Engineering Dynamic Hypergraph Partitioning

Bachelor Thesis of
Tobias Kempf

At the Department of Computer Science
Institute of Theoretical Informatics (ITI)

Reviewers:	Prof. Dr. rer. nat. Peter Sanders
	T.T.-Prof. Dr. Thomas Bläsius
Advisors:	Nikolai Maas
	Daniel Seemaier
	Lars Gottesbüren

08.05.2025

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 08. Mai 2025

- -

(Tobias Kempf)

Abstract

Graph partitioning is an essential problem in computer science, with numerous applications, including large social networks or distributed databases. These require partitioning the graph into k blocks while satisfying a balance constraint. A generalised form of graphs, called *hypergraph*, allows for more accurate representations of complex problems. A subsequent topic is *dynamic graph partitioning*, requiring the partitioning of a graph that is subject to the addition or removal of vertices, edges or pins. The aim of this thesis is to develop novel algorithms that are capable of partitioning *hypergraphs* that are subject to fully dynamic changes. To the best of our knowledge, this thesis presents the first fully dynamic hypergraph partitioner.

We show that by combining advanced techniques of static hypergraph partitioning with incremental techniques, we are able to reach high quality partitions in a fraction of the time required to use a static partitioner after every change. The proposed algorithms are implemented in the hypergraph partitioner Mt_KaHyPar. We generate synthetic changes for a set of 100 hypergraphs to evaluate the algorithms.

Zusammenfassung

Ein wesentliches Problem der Informatik ist die Partitionierung von Graphen in k verschiedene Blöcke unter Einhaltung einer Gleichgewichtsbedingung. Diese Partitionierung ist NP-vollständig und hat zahlreiche Anwendungen wie große soziale Netzwerke oder verteilte Datenbanken. *Hypergraphen*, eine verallgemeinerte Form von Graphen, erlauben eine genauere Darstellung dieser Probleme und damit eine effizientere Lösung. Ein typisches weiterführendes Problem ist die *dynamische Graphenpartitionierung*, die die Partitionierung eines Graphen erfordert, der Veränderungen unterliegt. Änderungen können das Hinzufügen oder Entfernen von Knoten, Kanten oder deren Zuordnungen sein. Diese Arbeit entwickelt Algorithmen, die in der Lage sind, *voll-dynamische Änderungen* für *Hypergraphen* zu verarbeiten. Soweit wir wissen, ist dies die erste Arbeit, die sich mit voll-dynamischer Hypergraph Partitionierung beschäftigt

Wir zeigen, dass wir durch die Kombination von fortgeschrittenen Techniken der statischen Hypergraphen Partitionierung mit inkrementellen Techniken in der Lage sind, qualitativ hochwertige Partitionierungen in einem Bruchteil der Zeit zu erreichen, die für eine Neupartitionierung nach jeder Änderung erforderlich wäre. Um die Algorithmen zu evaluieren, generieren wir synthetische Änderungen für bestehende nicht-dynamische Hypergraphen. Die vorgeschlagenen Algorithmen werden im Hypergraph Partitionierer Mt_KaHyPar implementiert. Wir evaluieren diese Algorithmen auf einer Menge von 100 Hypergraphen.

Contents

1 Introduction

Graphs are a common and widely used way of modelling a variety of theoretical and real-world problems. These are composed of vertices and their connections, known as edges. A graph requires edges to connect exactly two vertices. We focus on a more generalised form of graphs, called *hypergraphs*. These allow for *hyperedges* to connect an arbitrary number of vertices. This allows for more accurate problem representations.

K-way balanced (hyper)graph partitioning is a common NP-hard [1] problem with many applications, including social networks [2], very-large-scale integration (VLSI) design [3], load balancing [4], machine learning [5] and distributed databases [6], [7], [8]. (Hyper)Graph partitioning requires the partitioning of a graph into a number of blocks, while ensuring, that these blocks are of similar size.

In the context of distributed databases, hypergraphs can be used to accurately model data records and queries. The partition of the hypergraph is used to distribute the data records across different servers, thereby ensuring that queries access a minimal number of servers. This reduces the communication volume and improves query latency.

Many applications of (hyper)graph partitioning, including but not limited to distributed databases or social networks, are subject to frequent, minor changes. Here it is desirable to use a *dynamic graph partitioning* algorithm that *updates* the partition instead of recomputing it from scratch, resulting in a significantly reduced running time.

Our focus is on the *fully dynamic* ***hypergraph*** *partitioning* problem. We allow for arbitrary changes to the hypergraph, including deletions, and utilize the migration of vertices after they have been placed. Unlike other dynamic graph partitioners, we focus on the more general problem of hypergraph partitioning.

To the best of our knowledge, no work has been done in the area of fully dynamic hypergraph partitioning. Thus, we explore several different strategies to present multiple variants of the arising trade-off between running time and quality. We expect that by utilizing the information of the previous partition, we can achieve a high quality partition in a fraction of the time required to repartition the hypergraph from scratch.

1.1 Problem Statement

The objective of this thesis is to develop algorithms that are capable of partitioning hypergraphs that are subject to the addition and removal of vertices, edges and pins (*fully dynamic changes*).

The proposed algorithms should provide partitions significantly faster than partitioning the hypergraph from scratch after every change, while maintaining a comparable quality.

The algorithms should be evaluated on a large dataset for quality and running time.

1.2 Contribution

The primary contributions of this thesis are the development and implementation of advanced hypergraph partitioning algorithms capable of processing fully dynamic changes.

We combine fast local improvements with occasional global passes over the data to achieve a high quality partition. We show that by applying an advanced refinement algorithm (*FM*) only to a small selection of vertices we can achieve comparable quality in much less time.

We incrementalize existing structures (*rebalancers*) to improve the running time and open up new opportunities for long term global improvements.

We utilize high quality improvement techniques (*v-cycles*), used to improve partitions of static partitioners, to further improve the quality of our algorithms.

We implement the developed algorithms in the hypergraph partitioner Mt-KaHyPar and evaluate them on two types of synthetic changes on a set of 100 hypergraphs. We show that we can compute high quality partitions several orders of magnitude faster than the time required by naively partitioning the hypergraph from scratch after every change (*repartitioning*).

1.3 Outline

The thesis is structured as follows:
In Section 2 we introduce the basic concepts and notation of hypergraphs and (dynamic) partitioning that is used throughout the thesis. We then present related work in the field of dynamic graph partitioning in Section 3. In Section 4 we introduce multiple algorithms to process changes in a dynamic hypergraph. We start by setting baselines for quality and running time in Section 4.1. Then we present our proposed algorithms in Section 4.2, Section 4.3 and Section 4.4. In Section 5 we evaluate the algorithms on a large set of hypergraphs and compare them to a state-of-the-art streaming partitioner. We conclude the thesis with a summary of the results and a discussion of possible future work in Section 6.

2 Preliminaries

A *hypergraph* $H = (V, E)$ consists of a set of n vertices V with vertex *weights* $w : V \to \mathbb{R}_0^+$ and a set of m *hyperedges* E with *costs* $c : E \to \mathbb{R}_0^+$. A *hyperedge* $e \in E$ has a unique set of vertices $e_V \subseteq V$ called *pins*. A vertex $v \in V$ is *incident* to a hyperedge e if $e \in I(v)$. The *size* of a hyperedge $e \in E$ is the number of pins it has $|e_V|$. The *degree* of a vertex $v \in V$ is the number of hyperedges it is a pin of $d(v) = |I(v)|$. A vertex that is not a pin of any hyperedge, i.e. $d(v) = 0$, is called a *degree-0-vertex*. The *neighbours* of a vertex $v \in V$ are the vertices that are pins of a hyperedge incident to v $N(v) = \{u \in V \mid \exists e \in I(v) : v \in e \land u \in e\}$. A vertex u is called *adjacent* to a vertex v if $u \in N(v)$.

A *graph* $G = (V, E)$ is a special case of a hypergraph where all hyperedges have size 2 $\forall e : |e_V| = 2$.

A *ε-balanced partitioned hypergraph* is a hypergraph with Blocks $\Pi = \{V_1, ..., V_k\}$ where each vertex v is assigned to a block V_i and the weight of each block has to be within a given range of the average block weight $c(V_i) = \sum_{v \in V_i} w(v) \leq (1 + \varepsilon) \lceil \sum_{v \in V} \frac{w(v)}{k} \rceil$.

To *rebalance* a hypergraph means to move vertices between blocks to reduce the imbalance of the partition in order to fulfill the balance constraint.

The *connectivity* of a hyperedge $e \in E$ is the number of different blocks its pins are in $\lambda(e) = |\{V_i \in \{V_1, ..., V_k\} \mid V_i \cap e_V \neq \emptyset\}|$. The connectivity set of a hyperedge e is the set of blocks it is connected to $\Lambda(e) = \{V_i \in \{V_1, ..., V_k\} \mid V_i \cap e_V \neq \emptyset\}$.

We use the *connectivity metric* to evaluate the quality of a partition. It is the sum of the costs of all hyperedges that are cut by the partition $(\lambda - 1)(\Pi) = \sum_{e \in E} c(e) \cdot (\text{connectivity}(e) - 1)$.

The *gain* of a vertex $v \in V$ is the change in connectivity metric if the vertex is moved to a different block $\text{gain}(v, V_i, V_j)$.

The problem we attempt to solve is the *dynamic hypergraph partitioning problem*. It is defined as follows:
Given a hypergraph $H = (V, E)$ with a partition $\Pi = \{V_1, ..., V_k\}$ and a set of changes ΔG. Consider $H' = H \oplus \Delta G$ beeing the hypergraph with the changes applied to it. Find a new partition $\Pi' = \{V_1', ..., V_k'\}$ for the hypergraph H' that minimizes the connectivity metric and satisfies the balance constraint in a feasable amount of time.

⬡ **Lemma 1**: The dynamic hypergraph partitioning problem is NP-hard.

Proof of Lemma 1:

Assume, that there exists an algorithm that solves the dynamic hypergraph partitioning problem in polynomial time, say in time Δ per update.

Let $H = (V, E)$ be a hypergraph instance of the static partitioning problem, where $|V| = n$. We construct this hypergraph dynamically by starting with an empty hypergraph (which is trivially partitioned) and incrementally add the n vertices (along with the associated edges, as needed) one at a time. After each update, we apply the assumed dynamic partitioning algorithm to maintain a valid partition.

As a result, the complete hypergraph H will be partitioned after n updates, each taking at most Δ time, leading to a total time of $n \cdot \Delta \in$ poly. This contradicts the known NP-hardness [1] of the static hypergraph partitioning problem. □

3 Related Work

Mt-KaHyPar

We develop on our dynamic partitioner on top of *Mt-KaHyPar* [9], a high quality shared memory hypergraph partitioner.

Mt-KaHyPar utilizes the *multilevel* partitioning paradigm [10], [11], to partition the hypergraph.

The multilevel partitioning paradigm consists of three phases:
In the *coarsening* phase, the hypergraph is iteratively reduced to smaller hypergraphs by contracting clusters of nodes, over the course of multiple levels. The *initial partitioning* phase partitions the coarsened hypergraph of the smallest level into k blocks. The *uncoarsening* phase uncoarsens the partition to the original hypergraph level by level. This is done by iteratively *refining* the partition of the coarsened hypergraph to improve the quality of the partition and then applying it to the uncoarsened hypergraph.

When refining the hypergraph Mt-KaHyPar uses a *gain cache* to speed up the computation [9]. The gain cache stores the gain of a vertex for all possible moves. The *gain* of a vertex is the change in the connectivity metric if the vertex is moved to a different block.

The *parallel k-way localised FM* algorithm is a search algorithm that refines a partition by moving vertices to different blocks. It is based on the Fiduccia-Mattheyses (FM) algorithm [12] and is used by Mt-KaHyPar to refine the partition in the multilevel partitioning process [9]. The algorithm starts with a given partition and moves vertices iteratively, prioritizing high gain moves. An important property of the algorithm is that it allows for moves with negative gain to overcome local minima. The algorithm uses a rollback mechanism to revert to the best previous state at the end of the search.

