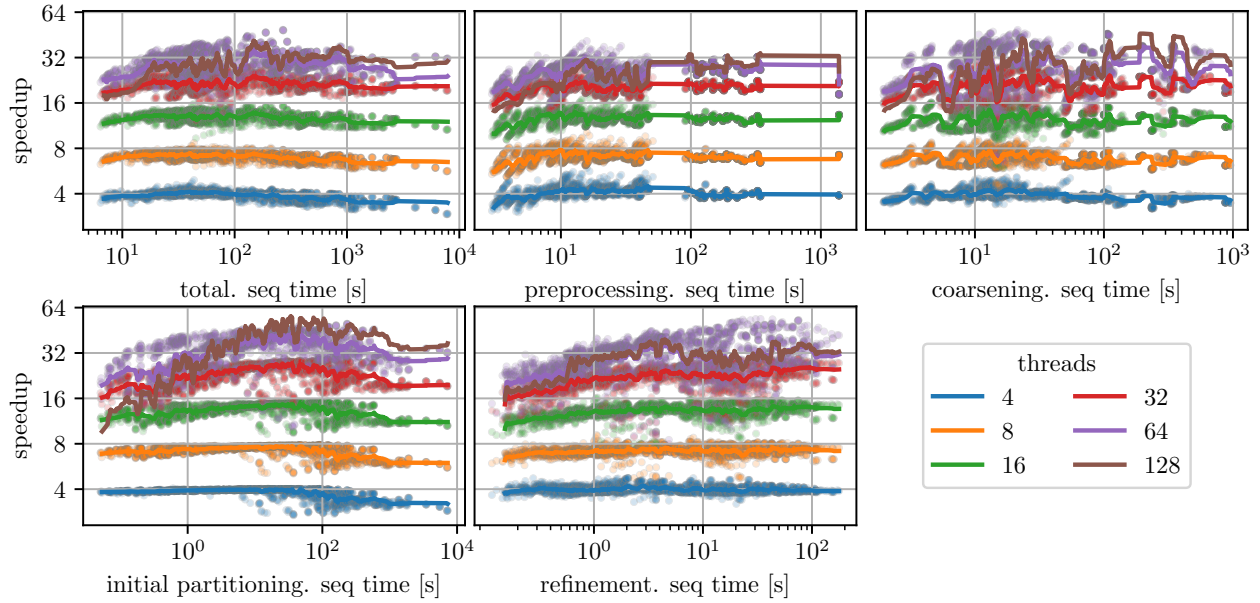
Figure 3. Speedups for `Mt-KaHyPar-SDet` in total as well as its components separately. The x-axis shows the sequential time in seconds, the y-axis the speedup. The lines are rolling geometric means (window size 50) of the per-instance speedups (scatter).
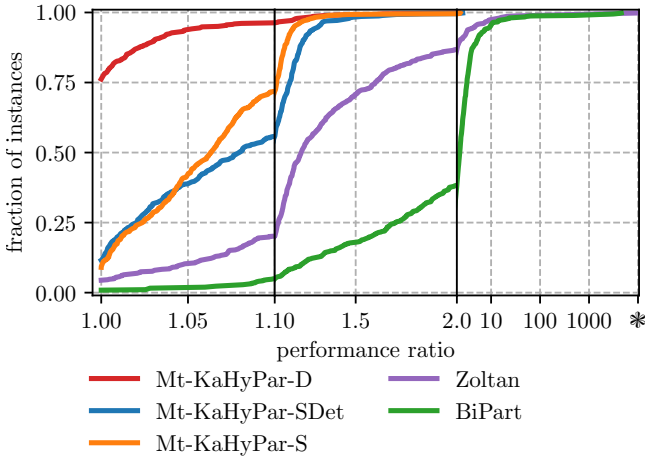

Figure 4. Performance profiles comparing the solution quality of `BiPart`, `Zoltan`, our algorithm `Mt-KaHyPar-SDet` and the existing `Mt-KaHyPar` variants. The ✳ symbol marks segmentation faults (6 instances for `Zoltan`).

is again surprising, as frequently synchronizing should allow for more informed move-decisions. One cause we noticed is that with more sub-rounds the pair-wise swaps did not have sufficiently many moves to balance, as moves from earlier sub-rounds are not considered. Using such moves as back-up could be included in future versions of the algorithm.

Impact of Preprocessing. In Figure 2, we show that the preprocessing phase is important for solution quality, which justifies the overhead for the floating-point volume updates.