Mt-KaHyPar implements the *unconstrained label propagation* refining strategy. Contrary to constrained label propagation, unconstrained label propagation allows for the temporarly assignment of vertices to blocks that would violate the balance constraint. This enables the search to escape local minima, which is not possible with constrained label propagation.

Unconstrained refinement necessitates the rebalancing of the hypergraph. Mt-KaHyPar uses the *static rebalancer* [13] to rebalance the hypergraph. It pushes vertices out of overfull blocks to achieve a balanced state.

V-cycles are a multilevel method, that can enhance the quality of a partition [9]. V-cycles combine a multilevel approach with the knowledge of an existing partition to refine the partition. A V-cycle coarsens the graph using the same heuristics as a normal multilevel-partitioning but also propagates the existing partition down to the coarsened levels by restricting contractions to the same block of the previous partition. Subsequently, during the process of uncoarsening, it projects the coarsend partition upwards allowing changes to the previous partition of that level only if they are beneficial.

Repartitioning and Dynamic Partitioning

A comprehensive overview of the field of graph repartitioning is given in [14].

There are multiple closly related works to the field of dynamic hypergraph partitioning. Most of it is in the field of dynamic *non-hyper* graph partitioning [15], [16], [17], [18], [19], [20], [21], [22].

Nicoara et al. [15] build a lightweight graph repartitioner for the Hermes addon for neo4j. Their goal is to modify existing partitions with minimal migration cost. They achieve this goal by splitting the migration of a vertex in two phases, moving only lightweight auxilary data in the first. A new vertex is added to a partition based on connectivity data collected during previous changes. They evaluate their algorithm on large social networks.

Huang et al. [16] introduce a dynamic graph partitioning algorithm that allows replication of vertices to improve the quality of the partition. Instead of moving vertices between blocks, they replicate them to multiple blocks. This reduces migration cost and improves partition quality at the cost of increased memory consumption. They evaluate their algorithm on several real world datasets.

Fan et al. [17] incrementalize multiple non-dynamic graph partitioning algorithms to construct dynamic graph partitioning algorithms. They are also able to provide *relative* theoretical bounds on the quality of the partition, showing that the incremental versions retain the same quality bounds as their non-incremental counterparts regarding cut-size and balance. Note that this does not provide any guarantee in relation to the optimal solution. Instead the incremental versions are quality bounded relative to there non-incremental counterparts, meaning, that a high quality static partitioner is transformable into an high quality incremental partitioner. They compare their algorithms to Hermes [15] and ParMetis [23].

Xu et al. [24] use hypergraphs as interim structures to improve non-hypergraph dynamic partitioning, concerning workload balancing. They combine the information of the previous graph partition with historical access logs to hypergraphs, in order to refine the partition. Thus they call their approach LogGP.

Catalyurek et al. [4] introduce a repartitioning, in this case reassigning vertices of the previous partition to different blocks, hypergraph model for dynamic load balancing. They aim to redistribute workload among processors to reduce communication volume while keeping migration cost low at the same time. First they scale the net's costs with the current connectivity and assign a *partition vertex* for each partition. Then they add *migration nets*, between vertices that are moved in the current iteration, with the weight beeing the cost of migration. Finaly they partition the hypergraph with a multilevel partitioner.

There are also multiple works regarding incremental graph partitioning.

Ou and Ranka [22] formulate incremental graph partitioning in terms of linear programming. Their algorithm starts by assigning new vertices to the block closest to the vertex. They then layer the partition to identify border vertices that can be moved to rebalance the partition. After rebalancing, they refine the partition using a linear programming approach to maximise the number of vertices moved without increasing the edge cut.

Lee et al. [18] propose a GPU-based incremental graph partitioning algorithm. They introduce a new data structure for the GPU that can handle changes to the graph without requiring complete rebuilds, by allocating extra space for potential future insertions. They model deletions as moving the vertex to a special *deleted* block. They design a GPU-parallel algorithm that refines and rebalances the partition.

Dai et al. [21] design an online algorithm that responds to incremental vertex degree changes. The algorithm operates in three stages: In the quiet stage, incoming vertices are assigned to blocks based on a simple deterministic hash function. In the reassigning stage, the algorithm reassigns vertices to blocks based on the current connectivity (edge-cut). In the final stage, the algorithm splits vertices with high degrees to multiple blocks by moving the (directed) edges to their source vertices. This allows for a amortization of the loads of accessing high degree vertices in the context of path traversals.

Streaming partitioning is a special case of incremental partitioning, where the changes are only additions of vertices or edges. Streaming partitioners are not able to move vertices between blocks after they have been assigned. Streaming also adds the objective of minimizing the memory usage of the partitioner.

Durbeck and Athanas [19] address the streaming partitioning problem. They introduce a state-based partitioning algorithm that is capable of processing batches of insertions. The approach uses a constant-time streaming heuristic to assign incoming vertices, and rebalances if necessary using a linear-time global spectral method.

Ju et al. [20] introduce a hash-based strategy to partition the stream of incoming changes. They also design a graph computation engine and a workload rebalancer that support incremental algorithms.

Problem	**Hypergraph**	**Change Operations**	**Migration Cost**	**Streaming**
Incrementalization of GP Algorithms [17]	No	Add/Remove edges	Yes	No
Dynamic GP for Hermes [15]	No	Add/Remove vertices/edges	Yes	No
`iG-kway` [18] Incremental GP on GPU	No	Add/Remove vertices Add edges	No	No
Incremental Streaming GP [19]	No	Add vertices	No	Yes
`IGRAPH` [20] Incremental dynamic GP	No	Add vertices	No	Yes
`IOGP` [21] Incremental Online GP	No	Add/Remove vertices/edges	No	No
Parallel Incremental GP [22]	No	Add/Remove vertices/edges	Yes	No
`LogGP` [24]	No	None	Yes	No
Repartitioning HGP Model [4]	Yes	None	Yes	No
`FREIGHT` [25] Fast Streaming HGP	Yes	Add vertices	Yes	Yes
Fully Dynamic HGP	Yes	Add/Remove vertices/edges/pins	No	No

Table 1: Comparison of our Fully Dynamic Hypergraph Partitioning (HGP) problem, to the literature. The columns indicate whether the work partitions hypergraphs, whether it allows for the addition and/or removal of vertices, edges or pins, whether its objective includes migration cost and whether it focusses on streaming partitioning.

In Table 1 we compare our problem with the other works discussed above. We can see, that most of the works are for non-hypergraphs and only a few are able to process the addition and removal of all types of structures. We also see that about half of the works include migration costs in their objective. Our work aims to fill the gap of high quality dynamic hypergraph partitioning algorithms that are able to process all types of changes.

In Section 5.4 we compare our algorithms with the one-pass streaming partitioner `FREIGHT` [25]. `FREIGHT` uses an approximation for the gain computation. This allows for even faster processing of changes. They conclude that their algorithm outperforms all existing (buffered) streaming algorithms in terms of partition quality. Its high partition quality and hypergraph compatibility renders them a suitable subject for comparison.

4 Dynamic Hypergraph Partitioning

In this section, we introduce our dynamic hypergraph partitioning strategies. First, we present the two baseline algorithms for our dynamic hypergraph partitioning problem in Section 4.1. Then we present our main approach to keep the quality of the partition high by using FM to refine the changed hypergraph locally, in Section 4.2. Subsequently, we introduce our incremental rebalancer in Section 4.3 and finally we present how to further improve the partition quality by using v-cycles in Section 4.4.

4.1 Baselines

We introduce two baseline algorithms used to set baselines for quality and running time of the dynamic hypergraph partitioning problem. For our quality baseline, we repartition the hypergraph from scratch after every change. For our speed baseline, we use a greedy insert and remove algorithm and rebalance the hypergraph if necessary.

4.1.1 Quality Baseline

To provide a baseline for quality, we repartition the whole hypergraph after every move using a high quality non-dynamic partitioner. This is very slow (from a dynamic point of view) but ensures a high quality partition. Repartitioning ignores the previous partition of the hypergraph and partitions the hypergraph from scratch.

We use a repartitioning algorithm that implements the multilevel partitioning paradigm to partition the hypergraph. We use the default configuration of Mt-KaHyPar, which is configured to use unconstrained label propagation and unconstrained refinement.

4.1.2 Speed Baseline

To achieve a baseline for running time we use a greedy insert and remove algorithm along with a global rebalancer. This algorithm is very fast but does not guarantee a high quality partition.

We assign each vertex $v \in V_+$ to the block V_i with the maximum number of incident edges $e \in I(v)$, that have pins in V_i, that do not violate the balance constraint.

```
GreedyInsert($v$):
1 block_connectivities := [0; k]
2 for $e \in I(v)$ do
3     for $i \in \Lambda(e)$ do                // for all blocks in the connectivity set of the hyperedge
4         block_connectivities[i] += 1
5     end
6 end
7 $V_f$ := filtered(V, v)                      // only consider blocks with enough space for v
8 $\pi(v)$ := $\arg\max_{\{i | V_i \in V_f\}}$ block_connectivities[i]
```

There may be cases where the vertex cannot be inserted into any block without violating the balance constraint. In this case, the vertex is inserted into the block with the maximum number of incident edges connected to it. This usually happens when $|V|$ is very small and does not occur in our evaluation set of hypergraphs.

Removed vertices are removed from their current block. This can lead to an imbalanced hypergraph which is why we need to rebalance the hypergraph. This is done by utilizing the *static rebalancer* [13]. The implementation of the static rebalancer is not incremental and requires a full pass over the hypergraph. Due to the rarity of rebalancing operations in this algorithm, the static rebalancer proved to be faster than our incremental rebalancer in this case. This is due to the fact that our incremental rebalancer introduces some overhead for staying up to date, which synergizes well with our more advanced algorithms, but is not needed for the greedy algorithm.

4.2 Local FM

Our main approach, to maintain a hight partition quality, is to refine the changes locally.

We achieve this by using a *localised Fiduccia-Mattheyses (FM) algorithm* [9]. The FM algorithm [12] is a search algorithm that refines the partition of a hypergraph by moving individual vertices to different blocks. It decides which vertex to move based on the gain of the move. The search can be performed locally by initially considering only a local set of vertices, the *initial vertex set*, to be selected for moves. When a vertex is moved, the algorithm updates the gain of all affected vertices and adds the affected neighbours of the moved vertex to the set of vertices to be considered. This allows the search to expand beyond the initial vertex set. The search terminates when there are no more vertices in the queue or the stopping rule of Osipov and Sanders [26], is triggered, thus assuming that further gains are unlikely [27] .

We use the added vertices as well as the neighbours of the removed vertices of the current change for the initial vertex set of the FM algorithm.

Local FM requires a correctly initialized *gain cache* to work. This poses a challenge for the incremental processing of changes. The gain cache needs to be updated after every addition or removal of a vertex as well as after every migration of vertices. Naively updating the gain for every vertex of the hypergraph, thus performing a full pass after every move, is not feasible. The workload can be reduced dramatically by only updating values that actualy change.

A first improvement is gained by understanding, that the gain of a vertex move is solely determined by the hyperedges it is a pin of. Therefore, only vertices that are pins of hyperedges that have changed need to be updated.

Another insight is that the connectivity of a hyperedge is not affected by the number of vertices in a block, but only by the number of blocks connected by the hyperedge. The gain of a vertex only changes if its potential to alter this connectivity changes. This is only the case for vertices that are added or removed from blocks $V_\triangle$ with very few other pins of the hyperedge. By checking this limited set of cases we can reduce the workload significantly.

For an edge incident to a *removed* vertex $e \in \{I(v) \mid v \in V_-\}$ we update pins in the following cases:

1. If there are two pins in the block $V_\triangle$ *before* removing v:
   The gain for moving the other pin, to other blocks with pins of e, increases.
2. If there is one pin in $V_\triangle$ *before* removing v:
   The gain for moving other pins of e to the block of the removed vertex of decreases.

For an edge incident to an *added vertex* $e \in \{I(v) \mid v \in V_-\}$ we update pins in the following cases:

1. if there are two pins in $V_\triangle$, *after assigning* $\pi(v) = V_\triangle$:
   The gain for moving the other pin, to other blocks with pins of e, decreases.
2. If there is one pin in $V_\triangle$:
   The gain for moving other pins of e to the block of the removed vertex of increases.

```
GAIN CACHE UPDATE():
for $v \in V_{-}$:
    for $e \in I(v)$ do
        if $|V_e \cap V_{\{\pi(v)\}}| = 2$:
            for $u \in V_e \cap V_{\{\pi(v)\}}$ do        // prior to removal
                updateGainCache(u)
            end
        else if $|V_e \cap V_{\{\pi(v)\}}| = 1$:
            for $u \in V_e$ do
                updateGainCache(u)
            end
        end
    end
end
assignAndRemoveVertices()
for $v \in V_{+}$:
    if $|V_e \cap V_{\{\pi(v)\}}| = 2$:
        for $u \in V_e \cap V_{\{\pi(v)\}}$ do        // after addition
            updateGainCache(u)
        end
    if $|V_e \cap V_{\{\pi(v)\}}| = 1$:
        for $u \in V_e$ do
            updateGainCache(u)
        end
    end
end
```

With this update strategy, we are again able to significantly reduce the workload without sacrificing any quality. In Section 5.3.2 we compare this stage of the algorithm, to later stages. For example, in Figure 2, this stage is depicted as *LocalFM*.

Small Block Threshold

Another important aspect for running time and quality is the selection of vertices to be refined by the local FM search. The selection is crucial for the quality of the partition as it determines which moves can be considered by the local FM search and thus impacts the running time of the algorithm. The goal is to select as few vertices as possible while still ensuring a high quality partition.

For the removed vertices, we select all neighbours of the removed vertices that are in blocks with few other vertices of the same hyperedge. This takes into account that we have to clear a complete block of vertices of a hyperedge in order to achieve a quality improvement, this is unlikely to happen for a large group of vertices. We introduce a parameter called *small_block_threshold* to determine how many vertices are considered few enough to be included in the local FM search. We later show in Section 5.3.3 that a threshold of 5 is a good default value.

For added vertices that open up a new block for a hyperedge, we select all neighbours of the added vertex from that hyperedge in addition to the vertex itself, since there is a chance for all of those neighbours to be moved to the new block to reduce the connectivity for other hyperedges.

```
LOCAL FM VERTEX SELECTION($small\_block\_threshold\ \tau$):
for $v \in V_{-}$:                                    // for removed vertices
    for $e \in I(v)$ do
        select($\{u \in V_e \mid |V_{\{\pi(u)\}} \cap V_e| < \tau\}$)   // neighbours of v in blocks with pincount < $\tau$
    end
end
for $v \in V_{+}$:                                    // for added vertices
    select(v)                                         // select the vertex itself
    for $e \in I(v)$ do
        if $|V_e \cap V_{\{\pi(v)\}}| = 0$:           // if a previously empty block is targeted
            select($V_e$)                             // select all vertices of the edge
        end
    end
end
```

If the change only involves vertex removals, the set of selected vertices may be empty. In this case the local FM refinement is skipped completely.

Additionally, the refinement requires the current connectivity value of the partition as well as the current imbalance. The imbalance is computed from scratch before the refinement. The connectivity value, on the other hand, is far to costly to compute from scratch every time. Instead, the connectivity value is computed incrementally. This is done by storing the connectivity value of the previous state of the hypergraph and updating it after every move. This requires some components, such as the rebalancer, to update the connectivity value after moving vertices.

Using this threshold we are able to achieve a fast refinement of the hypergraph with little to no loss of quality. We will later evaluate this stage of the algorithm as *small-blocks* in Section 5.3.3, for example in Figure 1.

So far we have focused on improving the running time of local FM without losing quality. Our further improvements focus on sacrificing as little running time as possible to improve quality.

4.3 Incremental Rebalancer

Rebalancing the hypergraph is necessary when parts of the algorithm perform moves that violate the balance constraint. The rebalancer moves vertices between blocks to reduce the imbalance. The static rebalancer [13] is a non-incremental rebalancer, that selects moves based on their gain, to restore the balance constraint. The static rebalancer requires for global passes over the hypergraph on every call. Re-designing it can not only lead to a speedup when combined with advanced refining techniques but also its structures can also be used to find quality-improving moves.

The typical use case of a rebalancer is to push vertices with high weight from overfull blocks to blocks with low weight, while priorising moves with high or at least not to negative gains. We attempt to use an inverted version of this method to also fill blocks with positive gain moves to achieve global improvements.

Our incremental rebalancer consists of two priority queues for each block. One for vertices that are to be moved into the block (*pull*) and one for vertices that are to be moved out of the block (*push*). Each vertex is added to the push queue of the block containing it, as well as to the pull queue of every other block.

The priority of the vertices is determined by the gain of the move and the weight of the vertex. Pushing a large vertex is more advantageous than pushing a small vertex. On the other, hand pulling a small vertex is more advantageous than pulling a large vertex. The opposite is true when the gain drops below zero.

$$\forall v \in V_i \ \text{push\_prio}(v) := \begin{cases} \max\{\text{gain}(v, V_i, V_j) \mid V_j \in \Pi\} \cdot w(v) & \mid \text{if} \max\{\text{gain}(v, V_i, V_j) \mid V_j \in \Pi\} \geq 0 \\ \frac{\max\{\text{gain}(v, V_i, V_j) \mid V_j \in \Pi\}}{w(v)} & \mid \text{if} \max\{\text{gain}(v, V_i, V_j) \mid V_j \in \Pi\} < 0 \end{cases}$$

$$\forall v \notin V_i \ \text{pull\_prio}(v) \ := \begin{cases} \frac{\text{gain}(v, V_j, V_i)}{w(v)} & \mid \text{if gain}(v, V_j, V_i) \geq 0 \\ \text{gain}(v, V_j, V_i) \cdot w(v) & \mid \text{if gain}(v, V_j, V_i) < 0 \end{cases}$$

The push priority queues can use the fact that each vertex is only in one push queue at a time to share parts of their datastructure for memory efficiancy.

The gain cache, as well as the push- and pull-queues, must be updated after every move. This is done by inserting the moved vertex into the push priority queue of the target block's and updating it in every other pull priority queue. Additionally, all vertices that are affected by the move are updated in their pull and push priority queues. To decide which vertices are affected we use the filter described in Section 4.2.

Pulling into Partition

Once per change, we try to pull beneficial vertices into each block. This is done prior to the usual rebalancing, after the change has been processed. This allows for some moves to be performed, that would otherwise be restricted after rebalancing

When pulling into a block, we pull the vertex with the highest gain from our block's pull queue. We then check if the move is valid and whether the move would cause a block overload. If the vertex is to heavy we buffer it and reinsert it after we finished pulling into our block. If the move is valid, we execute the move, update the gain cache and the heaps. We continue this process until we reach a vertex with a negative gain or the pull queue is empty.

```
PULLINTOPARTITION($target\_partID$):
while not empty blocks[target_partID].pull do
    let $u$, gain:= blocks[target_partID].pull.top()
    if gain $< 0$:
        break
    if move violates balance:
        bufferAndReinsertAfterwards(u, gain)
        continue
    let move := (partID(u), target_partID, u, gain)
    ExecuteMove(move)
    updateGainCacheForMove(move)
    updateHeapsForMove(move)
end
```

Pushing from Partition

If the partition becomes imbalanced it must be rebalanced. This is done by using the push queues. We start by popping the vertex with the highest gain from the push priority queue of our block. If the move is valid, we execute the move, update the gain cache and the heaps. We continue until all blocks are balanced.

```
REBALANCE():
while imbalanced do
    let source_partID := imbalanced_partition()
    let $u$, gain:= blocks[source_partID].push.top()
    let target_partID := getBestTargetPartID(u, source_partID)
    if move violates balance:
        bufferAndReinsertAfterwards(u, gain)
        continue
    let move := (source_partID, target_partID, u, gain)
    ExecuteMove(move)
    updateGainCacheForMove(move)
    updateHeapsForMove(move)
end
```

In section Section 5.3.4 we evaluate the incremental rebalancer. Due to the fact, that we introduce a new step in the algorithm, by using the pull queues, we are able to achieve a higher quality partition. The additional overhead of the pull queues is countered by the speedup gained by incrementalizing the rebalancer. In summary, we gain a higher quality partition while sacrificing little to no running time.

4.4 V-Cycles

V-cycles are multilevel methods that are used to gain a high quality refinement of the partition. Using a v-cycle is significantly slower than the other components we introduced but can provide a high quality partition in the iteration it is used. We use it to improve the partition in intervals. V-cycles use information from the previous partition to improve the coarsening heuristic by restricting contractions to vertices in the same block.

We provide a parameter *vcycle_stepsize* to specify the interval at which the v-cycle is used. Since changes can vary in size and impact, the interval is not determined by the number of changes, but by the weight difference since the last v-cycle. The weight difference is the sum of the weights of the added and removed vertices.

$$\forall \Delta G : \text{weight\_diff}(\Delta G) \coloneqq \sum_{v \in V_+} w(v) + \sum_{v \in V_-} w(v)$$

The parameter is the fraction of the weight difference to the total weight of the hypergraph. Setting it to 0.1 means that the v-cycle is used if the summed weight difference of the processed changes is more than 10% of the weight of the hypergraph since the last v-cycle. Removing vertices from the hypergraph counts to the sum in the same way as adding vertices.

After using the v-cycle, the gain cache and our incremental rebalancer are updated, to reflect the new partition.

In Section 5.3.5 we evaluate the v-cycle parameter. We show that a v-cycle step size 10% is a good default value, but other values are viable as well.

5 Evaluation

5.1 Methodology

We evaluate our algorithms on a set of 100 representative hypergraphs derived from a benchmark set of Heuer and Schlag [28], which has already been shown to be useful for evaluating hypergraph partitioning in another paper [29]. The subset contains instances from several benchmark sets, such as the ISPD98 VLSI circuit benchmarks [30], the DAC 2012 routability-driven competition [31], and the 2014 international SAT competition [32]. A detailed overview of the hypergraphs in the set can be found in Figure 7 and in Table 2. The geometric mean number of vertices is 116 376.4. The largest hypergraph in the set has 1 416 850 vertices and 331 196 hyperedges. We generate changes for those hypergraphs using our change generator to generate the two change-lists described in Section 5.2.2.

We implement the proposed algorithms within the Mt-KaHyPar framework (v1.4). In the evaluation we typically use the default configuration, unless otherwise stated. Its components have been used and discussed in previous work [9], [13].
For the parameters of the partitioner we opt for the following values:

- We choose the de facto standard in the literature for the *imbalance constraint*, which is set to $\varepsilon = 0.03$. This allows blocks to be up to 3% larger than the average block weight.
- In order to reduce the execution time of our experiments, we primarily focus on runs with k = 4 unless otherwise stated. However, as we will demonstrate in Section 5.3.6, greater values for k yield similar results.

Recall that our goal is to process changes to a hypergraph in a way that optimises connectivity and running time. A change is the removal or addition of a vertices, hyperedges or pins. A change list is a sequence of changes. In Section 5.2.2 we describe two change list types we use to evaluate our algorithms. We compare the different algorithms by plotting their connectivity and running time for both change list types aggregated via gmean for the 100 hypergraphs. The x-axis typically is the percentage of changes that have been processed, while the y-axis is the connectivity or running time. The quality is evaluated after every change while the running time is accumulated over all changes up to the current change.

Our change lists types do not start with an empty hypergraph but with a hypergraph that has been partitioned using the default configuration of Mt-KaHyPar. At 0% of the changes, the hypergraph is in its initial partitioned state. This is why the connectivity does not start at 0. Instead, all strategies start at the same connectivity value determined by the innitial high quality partitioning. This is part of the setup phase and is therefore not measured in the running time.

All experiments are run on an Intel(R) Xeon(R) Gold 6314U CPU (Ice Lake-SP) with 32 cores with 2 threads each. The CPU is clocked at 2.30 GHz (3.4 GHz Turbo) with 512 GB main memory, 32x1.5 MiB L1d-, 32x1 MiB L1i-, 32x40 MiB L2- and 48 MiB L3-Cache. The node is running Rocky Linux 9.5 (Blue Onyx). The measured code was compiled using gcc version 14.2.0. and the flags `-O3 -mtune=native -march=native`.