4.5 Scalability

In Figure 3 we show self-relative speedups of the overall algorithm and the separate components, plotted against the sequential running time on that particular instance. In addition to the scatter plot, we show rolling geometric means with window size 50. The overall geometric mean speedups of the full partitioning process are 3.91, 7.04, 12.79, 21.32, 28.73, 29.09 for 4, 8, 16, 32, 64, and 128 threads, respectively, and the maximum speedups are 4.9, 8.7, 15.8, 29.1, 48.9, and 72.6. Since our algorithms are memory-bound workload types these are very good results. On about 37% of the runs with 4 threads, and 0.32% of runs with 8 threads, we observe super-linear speedups which occur in all phases except initial partitioning. We identified two reasons for this. First, even sequential runs had running time fluctuations, and as super-linear speedups occur mostly for small sequential times, the speedups are more easily affected. Second, while most of the work performed is deterministic, in all phases except initial partitioning we sort vectors that are filled in non-deterministic order. Sorting algorithms have checks for pre-sorted sub-sequences to speed up execution.

Looking at speedups for the indidivual phases, we see that most phases exhibit very consistent speedups, even for small sequential running times. Only initial partitioning exhibits
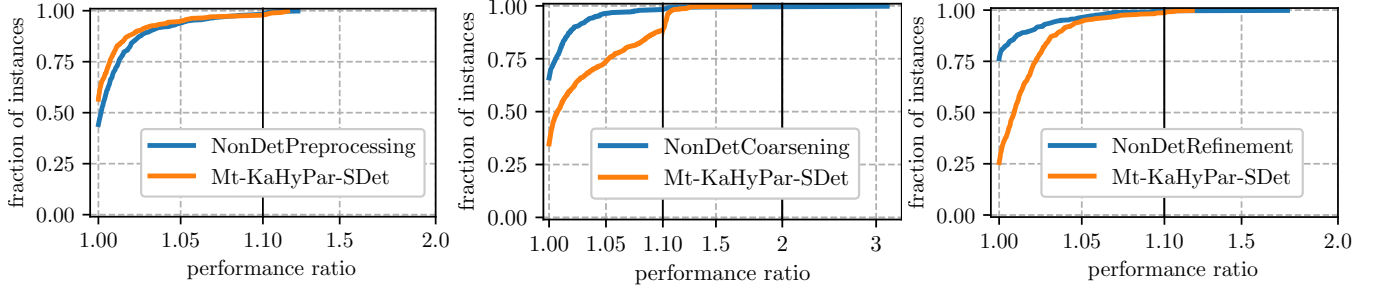
Figure 5. Performance profiles illustrating the quality impact of determinism in each component.
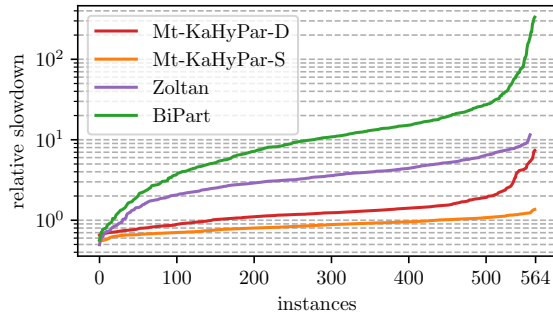

Figure 6. Slowdown of the other algorithms relative to Mt-KaHyPar-SDet, each using 64 threads. The instances on the x-axis are sorted independently for each algorithm.
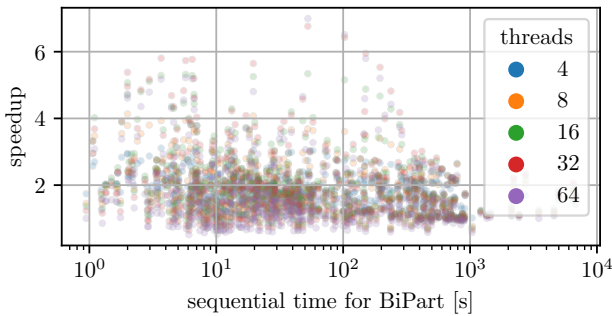

Figure 7. Speedups for BiPart.