5.2 Change Lists

Recall that a change is the removal or addition of vertices, hyperedges or pins.

Change lists are used to describe multiple dynamic changes to a graph in a single file. We introduce our format for change lists and show how we generate them. A change list is generated based on a given hypergraph, allowing us to generate different kinds of dynamic behaviour for existing hypergraphs.

5.2.1 Change Generation

Due to the lack of real-world dynamic hypergraphs, we generate our change lists based on a static hypergraph. We use a random generator in order to generate diverse change lists.

We remove vertices one-by-one in a random order. Removing a vertex also removes the it from all hyperedges it is a pin of. If this leads to an edge being connected to less than 2 vertices the edge is removed as well.

Similar to the removal of vertices, we add vertices from the list of removed vertices. If the addition of a vertex leads to an edge being connected to 2 vertices, the edge is added as well.

Randomly removing and adding vertices can result in vertices that are disconnected from the existing graph (*degree-0-vertices*). This is unrealistic behaviour as we would expect the new vertices to be connected to the existing hypergraph somehow. More importantly, they reduce the complexity of the partitioning task. The unconnected vertices basically act as jokers for the partitioner, and can fix the balance constraint without increasing the connectivity. In particular, change lists that do not reach the final size of a hypergraph, but rather maintain a lower size level, are affected by this, since vertices that would later become connected are removed.

We therefore postpone vertices selected for addition or removal if they lead to degree-0-vertices. There are cases where this is not possible, for example when the hypergraph features unconnected components. In that case we use one of the postponed vertices.

Change List Format

The change list format is a plain text file structure used to represent a sequence of modifications to a hypergraph. The first line indicates the number of changes, followed by multiple entries, each consisting of six lines describing the changes:

1. **Number of Changes** The first line specifies the total number of changes
2. Each change consists of six lines in the following order:
   - **Added Vertices** A space-separated list of IDs for vertices added to the hypergraph:
   - **Added Edges** A space-separated list of IDs for added edges:
   - **Added Pins** Pairs of hypervertex and hyperedge IDs for added pins:
   - **Removed Vertices** A space-separated list of IDs for removed vertices:
   - **Removed Edges** A space-separated list of IDs for removed edges:
   - **Removed Pins** Pairs of hypervertex and hyperedge IDs for removed pins:

5.2.2 Change Lists used for Evaluation

Using the change generation methods described in Section 5.2.1, we can now combine them to create change lists. We use two different types of change lists to evaluate our algorithms. The first type is a simple change list that removes a set of vertices and then adds them back. The second type combines both operations into batch changes. Both also include the removal and addition of the corresponding hyperedges and pins.

Remove then Add

This change list type removes a percentage of vertices from the hypergraph, as described in Section 5.2, and then adds the same percentage of vertices back to the hypergraph. In our evaluation, we choose the percentage of vertices to be removed and added 25% of the total number of vertices. Each change only affects a single vertex.

Expectedly this results in a v-shaped graph for the connectivity metric, since the size of the hypergraph is reduced by 25% and then increased again by 25%.

Mixed

This change list type starts with 25% of the vertices already removed.
The actual changes are n batch changes with 50 vertices changed per batch. n being 5% of the original number of vertices.
The distribution of added and removed vertices per batch is randomly chosen at a chance of 50/50.

Expectedly this results in an almost horzontal line for the connectivity since the number of vertices remains about the same.

Comparison

Due to the batch nature of the mixed change list type, the two change lists scale differently.
The mixed change list modifies $|V| \cdot 0.05 \cdot 50$ vertices in $|V| \cdot 0.05$ changes.
The remove then add change list modifies $|V| \cdot 0.5$ vertices in $|V| \cdot 0.5$ changes.
This means that the mixed change lists modifies 5 times more vertices in 10 times less changes.
This should be kept in mind when comparing the two change list types, since strategies that scale well with the number of changes may not scale well with the number of vertices changed and vice versa.

5.3 Strategy Evaluation

5.3.1 Baselines and Metrics

The **baseline for quality**, called *repartition*, is to repartition the hypergraph from scratch each time a change is processed. This ensures a high quality partition, but is very slow. Since repartitioning is done from scratch, it is independent of the previous partitioning. We exploit this by calculating only the first value of each 1% bucket of changes for our evaluation, measuring the time for 100 changes. This is done to reduce the running time of the benchmark, which would otherwise take approximately 64.31 hours for the geometric mean of the rta change list. To compare the running time with the other strategies, we divide the running time of the *repartition* baseline by 100 and multiply it by the number of changes.

In Figure 1 we can see a large gap between the running times of the two change lists. The rta change list is more than 7 times slower than the mixed change list. This is due to the fact that the running time of this strategy scales drastically with the number of changes. The rta change list has 10 times more changes than the mixed change list.

The **baseline for running time**, called *greedy*, is the greedy insertion and removal of vertices. This is very fast (0.52 seconds on average for the mixed change list, as seen in Figure 1), but does not ensure a high quality partition. It also requires a rebalancer to restore the imbalance in case the partition is imbalanced due to the removal of vertices. It utilizes the *static rebalancer* for this purpose. In contrast to the *repartition* baseline, the *greedy* baseline is only about 1.6x slower for the rta change list than for the mixed change list. This is due to the fact that the *greedy* baselines run time scales not only with the number of changes, but also with the number of vertices changed. The mixed change list changes 5 times more vertices than the rta change list.

Comparing the quality of our baselines we see large gaps between our baselines both regarding quality as well as running time. Specifically, the *greedy* baseline is more than 5 orders of magnitude faster, but achieves approximately 5x poorer quality than the *repartition* baseline, for the mixed change list. These strategies are both not viable for practical usage. The *repartition* baseline is far too slow to be used in practice, while the *greedy* baseline does not provide a good enough quality partition.

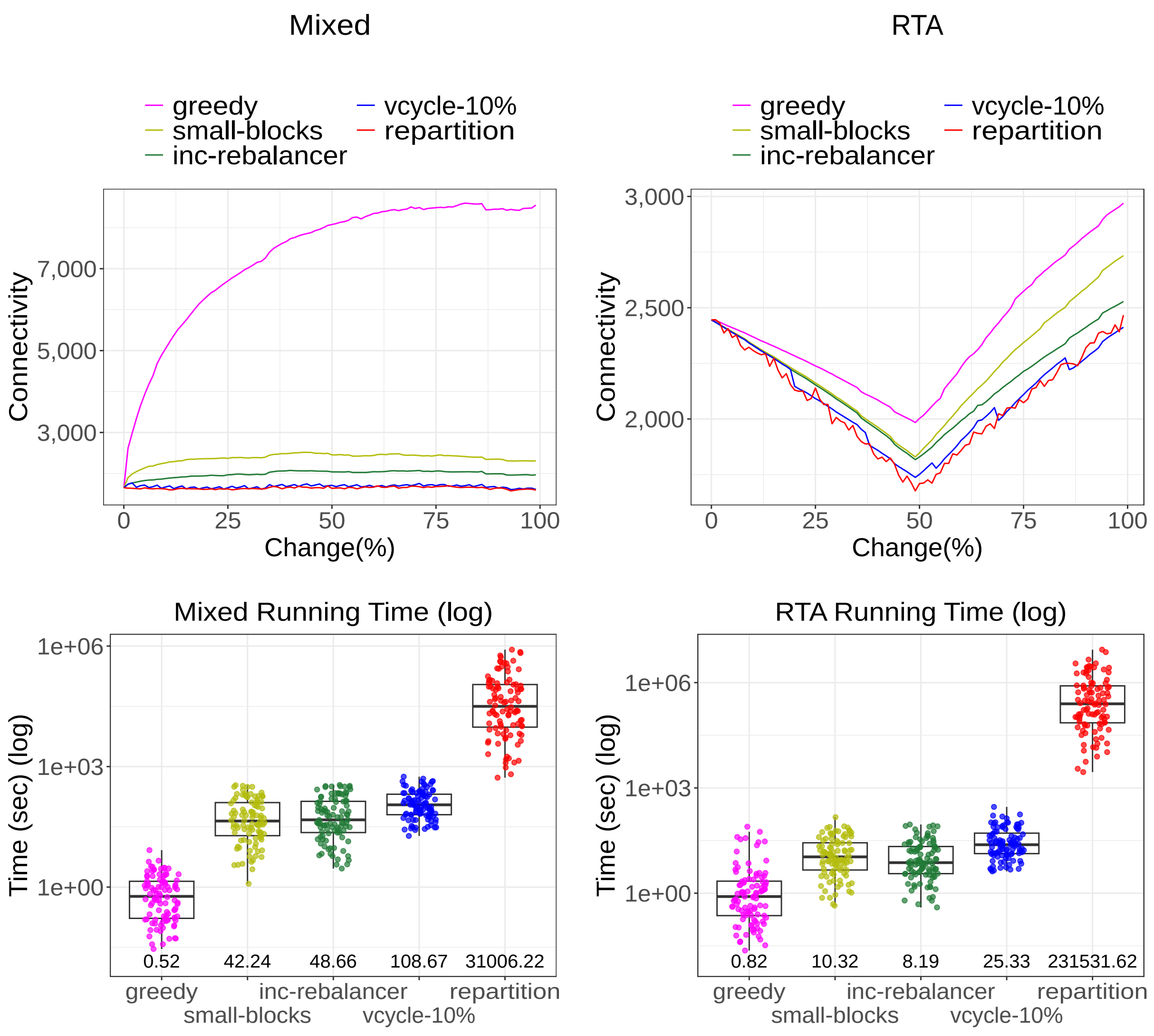

Figure 1: Quality and speed for the `mixed` and `remove-then-add` change lists. The annotations above the x-axis, in the running time plots indicate the geometric mean of the running time for the respective strategy across all 100 hypergraphs. The plots on the left show the connectivity and running time for the mixed change list, while the plots on the right show the connectivity and running time for the remove-then-add change list.

This leaves a lot of room for improvement for our advanced dynamic strategies. In particular, we expect our strategies to be several orders of magnitude faster than the *repartition* baseline while maintaining comparable quality.

5.3.2 Local FM

Recall that the core idea of our dynamic strategy is to use local FM refinement on the hypergraph to refine our greedy insertions and removals. We refine all neighbours of all changed vertices using the Mt-KaHyPar implementation of kway localised FM. Contrary to Mt-KaHyPars default settings, we use constrained refinement and only one round of FM. These changes lead to a reasonable speedup without noticably sacrificing quality. This is largely due to the fact that we are calling the FM algorithm very frequently (after every change) compared to a static partitioning.

In order to trigger the refinement, we must provide a valid gain cache. Instead of updating the entire gain cache in every iteration we design an incremental gain cache as described in Section 4.2. We implement the incremental gain cache with the reduced gain update filter described in Section 4.2.

In Figure 2 this version of our local FM strategy *localFM* is depicted. It already achieves a significant speedup over the *repartition* baseline, beeing about 426x slower than the *greedy* baseline for the mixed and 42x slower for the remove then add change list. On the other hand it achieves connectivity values only 1.5x higher than the *repartition* baseline for the mixed change list.

5.3.3 Small Blocks Threshold

A final running time improvement for localFM is achieved by introducing a small block threshold. This threshold is used to determine whether the number of vertices of a hyperedge in a block is considered small enough to be likely to be moved. This is used to reduce the number of vertices that are selected for refinement, improving the running time for a small quality tradeoff.

A small-block-threshold of 5 means that only hyperedges with 5 or less vertices in the block are selected for refinement. The threshold significantly reduces the number of vertices that are selected for refinement which improves the running time for a minor quality trade-off.

We can see in Figure 2 that the small block threshold provides a good trade-off between running time and quality. The threshold 5 provides an about 5x speedup without impacting the solution quality significantly. Setting it to 1 would provide us with another 3x speedup but, as can be seen for the mixed change list in Figure 2, it would increase the connectivity noticably, thereby reducing the quality, which is why we opt for the threshold 5 for our further evaluation. This value can be set freely for fine tuning.

5.3.4 Incremental Rebalancer

We now achieve feasible running times for even large amounts of changes, beeing only about 12 - 100 times slower than our `greedy` baseline, depending on the change list composition (Figure 1). We now aim to improve the quality of our partitioner. We do this by introducing the incremental rebalancer to our strategy. As we described in Section 4.3, we use the incremental rebalancer for two purposes. First, we use it to pull vertices that improve the quality of the partition into blocks that are not overloaded. This is done even if the partition is currently imbalanced. The second use is to push vertices from blocks that are overloaded to blocks that are not overloaded. This is done until the partition is balanced. We add the incremental rebalancer to our small-blocks-strategy with a small block threshold of 5.

We compare this approach, including the incremental rebalancer, to our baselines and to the prior version of our strategy in Figure 1. We can see that it is equaly fast as our previous version, with some fluctuation depending on the changes provided. The quality of the partition on the other hand is improved by about 15% for the mixed change list and about 9% for the remove then add change list. Notably, the improvement for the remove then add change list is concentrated in the second half of the change list. This demonstrates the impact on additive changes.

We expect the balancing part of the incremental rebalancer to be faster than the previous static rebalancer, since we no longer have to do a global pass over the hypergraph anymore. We can see this effect for the `rta` change list in Figure 1. On the other hand we can see that for the `mixed` change list, that we are slightly slower than the previous version. This is due to the second part of the new rebalancer, which adds an additional step to the algorithm. Whether this results in an overall speedup or slowdown depends on the change list composition, but remains within reasonable limits.

The evaluation of the incremental rebalancer shows that it is able to improve the quality of the partition noticably without sacrificing too much running time.

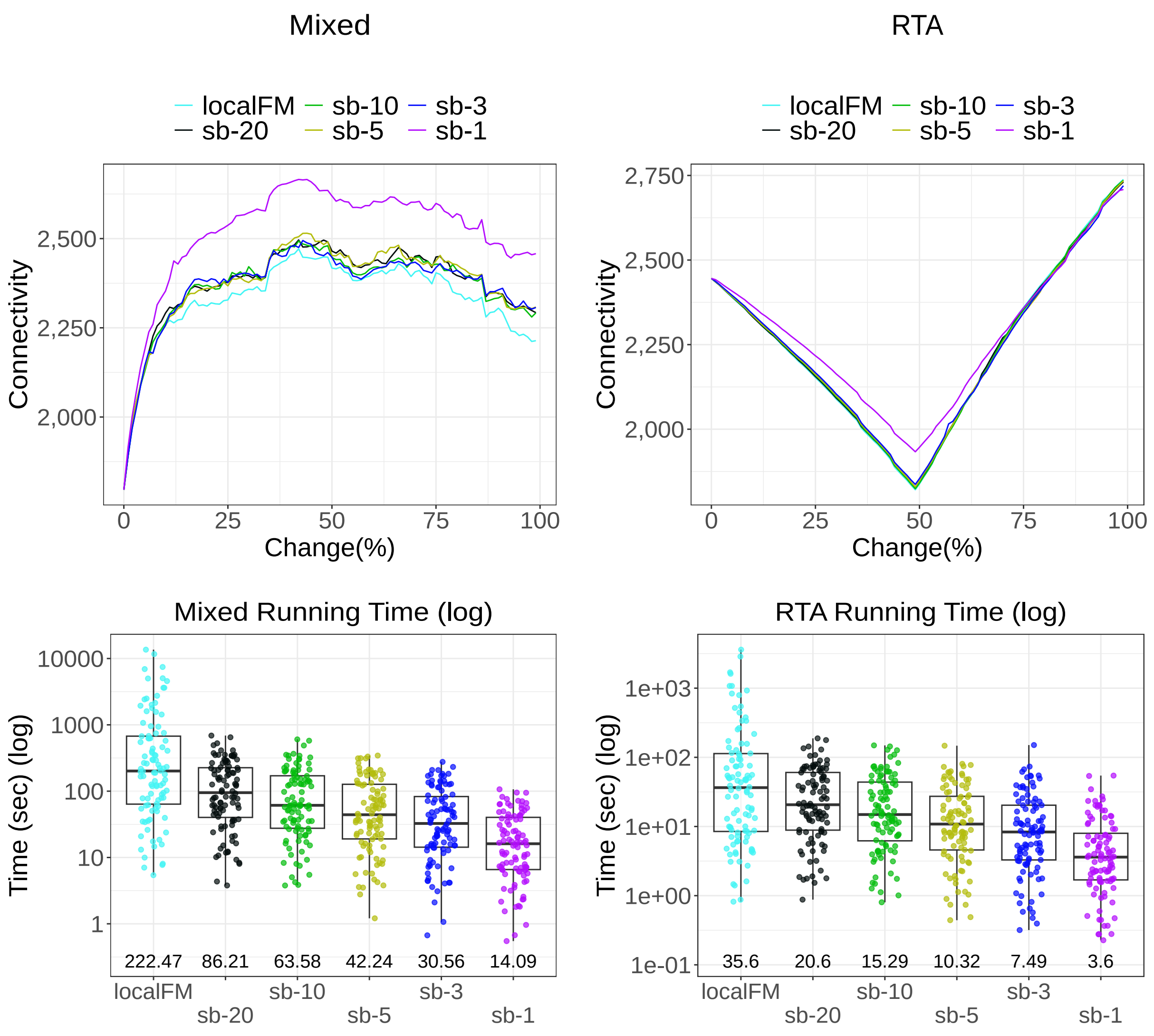

Figure 2: Quality and Speed of the small block threshold strategy for some thresholds for the mixed and remove then add change lists. SB-10 denoting a small block threshold of 10, i.e., the number of vertices in a hyperedge in a block must be 10 or less to be considered for local refinement. LocalFM, not implementing this threshold, could be seen as SB-∞

5.3.5 V-Cycle

Our strategy, including the incremental rebalancer, is already several orders of magnitude faster than repartitioning, while closing in on its quality. We can further improve the quality of the partition by performing a v-cycle in Mt-KaHyPar. We use the v-cycle implemented in Mt-KaHyPar [9] via the improvement interface. After the v-cycle, we update the gain cache and the incremental rebalancer to reflect the new partition.

Using a v-cycle allows us to improve the quality of our partition drastically. We can see in Figure 1 that every time we use a v-cycle we reach the quality baseline of the repartitioning strategy. Afterwards we start to deviate from the quality baseline. We therefore suggest to use the v-cycle in intervals.

We introduce a parameter (*vcycle_stepsize*) that determines how often the v-cycle is triggered. Setting it to 5% means that we trigger a v-cycle after the sum of the weight of the vertices that have been changed since the last v-cycle is more then 5% of the weight of the hypergraph vertices at the time of the last change. The smaller the step-size the more often the v-cycle is triggered. We demonstrate the trade-of between quality and running time by evaluating different values for the parameter in Figure 3.

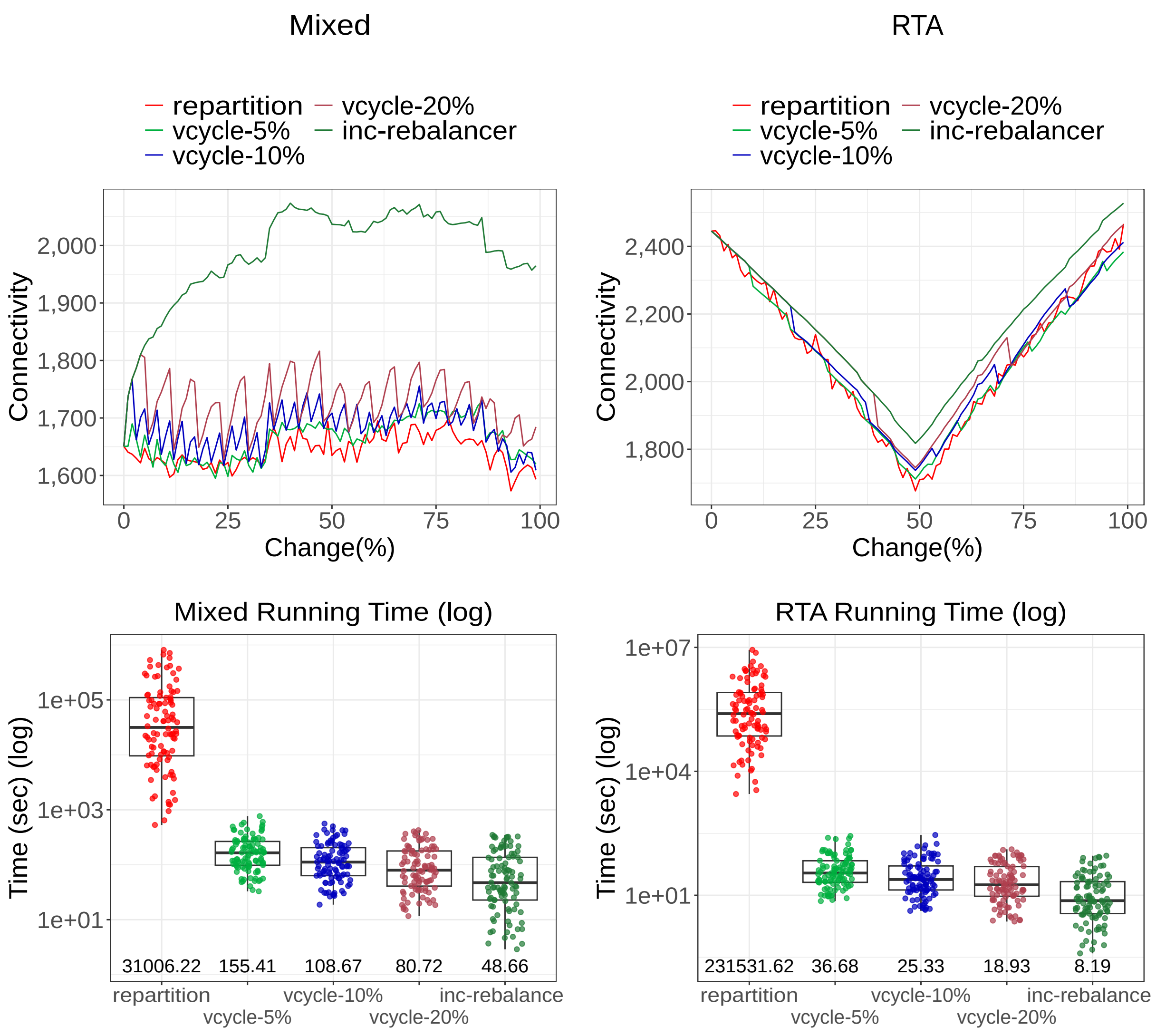

Figure 3: Quality and Speed of the v-cycle strategy for different parameters for the mixed and remove then add change lists.

We can see that both the factor of improvement as well as the running time is heavily impacted by the step-size parameter. There is no clear superior step-size parameter. The 10% step-size parameter is a good example for a high quality setting with reasonable running time. But depending on the use-case the 5% or 20% step-size parameter might be more suitable.

We choose a step-size of 10% for our comparison to the previous versions and the baselines in Figure 1. Here we can see that for the mixed change list, for example, the running time is about 2.2x slower for a step-size of 10% compared to no v-cycles. The quality on the other hand is improved by about 18%. More noticably the connectivity is 23x closer to the quality baseline at the end of the change list beeing only about 1% worse than the quality baseline. This shows that by utilizing v-cycles we are able to improve the quality of our partition drastically, reaching almost optimal quality, while still beeing more than two orders of magnitude faster than the *repartition* baseline.