4.6 Comparison with other Algorithms

Figure 4 shows performance profiles comparing our new algorithm Mt-KaHyPar-SDet with its non-deterministic variant Mt-KaHyPar-S, the stronger variant Mt-KaHyPar-D which uses parallel FM, the non-deterministic distributed algorithm Zoltan [16] as well as the deterministic BiPart algorithm [32]. In these experiments, each algorithm is run with 64 threads. As expected, Mt-KaHyPar-D performs best, contributing the best solutions on about 75% of the instances, followed by Mt-KaHyPar-SDet and Mt-KaHyPar-S which are similar, though Mt-KaHyPar-S is slightly better as it converges faster towards 1. BiPart is far off, contributing only 6 of the best solutions, and its quality is off by more than a factor of 2 on more than 50% of the instances; on some instances even by three orders of magnitude. Zoltan is situated between BiPart and Mt-KaHyPar-S. In a direct comparison, Mt-KaHyPar-SDet computes better partitions than BiPart on 551 of the 564 instances with a geometric mean performance ratio of 1.0032 compared to BiPart's 2.3805. In Figure 6, we report relative slowdowns, i.e., the running time of the other algorithm divided by running time of the baseline Mt-KaHyPar-SDet. Mt-KaHyPar-S is faster on all but 158 instances and never by a factor of more than 2. BiPart is between one and two *orders of magnitude* slower than the two speed variants of Mt-KaHyPar. The reason for this is shown in Figure 7 which shows self-relative speedups of BiPart for increasing number of threads. Most speedups are below 2 and the largest speedup is about 7.

sub-par speedups on larger instances, which is due to load imbalance from long running sequential FM refinement.

With 128 threads (only rolling geometric means shown for readability), the running times still improve, though not as drastically. Only small instances show a slight slowdown, predominantly in initial partitioning. Starting at > 64 threads, the second memory socket is used, so some slowdown is expected. We use interleaved memory allocations to cope with NUMA effects as much as possible.

4.7 The Cost of Determinism

In this section, we investigate in which phase the solution quality of Mt-KaHyPar-SDet gets lost, by swapping out one component for its non-deterministic counterpart, in each of the plots in Figure 5. Interestingly, the biggest quality loss comes from coarsening, whereas deterministic preprocessing even improves quality. The loss in refinement is expected due to the lack of up-to-date gains and the inability to leverage zero gain moves for rebalancing and diversification [22].

For coarsening, the results are unexpected, particularly because similar local moving algorithms [25] are not as affected by out-of-date gains. One reason for this is that multiple global rounds are performed, where vertices can back out of their first cluster assignment. We only use one round and even prematurely terminate the round to avoid coarsening too aggressively. Performing a second round, where only already clustered vertices may reassess their assignment, may be beneficial and we leave this for future research. Additionally, we unsuccessfully experimented with several features of the non-deterministic coarsening such as adapting hyperedge sizes to the current clustering in the rating function and *stable leader chasing*, where oscillations (vertices joining each other) and cyclic joins are resolved by merging all involved vertices.

5 Conclusion and Future Work

We presented the first scalable, deterministic parallel hypergraph partitioning algorithm. Our experiments show that determinism does incur sacrifices regarding both solution quality and running time compared to the previous non-deterministic version, but these are small enough to justify if determinism is desirable. Future work includes incorporating determinism into additional refinement algorithms, improving performance on multi-socket machines, and implementing these techniques for distributed memory.

For example for flow-based refinement [19, 21] this is well within reach, as scheduling on block-pairs can synchronize after each block was involved in a refinement step, and the flow algorithms need not be deterministic since the used cuts are unique. Parallel localized FM seems like a much more difficult target, though a promising approach may be to stick with approving expansion steps at synchronization points. Additionally, handling extremely large k and speeding up initial partitioning is possible by employing the deep multilevel approach [20] instead of recursive bipartitioning during initial partitioning.

Acknowledgments

References

[1] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2017. Engineering a Direct k-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 28–42. https://doi.org/10.1137/1.9781611974768.3

[2] Charles J. Alpert. 1998. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*. 80–85. https://doi.org/10.1145/274535.274546

[3] Charles J. Alpert and Andrew B. Kahng. 1995. Recent Directions in Netlist Partitioning: A Survey. *Integration* 19, 1-2 (1995), 1–81. https://doi.org/10.1016/0167-9260(95)00008-4

[4] Konstantin Andreev and Harald Räcke. 2004. Balanced Graph Partitioning. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*. 120–124. https://doi.org/10.1145/1007912.1007931

[5] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. 2008. Multi-level Direct k-Way Hypergraph Partitioning With Multiple Constraints and Fixed Vertices. *Journal of Parallel Distributed Computing* 68, 5 (2008), 609–625. https://doi.org/10.1016/j.jpdc.2007.09.006

[6] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. 2014. The SAT Competition 2014. http://www.satcompetition.org/2014/.

[7] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan (Eds.). ACM, 181–192. https://doi.org/10.1145/2145816.2145840

[8] Vincent D. Blondel, Jean Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment* 10 (2008). https://doi.org/10.1088/1742-5468/2008/10/P10008

[9] Robert L Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic By Default. *Usenix HotPar* 6 (2009).