A single v-cycle is not inherently faster than repartitioning, for example for the `remove-then-add` change list in Figure 1 we can see that the v-cycle is used about 5 times throughout the change list, whereas the `repartitioning` strategy repartitions the hypergraph once every change, averaging at about 50.000 when considering the average hypergraph size from Section 5.2.2. The `v-cycle` strategy

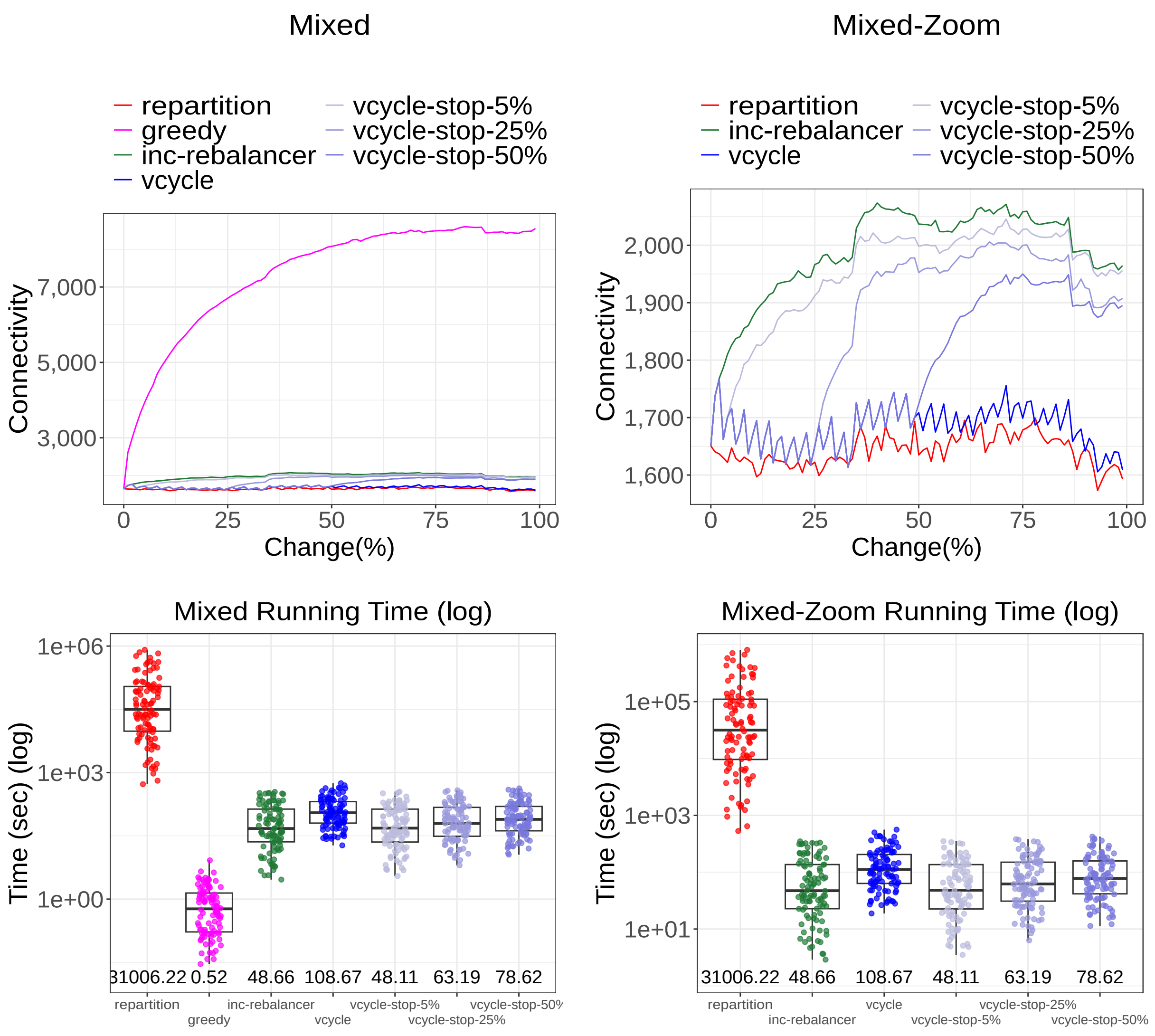

Figure 4: Comparing v-cycle strategy variations that stop using the vcycle after a certain percentage of changes. Quality and speed for the mixed change list for k = 4.
The plots on the right do not feature the greedy baseline to zoom in on the other strategies.
The v-cycle step-size is set to 10. The small block threshold is set to 5.

is about 4 orders of magnitude faster than the `repartitioning` strategy. This matches the expected speedup of the v-cycle strategy, stemming from the fact that it used rarely.

Long Term Effects of the V-Cycles

The v-cycle improves the quality of the partition immediately and very drastically. An interesting question is what happens if we stop using v-cycles at some point during the change list. We can see in Figure 4 that the connectivity value of the partition quickly rises after stopping the v-cycle, in a very similar pace as in the beginning of the change list, but plateaus at a lower level than the non-vcycle strategy. Over the course of more changes it then reaches a similar level as the non-vcycle strategy.

For example, if we stop using the v-cycles after 5% of the changes, we reach our high plateau after 25%, but stay at an offset to the non-vcycle strategy until about 85% of the changes have been applied.

This shows that using the v-cycle is not only beneficial for the first few changes, but also in the long term.

5.3.6 Scaling k

Since we have, up to now, evaluated our strategies only for $k = 4$ (the number of blocks), we now evaluate our strategies for $k \in \{16, 32, 64, 128\}$ in Figure 6. We use the same change lists as before. The v-cycle step-size parameter is set to 10. The small block threshold is set to 5.

We can see that both running time and connectivity increase with increasing k, but the relative quality and running time stays roughly the same for the different change lists respectively.

The relative increases in running time for the jump from $k = 16$ to $k = 128$ are about (3.1, 1.4, 2.9, 2.0) for the `mixed` and (3.3, 3.2, 4.6, 3.3) for the `rta` change list, for the strategies (`repartition`, `greedy`, `inc rebalancer`, `v-cycle`). The `greedy` baseline scales better than the `repartition` baseline, with our advanced `v-cycle` strategy scaling something in between.

More importantly, we can see that the quality of the partition of our advanced strategies remains very close to the quality of the `repartition` baseline, regardless of the value of k, compared to the quality of the `greedy` baseline. Additionally, the running time of our high quality `v-cycle` strategie remains several orders of magnitude faster than the `repartition` baseline, while scaling equally or better.

5.4 Comparing to other Partitioners

To the best of our knowledge, there are no fully dynamic hypergraph partitioners that we could compare against. Hence, we compare our strategies to the state-of-the-art streaming hypergraph partitioner FREIGHT [25].

Recall that streaming partitioning is a special case of dynamic partitioning, where the hypergraph is only changed by adding vertices, edges and pins. Streaming usually does not allow migration of vertices after they have been added. Since we do not abide by this restriction, we expect our approach to be slower but of higher quality. Additionally streaming algorithms try to minimize their memory usage. Our strategies, on the other hand, use a significantly larger amount of memory. For example, our incremental rebalancers memory usage scales in $\Omega(n \cdot (k + 1))$.

We compare our strategies to the FREIGHT partitioner using the andrews.mtx.netl streaming hypergraph provided by FREIGHT. It has 60 000 vertices and hyperedges. We translate the netl format into our change list format and run our strategies on the change list. We compare the connectivity and running time of our strategies to the connectivity and running time of FREIGHT in Figure 5.

We also compare our strategies to FREIGHT using a realworld timed bipartite amazon network. It connects users with the products they rated. We interprete the users as vertices and the products as hyperedges. We ignore all products that were rated by only one user, e.g. single pin hyperedges. The timed ratings, in our case pins, are used to determine the streaming order of the vertices. The transformed hypergraph has 2 146 058 vertices and 608 681 multi-pin hyperedges.

As expected, all our strategies are significantly slower than FREIGHT, but yield partitions of much higher quality. For example, for the andrews graph, for k beeing 64 our strategy without v-cycles is about 100x slower, but yields a partition with 13x less connectivity in Figure 5. This is not only due to the fact that we provide additional capabilities, but also FREIGHT only computes an approximate gain by only comparing blocks where vertices have been added recently. This difference is best seen when comparing FREIGHT to our `greedy` baseline, since our `greedy` baseline is essentially a streaming algorithm that does not migrate a vertex after it has been assigned when confronted exclusivly with adding operations. Due to FREIGHT's approximate gain, it is faster than our `greedy` baseline (about 3x for k = 64), but yields a partition with a higher connectivity (about 4x for k = 64). This should be kept in mind when comparing the algorithms, since it demonstrates how FREIGHT prioritizes running time in this trade-off.

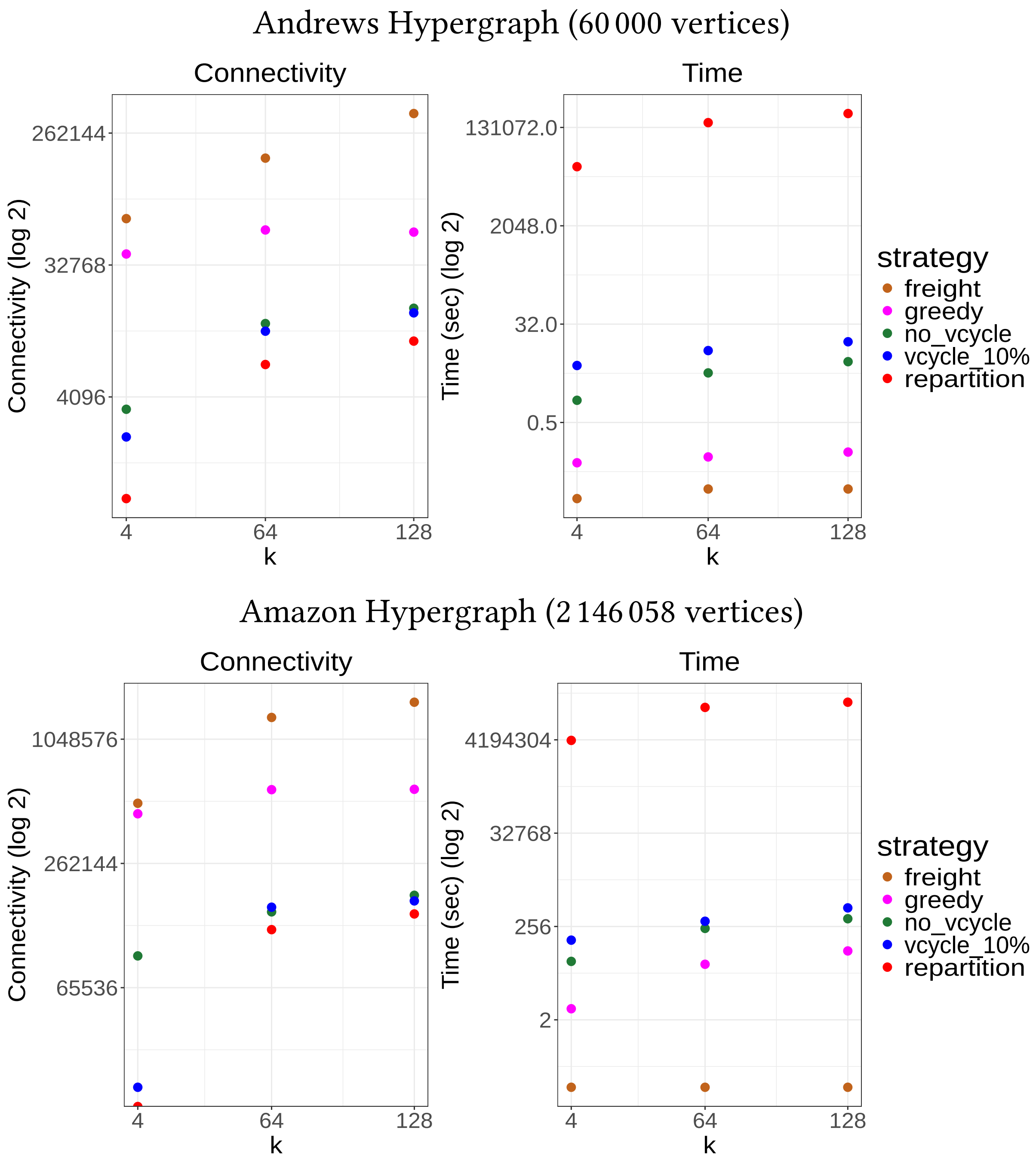

Figure 5: Connectivity metric and timings comparing FREIGHT [25] (streaming algorithm) to some of our strategy variants (no_vcycles, vcycles after every 10% weight change) and our baselines.

Another interesting insight is that adding v-cycles increases the running time without gaining much in quality. This is due to the fact that our strategies are optimized for large existing partitions, since our v-cycles are triggered after relative weight changes. When starting from 0 vertices, as in the evaluated case, the v-cycle is triggered very often during the first few changes. This is not the case for our typical change lists, since they start with a large number of vertices.

The conclusion of this comparison is that our strategies effectivly utilizes the fact that they are allowed to perform migrations of vertices after they have been added in order to achieve significantly better quality partitions. This comes at a considerable cost of running time, but is still feasible for state of the art hypergraphs and reasonable k.

6 Conclusion

We have developed and implemented a strategy to process dynamic changes to a hypergraph that provides a high quality partition while being more than two orders of magnitude faster than naively repartitioning from scratch. We have evaluated the strategy and the important changes that lead to the final strategy by generating two types of change lists for an existing set of 100 hypergraphs. We set baselines for the running time and quality of a dynamic hypergraph partitioning strategy and compared our strategy to these baselines. We also provided advanced tuning options for the strategy to provide a use case fitting trade-of between quality and running time. We evaluated and provided reasonable defaults for these tuning options.

6.1 Future Work

We have shown that it is possible to process fully dynamic changes to a hypergraph in a fast and high quality manner. However, this opens up many questions and opportunities for future work. Starting with the problem itself, we have yet to provide real-world fully dynamic change lists for hypergraphs to compare future work against. Also, there are multiple facets of the problem that are yet to be explored. For example, adding the minimization of the migration cost per change as a target for the partitioning, to make our algorithms suitable for use cases where vertex migration is expensive. Another interesting aspect is the lookahead potential of batch changes. Currently, we assign the vertices one-by-one regardless of the rest of the batch. By considering the entire batch, we could potentially improve the quality of the partition.

Large k values also remain an interesting target for future work. We have shown that our strategies uphold their qualitive and speed orderings for larger k values. But they all scale noticably with increasing k. For the quality, this is to be expected since the number of blocks increases. But for the running time, there may be strategies that are not as sensitive to the number of blocks.

There are many aspects of our strategies that could be improved. First of all, all our strategies are currently sequential. Especially for the costly v-cycles, this could lead to a significant speedup. Regarding the algorithms themselves, it would be interesting to explore the use of unconstrained refinement for our local FM strategy, considering the effect it has on static partitioning. This could make it viable for our local FM to be used in intervals instead of only at the beginning of the change list, similar to the v-cycle.

We have also provided many tuning possibilities. For example, the currently used v-cycle parameter is determined based on the evaluation at hand. It remains unnkown if there are optimal values, depending on the pattern of changes.

Bibliography

[1] M. R. Garey, D. S. Johnson, and L. Stockmeyer, “Some simplified NP-complete problems,” in *Proceedings of the sixth annual ACM symposium on Theory of computing*, 1974, pp. 47–63.

[2] I. Kabiljo *et al.*, “Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner.” [Online]. Available: https://arxiv.org/abs/1707.06665

[3] C. J. Alpert and A. B. Kahng, “Recent directions in netlist partitioning: a survey,” *Integration*, vol. 19, no. 1, pp. 1–81, 1995, doi: 10.1016/0167-9260(95)00008-4.

[4] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, “A repartitioning hypergraph model for dynamic load balancing,” *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 711–724, 2009, doi: 10.1016/j.jpdc.2009.04.011.

[5] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, “Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, in KDD '22. Washington DC, USA: Association for Computing Machinery, 2022, pp. 4582–4591. doi: 10.1145/3534678.3539177.

[6] A. Atrey, G. Van Seghbroeck, H. Mora, B. Volckaert, and F. De Turck, “UnifyDR: A generic framework for unifying data and replica placement,” *IEEE Access*, vol. 8, pp. 216894–216910, 2020.

[7] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden, “Schism: a workload-driven approach to database replication and partitioning,” 2010.

[8] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller, “SWORD: workload-aware data placement and replica selection for cloud data management systems,” *The VLDB Journal*, vol. 23, pp. 845–870, 2014.

[9] L. Gottesbüren, T. Heuer, N. Maas, P. Sanders, and S. Schlag, “Scalable High-Quality Hypergraph Partitioning,” *ACM Trans. Algorithms*, vol. 20, no. 1, Jan. 2024, doi: 10.1145/3626527.

[10] S. Barnard and H. Simon, “A fast implementation of recursive spectral bisection for partitioning unstructured problems,” *Parallel processing for scientific computing*, pp. 711–718.

[11] B. Hendrickson and R. Leland, “A multi-level algorithm for partitioning graphs.(1995).” Citeseer, 1995.

[12] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions,” in *Proceedings of the 19th Design Automation Conference*, in DAC '82. IEEE Press, 1982, pp. 175–181.

[13] N. Maas, L. Gottesbüren, and D. Seemaier, “Parallel Unconstrained Local Search for Partitioning Irregular Graphs,” in *2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 32–45. doi: 10.1137/1.9781611977929.3.

[14] Ü. Çatalyürek *et al.*, “More Recent Advances in (Hyper)Graph Partitioning,” *ACM Comput. Surv.*, vol. 55, no. 12, Mar. 2023, doi: 10.1145/3571808.

[15] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, “Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases.,” in *EDBT*, 2015, pp. 25–36.

[16] J. Huang and D. J. Abadi, “Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs,” *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 540–551, Mar. 2016, doi: 10.14778/2904483.2904486.

[17] W. Fan, M. Liu, C. Tian, R. Xu, and J. Zhou, “Incrementalization of graph partitioning algorithms,” *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1261–1274, Apr. 2020, doi: 10.14778/3389133.3389142.

[18] W. L. Lee *et al.*, "G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, in DAC '24. San Francisco, CA, USA: Association for Computing Machinery, 2024. doi: 10.1145/3649329.3656238.

[19] L. Durbeck and P. Athanas, "Incremental Streaming Graph Partitioning," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–8. doi: 10.1109/HPEC43674.2020.9286181.

[20] W. Ju, J. Li, W. Yu, and R. Zhang, "iGraph: an incremental data processing system for dynamic graph," *Frontiers of Computer Science*, vol. 10, p. , 2016, doi: 10.1007/s11704-016-5485-7.

[21] D. Dai, W. Zhang, and Y. Chen, "IOGP: An Incremental Online Graph Partitioning Algorithm for Distributed Graph Databases," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, in HPDC '17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 219–230. doi: 10.1145/3078597.3078606.

[22] C.-W. Ou and S. Ranka, "Parallel Incremental Graph Partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 884–896, Aug. 1997, doi: 10.1109/71.605773.

[23] G. Karypis, K. Schloegel, and V. Kumar, "Parmetis: Parallel graph partitioning and sparse matrix ordering library," p. , 1997.

[24] N. Xu, L. Chen, and B. Cui, "LogGP: a log-based dynamic graph partitioning method," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1917–1928, Oct. 2014, doi: 10.14778/2733085.2733097.

[25] K. Eyubov, M. F. Faraj, and C. Schulz, "FREIGHT: Fast Streaming Hypergraph Partitioning," in *21st International Symposium on Experimental Algorithms (SEA 2023)*, L. Georgiadis, Ed., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 265. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 1–16. doi: 10.4230/LIPIcs.SEA.2023.15.

[26] V. Osipov and P. Sanders, "n-Level Graph Partitioning," in *Algorithms – ESA 2010*, M. de Berg and U. Meyer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 278–289.

[27] S. Schlag, Y. Akhremtsev, T. Heuer, and P. Sanders, "Engineering a direct k-way Hypergraph Partitioning Algorithm," 2017, p. . doi: 10.1137/1.9781611974768.3.

[28] T. Heuer and S. Schlag, "Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure," in *16th International Symposium on Experimental Algorithms (SEA 2017)*, C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 75. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 1–19. doi: 10.4230/LIPIcs.SEA.2017.21.

[29] R. Andre, S. Schlag, and C. Schulz, "Memetic Multilevel Hypergraph Partitioning." [Online]. Available: https://arxiv.org/abs/1710.01968

[30] C. J. Alpert, "The ISPD98 circuit benchmark suite," in *Proceedings of the 1998 International Symposium on Physical Design*, in ISPD '98. Monterey, California, USA: Association for Computing Machinery, 1998, pp. 80–85. doi: 10.1145/274535.274546.

[31] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei, "The DAC 2012 routability-driven placement contest and benchmark suite," in *Proceedings of the 49th Annual Design Automation Conference*, in DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 774–782. doi: 10.1145/2228360.2228500.

[32] A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo, Eds., *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions.* in Department of Computer Science Series of Publications B. Finland: University of Helsinki, 2014.

Acknowledgments

The template and environment for this thesis was created by Dominik Lukas Rosch.

This thesis is written in the typesetting language typst[1] using the packages certz[2], ctheorems[3], and algo[4]. Graphics were created with R[5].

Proofreading and feedback was provided by my advisers.

Parts of scripts, like cosmetic improvements of the plots, where created with the help of GitHub Copilot[6] and debugged with the help of ChatGPT[7].

Also some translations were looked up using DeepL[8].

[1]typst.app
[2]https://typst.app/universe/package/cetz/
[3]https://typst.app/universe/package/ctheorems/
[4]https://typst.app/universe/package/algo/
[5]https://www.r-project.org/
[6]https://github.com/features/copilot
[7]https://chat.openai.com/chat
[8]https://www.deepl.com/translator

Appendix

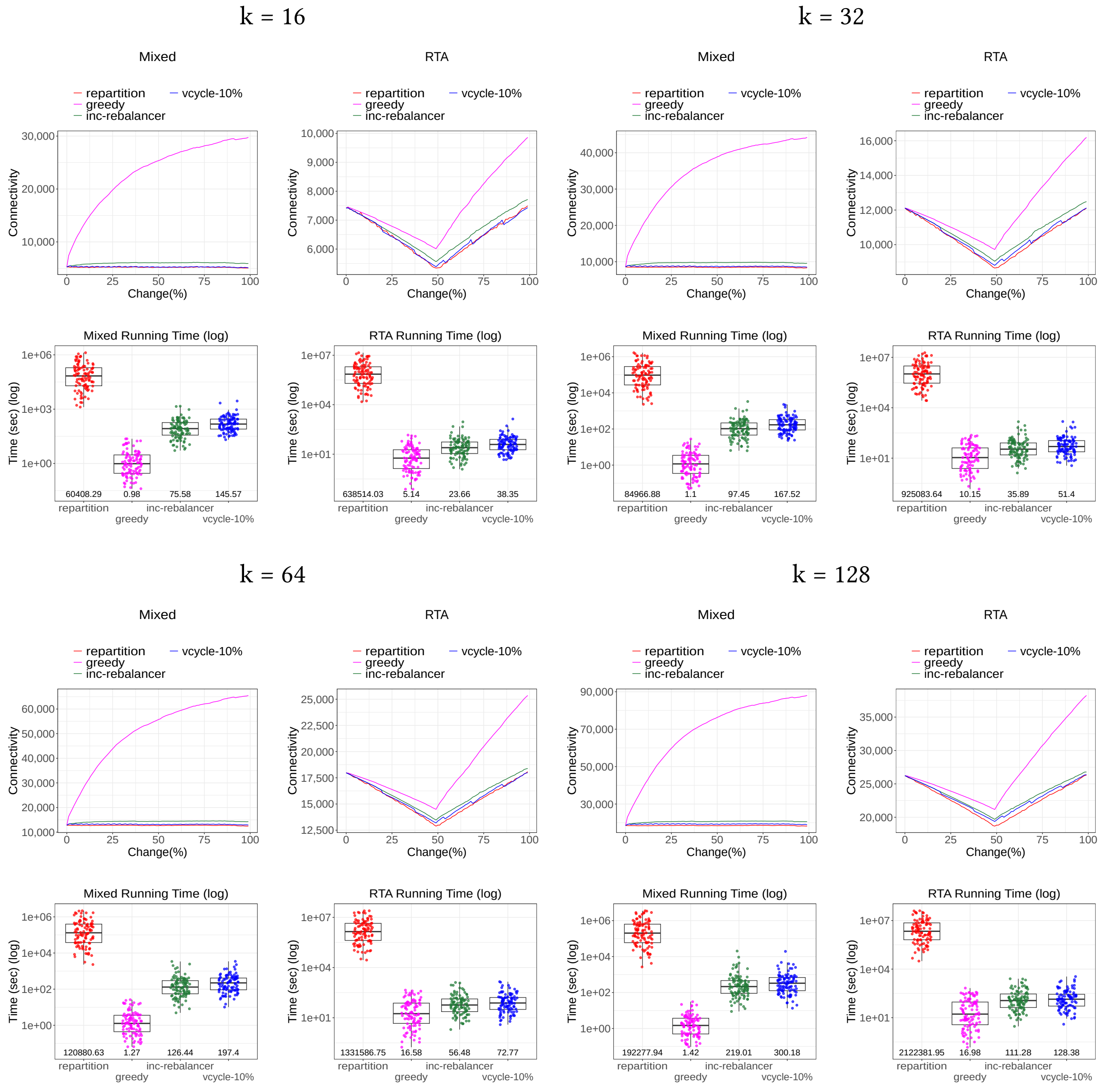

Figure 6: Quality and Speed for the mixed and remove then add change lists for $k \in \{16, 32, 64, 128\}$. The v-cycle step-size is set to 10. The small block threshold is set to 5.