[10] Ümit V. Catalyurek and Cevdet Aykanat. 1999. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 673–693. https://doi.org/10.1109/71.780863

[11] Guojing Cong and David A. Bader. 2005. An Empirical Analysis of Parallel Random Permutation Algorithms ON SMPs. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, USA*, Michael J. Oudshoorn and Sanguthevar Rajasekaran (Eds.). ISCA, 27–34.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms* (second ed.). MIT Press.

[13] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment* 3, 1 (2010), 48–57. https://doi.org/10.14778/1920841.1920853

[14] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (11 2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[15] Mehmet Deveci, Kamer Kaya, and Ümit V. Çatalyürek. 2013. Hypergraph Sparsification and Its Application to Partitioning. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*. 200–209. https://doi.org/10.1109/ICPP.2013.29

[16] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Catalyürek. 2006. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE. https://doi.org/10.1109/IPDPS.2006.1639359

[17] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming* 91, 2 (2002), 201–213. https://doi.org/10.1007/s101070100263

[18] Charles M. Fiduccia and Robert M. Mattheyses. 1982. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*. 175–181. https://doi.org/10.1145/800263.809204

[19] Lars Gottesbüren, Michael Hamann, and Dorothea Wagner. 2019. Evaluation of a Flow-Based Hypergraph Bipartitioning Algorithm. In *27th European Symposium on Algorithms (ESA)*. 52:1–52:17. https://doi.org/10.4230/LIPIcs.ESA.2019.52

[20] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep Multilevel Graph Partitioning. (2021).

https://arxiv.org/abs/2105.02022

[21] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. 2020. Advanced Flow-Based Multilevel Hypergraph Partitioning. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*. 11:1–11:15. https://doi.org/10.4230/LIPIcs.SEA.2020.11

[22] 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 2021, Virtual Conference, January 10-11, 2021*. 16–30. https://doi.org/10.1137/1.9781611976472.2

[23] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. 2021. Shared-Memory n-level Hypergraph Partitioning. *arxiv preprint* (2021). https://arxiv.org/abs/2104.08107

[24] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-Optimized Tensor Network Contraction. *Quantum* 5 (2021), 410.

[25] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. 2018. Distributed Graph Clustering Using Modularity and Map Equation. In *European Conference on Parallel Processing (Euro-Par)*. 688–702. https://doi.org/10.1007/978-3-319-96983-1_49

[26] Tobias Heuer and Sebastian Schlag. 2017. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 21:1–21:19. https://doi.org/10.4230/LIPIcs.SEA.2017.21

[27] Guy L. Steele Jr. 1990. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, Frances E. Allen (Ed.). ACM Press, 218–231. https://doi.org/10.1145/96709.96731

[28] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. 2017. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Proceedings of the VLDB Endowment* 10, 11, 1418–1429. https://doi.org/10.14778/3137628.3137650

[29] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1999. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 1 (1999), 69–79. https://doi.org/10.1109/92.748202

[30] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (2006), 33–42. https://doi.org/10.1109/MC.2006.180

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

[32] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. 2021. BiPart: A Parallel and Deterministic Multilevel Hypergraph Partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 161–174. https://doi.org/10.1145/3437801.3441611

[33] Zoltán Ádám Mann and Pál András Papp. 2014. Formula Partitioning Revisited. In *5th Pragmatics of SAT Workshop*. 41–56. https://doi.org/10.29007/9skn

[34] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel Graph Partitioning for Complex Networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638. https://doi.org/10.1109/TPDS.2017.2671868

[35] Walter Lau Neto, Max Austin, Scott Temple, Luca G. Amarù, Xifan Tang, and Pierre-Emmanuel Gaillardon. 2019. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence. In *Proceedings of the International Conference on Computer-Aided Design, IC-CAD 2019, Westminster, CO, USA, November 4-7, 2019*. 1–6. https://doi.org/10.1109/ICCAD45719.2019.8942145

[36] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E* 76, 3 (2007), 036106. https://doi.org/10.1103/PhysRevE.76.036106

[37] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. *k*-way Hypergraph Partitioning via n-Level Recursive Bisection. In *18th Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, 53–67. https://doi.org/10.1137/1.9781611974317.5

[38] Horst D. Simon and Shang-Hua Teng. 1997. How Good is Recursive Bisection? *SIAM Journal of Scientific Computing* 18, 5 (1997), 1436–1445. https://doi.org/10.1137/S1064827593255135

[39] Christian L. Staudt and Henning Meyerhenke. 2016. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (01 2016), 171–184. https://doi.org/10.1109/TPDS.2015.2390633

[40] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. 2012. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*. ACM, 774–782. https://doi.org/10.1145/2228360.2228500