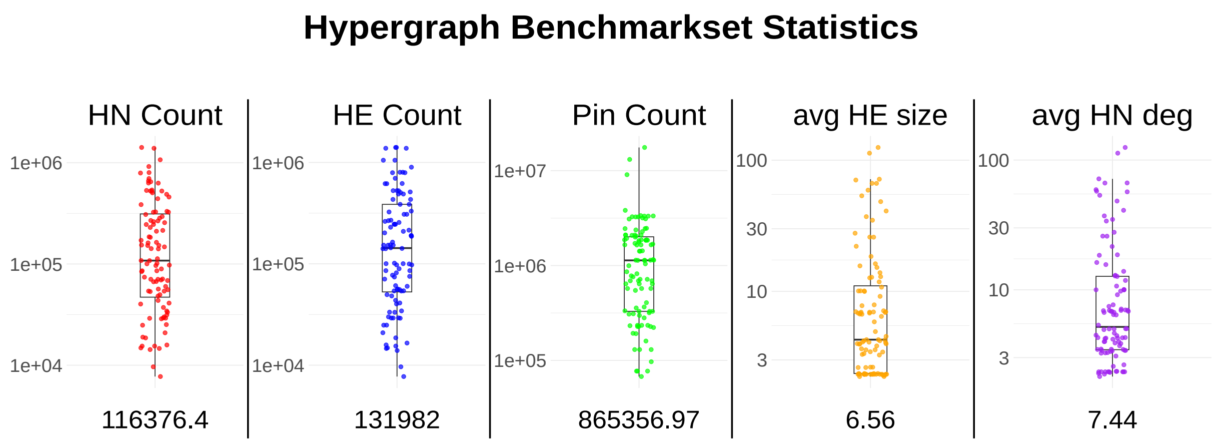

Figure 7: The number of hypernodes, hyperedges, pins, average size of hyperedges and average degree of hypernodes and their geometric means. The values are shown in logarithmic scale (base 10).

	HN Count	HE Count	Pin Count	avg HE size	avg HN deg
2cubes_sphere.mtx	101 492	101 492	1 647 264	16.2	16.2
2D_54019_highK.mtx	54 019	54 019	996 414	18.4	18.4
af_shell1.mtx	504 855	504 855	17 588 875	34.8	34.8
Andrews.mtx	60 000	60 000	760 154	12.7	12.7
as-caida.mtx	31 379	16 538	96 825	5.9	3.1
av41092.mtx	41 092	41 092	1 683 902	41.0	41.0
BenElechi1.mtx	245 874	245 868	13 150 490	53.5	53.5
c-61.mtx	43 618	43 618	310 016	7.1	7.1
case39.mtx	40 216	40 215	1 042 159	25.9	25.9
cfd1.mtx	70 656	70 656	1 828 364	25.9	25.9
ckt11752_dc_1.mtx	49 702	49 617	332 944	6.7	6.7
cnr-2000.mtx	325 557	208 961	3 177 612	15.2	9.8
dac2012_superblue14	630 802	619 789	2 048 877	3.3	3.2
dac2012_superblue16	698 339	697 416	2 280 375	3.3	3.3
dac2012_superblue19	522 482	511 653	1 713 764	3.3	3.3
dac2012_superblue3	917 944	897 935	3 109 380	3.5	3.4
denormal.mtx	89 400	89 400	1 156 224	12.9	12.9
gearbox.mtx	153 746	153 746	9 080 404	59.1	59.1
hvdc1.mtx	24 842	24 842	159 981	6.4	6.4
Ill_Stokes.mtx	20 896	20 896	191 368	9.2	9.2
ISPD98_ibm09	53 395	60 902	222 088	3.6	4.2
ISPD98_ibm10	69 429	75 196	297 567	4.0	4.3
ISPD98_ibm11	70 558	81 454	280 786	3.4	4.0
ISPD98_ibm12	71 076	77 240	317 760	4.1	4.5
ISPD98_ibm13	84 199	99 666	357 075	3.6	4.2
ISPD98_ibm14	147 605	152 772	546 816	3.6	3.7
ISPD98_ibm15	161 570	186 608	715 823	3.8	4.4

	HN Count	HE Count	Pin Count	avg HE size	avg HN deg
ISPD98_ibm16	183 484	190 048	778 823	4.1	4.2
ISPD98_ibm17	185 495	189 581	860 036	4.5	4.6
ISPD98_ibm18	210 613	201 920	819 697	4.1	3.9
laminar_duct3D.mtx	67 173	54 149	3 820 053	70.5	56.9
lhr14.mtx	14 270	13 944	307 532	22.1	21.6
light_in_tissue.mtx	29 282	29 282	406 084	13.9	13.9
Lin.mtx	256 000	256 000	1 766 400	6.9	6.9
lp_pds_20.mtx	108 175	33 250	232 099	7.0	2.1
m14b.mtx	214 765	214 765	3 358 036	15.6	15.6
mc2depi.mtx	525 825	525 825	2 100 225	4.0	4.0
mixtank_new.mtx	29 957	29 957	1 995 041	66.6	66.6
mult_dcop_01.mtx	25 187	24 817	192 906	7.8	7.7
opt1.mtx	15 449	15 449	1 930 655	125.0	125.0
poisson3Db.mtx	85 623	85 623	2 374 949	27.7	27.7
powersim.mtx	15 838	15 838	67 562	4.3	4.3
Pres_Poisson.mtx	14 822	14 822	715 804	48.3	48.3
RFdevice.mtx	74 104	74 104	365 580	4.9	4.9
rgg_n_2_18_s0.mtx	262 144	262 105	3 094 530	11.8	11.8
sat14_6s133.cnf	96 430	140 967	328 923	2.3	3.4
sat14_6s133.cnf.dual	140 968	48 215	328 924	6.8	2.3
sat14_6s133.cnf.primal	48 215	140 967	328 923	2.3	6.8
sat14_6s153.cnf	171 292	245 439	572 691	2.3	3.3
sat14_6s153.cnf.dual	245 440	85 646	572 692	6.7	2.3
sat14_6s153.cnf.primal	85 646	245 439	572 691	2.3	6.7
sat14_6s184.cnf	66 730	97 515	227 535	2.3	3.4
sat14_6s184.cnf.dual	97 516	33 365	227 536	6.8	2.3
sat14_6s184.cnf.primal	33 365	97 515	227 535	2.3	6.8
sat14_6s9.cnf	68 634	100 383	234 227	2.3	3.4
sat14_6s9.cnf.dual	100 384	34 317	234 228	6.8	2.3
sat14_6s9.cnf.primal	34 317	100 383	234 227	2.3	6.8
sat14_aaai10-planning-ipc5-path-ways-17-step21.cnf	107 838	307 821	690 052	2.2	6.4
sat14_aaai10-planning-ipc5-path-ways-17-step21.cnf.dual	308 235	53 919	690 466	12.8	2.2
sat14_aaai10-planning-ipc5-path-ways-17-step21.cnf.primal	53 919	307 821	690 052	2.2	12.8
sat14_ACG-20-5p0.cnf	649 432	1 379 528	3 257 729	2.4	5.0
sat14_ACG-20-5p0.cnf.dual	1 390 931	324 716	3 269 132	10.1	2.4
sat14_ACG-20-5p0.cnf.primal	324 716	1 379 528	3 257 729	2.4	10.0
sat14_ACG-20-5p1.cnf	662 392	1 405 447	3 322 128	2.4	5.0
sat14_ACG-20-5p1.cnf.dual	1 416 850	331 196	3 333 531	10.1	2.4

	HN Count	HE Count	Pin Count	avg HE size	avg HN deg
sat14_ACG-20-5p1.cnf.primal	331 196	1 405 447	3 322 128	2.4	10.0
sat14_AProVE07-27.cnf	15 458	29 193	77 123	2.6	5.0
sat14_AProVE07-27.cnf.dual	29 194	7 729	77 124	10.0	2.6
sat14_AProVE07-27.cnf.primal	7 729	29 193	77 123	2.6	10.0
sat14_atco_enc1_opt2_05_4.cnf	28 738	386 064	1 652 701	4.3	57.5
sat14_atco_enc1_opt2_05_4.cnf.dual	386 163	14 618	1 652 782	113.1	4.3
sat14_atco_enc1_opt2_05_4.cnf.primal	14 636	386 064	1 652 701	4.3	112.9
sat14_atco_enc1_opt2_10_16.cnf	18 930	152 611	641 006	4.2	33.9
sat14_atco_enc1_opt2_10_16.cnf.dual	152 744	9 641	641 137	66.5	4.2
sat14_atco_enc1_opt2_10_16.cnf.primal	9 643	152 611	641 006	4.2	66.5
sat14_atco_enc2_opt1_05_21.cnf	112 732	526 856	2 097 377	4.0	18.6
sat14_atco_enc2_opt1_05_21.cnf.dual	526 872	56 533	2 097 393	37.1	4.0
sat14_atco_enc2_opt1_05_21.cnf.primal	56 533	526 856	2 097 377	4.0	37.1
sat14_countbitssrl032.cnf	37 213	55 722	130 018	2.3	3.5
sat14_countbitssrl032.cnf.dual	55 724	18 606	130 019	7.0	2.3
sat14_countbitssrl032.cnf.primal	18 607	55 722	130 018	2.3	7.0
sat14_dated-10-11-u.cnf	283 720	617 431	1 417 842	2.3	5.0
sat14_dated-10-11-u.cnf.dual	629 461	141 860	1 429 872	10.1	2.3
sat14_dated-10-11-u.cnf.primal	141 860	617 431	1 417 842	2.3	10.0
sat14_dated-10-17-u.cnf	459 088	1 051 601	2 451 966	2.3	5.3
sat14_dated-10-17-u.cnf.dual	1 070 757	229 544	2 471 122	10.8	2.3
sat14_dated-10-17-u.cnf.primal	229 544	1 051 601	2 451 966	2.3	10.7
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf	327 243	488 118	1 138 942	2.3	3.5
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.dual	488 120	163 621	1 138 943	7.0	2.3
sat14_hwmcc10-timeframe-expansion-k45-pdtvisns3p02-tseitin.cnf.primal	163 622	488 118	1 138 942	2.3	7.0
sat14_itox_vc1130.cnf	294 326	431 542	1 133 787	2.6	3.9
sat14_itox_vc1130.cnf.dual	441 729	143 918	1 135 636	7.9	2.6
sat14_itox_vc1130.cnf.primal	152 256	431 542	1 133 787	2.6	7.4
sat14_manol-pipe-c8nidw.cnf	538 096	799 866	1 866 354	2.3	3.5
sat14_manol-pipe-c8nidw.cnf.dual	799 867	269 048	1 866 355	6.9	2.3
sat14_manol-pipe-c8nidw.cnf.primal	269 048	799 866	1 866 354	2.3	6.9
sat14_manol-pipe-g10bid_i.cnf	532 810	792 174	1 848 406	2.3	3.5
sat14_manol-pipe-g10bid_i.cnf.dual	792 175	266 405	1 848 407	6.9	2.3
sat14_manol-pipe-g10bid_i.cnf.primal	266 405	792 174	1 848 406	2.3	6.9
sme3Db.mtx	29 067	29 067	2 081 063	71.6	71.6

Table 2: Benchmark set statistics