Engineering Hypergraph Compression Algorithms for Hypergraph Partitioning

Bassit Agbéré

September 30, 2025

3718104

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Adil Chhabra, Daniel Seemaier, Nikolai Maas

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Christian Schulz for introducing me to the very fundamentals of computer science and algorithm engineering throughout my studies and for enabling me to write this thesis under his supervision. Furthermore, I am deeply thankful to my co-supervisors Adil Chhabra, Daniel Seemaier and Nikolai Maas. They have provided scientific guidance, critical ideas and have equipped me with decisive insight into the large codebase I worked in during this thesis. Lastly, I would like to thank my friends and family who have allowed me to focus on my thesis by supporting me in many ways.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, September 30, 2025

B. Agbéré

Bassit Agbéré

Abstract

Hypergraphs are a generalisation of the graph data structure where edges are allowed to connect an arbitrary number of nodes instead of exactly two. Consequently, they are not limited to modelling binary relationships and increase the flexibility of traditional graphs. However, their applicability is still bound by memory constraints. This thesis develops a compressed hypergraph representation which is then implemented in the parallel partitioning library Mt-KaHyPar. In the stand-alone representation, Hypergraphs are translated into a bipartite graph where hyperedges are represented by artificially inserted vertices, which are then connected to all of the vertices they connected in the original graph. In Mt-KaHyPar, the compressed hypergraph is partitioned 36% slower on average, but has identical partition quality and achieves a mean compression ratio of ≈ 2.027 to store the graph and ≈ 1.542 during partitioning.

Contents

List of Abbreviations

CSA	compressed suffix array
CSR	compressed sparse row representation
HV	hyperedge vertices
Mt-KaHyPar	Multi-Threaded Karlsruhe Graph and Hypergraph Partitioner
MGP	Multilevel Graph Partitioning
NAR	nested array representation
OV	original vertices

CHAPTER 1

Introduction

1.1 Motivation

Graphs are fundamental to a wide range of applications. They power navigation software on global street networks, allow the efficient storage and lookup of complex relational data, drive web search engines and even enable artificial intelligence implementations. [13, 15, 10]

However, graphs are limited to modelling binary relationships. A traditional graph consists of nodes and edges. While both of these can have weights—for example, to indicate 'distance' when edges model roads or 'throughput' when they model pipes—and both directed and undirected relationships can be captured, one edge can only connect precisely two vertices. Hypergraphs extend the capabilities of traditional graphs, as "hyperedges"—sometimes referred to as "nets"—can connect an arbitrary number of vertices instead of exactly two. This grants them high flexibility in modelling complex relationships, as they are essentially just elements and groupings that can combine these elements arbitrarily. Possible practical applications range from modelling intricate social and neural networks to circuit boards and co-authorships [3, 8]. On the other hand, they allow the simplification of theoretical insights about graphs, sometimes greatly simplifying theorems that are complex when defined with traditional graphs. [3]

With their ability to model extensive real-world problems, they also inherit an issue directly from their traditional graph parent: size. The larger the graph instances get, the higher the difficulty of working with them, as applications such as graph partitioning require graphs to fit into memory [5]. In some cases, large instances like *webbase* or *Sat14* cannot be partitioned at all because the partitioner runs out of memory or reaches a timeout [11]. There are a few different strategies to deal with very large instances, each with its own advantages and drawbacks. Using a larger machine with more memory, for example, is associated with higher costs and may not be practical, as real examples of large graphs may take up to 800 GB of memory during partitioning without memory optimisations [16, 18]. This amount of memory may simply be unavailable at a given budget. Furthermore, a higher memory consumption might result in longer processing times if virtual memory is used or may lead to lower solution quality. On the other hand, sharing the processing load across multiple machines inquires all of the problems typically associated with distributed memory systems: Communication overhead, latency, infrastructure cost, load balancing and overall added programming complexity [22]. This thesis aims to address this problem by utilising streaming and compression techniques to reduce memory footprints, increasing the set of graphs that can be processed on a given machine.

1.2 Contribution

This thesis develops a compressed, memory-efficient hypergraph representation that can be streamed from a file with minimal memory overhead. This representation is then adapted for and applied to the shared-memory hypergraph partitioning library Mt-KaHyPar. During the streaming process, the input hypergraph is transformed into the bipartite graph structure outlined in Section 2.3. The representation achieves its memory savings by employing various encoding techniques introduced in Section 2.4. Basic metadata about the graph, such as the number of nodes and edges of the bipartite graph and the number of hyperedges in the original graph, is stored alongside node adjacency lists, which are decompressed on access.

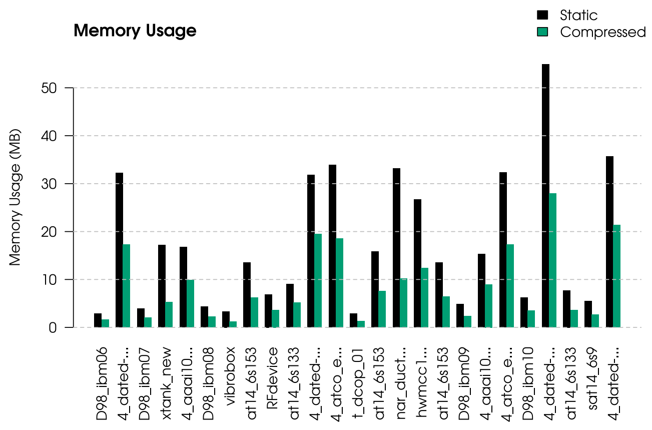

Figure 1.1: Hypergraph Compression Ratio

For use in Mt-KaHyPar, the streaming algorithm is adapted to support weights and other requirements for partitioning. The resulting data structure is a fully featured substitute for the existing static hypergraph. It can be used to partition hypergraphs while significantly reducing their memory footprint. Experiments indicate that while it often lowers construction times, it significantly increases partition runtimes. Partition quality remains unaffected by compression except for minor random variation caused by heuristics.

1.3 Structure

After various central concepts and compression techniques are introduced in Chapter 2, related work and recent developments are presented in Chapter 3. The main contribution is located in Chapter 4. It examines hypergraph compression techniques, develops an algorithm to stream them from a file and implements compression into Mt-KaHyPar. Finally, we will run experiments on hypergraphs from several benchmark sets [18]. These experiments examine the efficacy of different compression schemes and evaluate the impact of compression on the running time and memory consumption during graph partitioning in Mt-KaHyPar. Lastly, Chapter 6 will summarise the results and suggest further research.

CHAPTER 2

Fundamentals

This section introduces terms and definitions which are crucial for understanding the problems that arise during hypergraph compression and the solutions outlined later.

2.1 Data structures

Graph. A graph $G = (V, E)$ is a mathematical structure defined as a set of vertices (or nodes) V and edges E. An edge connects precisely two vertices. In a *directed* graph, edges have a specific source and target node, meaning the described connection is one-way. The number of edges a node is connected to is referred to as its *degree*. Graphs can also be weighted, meaning either their vertices, edges, or both can be assigned weights [23]. In the following, the term graph will refer to an undirected weighted graph, and a weight of 1 will be assumed as the default for vertices and edges if not otherwise specified. Figure 2.1 visualises an undirected graph. Each node is assigned a natural number as a unique identifier.

Hypergraph. Hypergraphs $H = (V, E)$ are a generalisation of traditional graphs where hyperedges (or nets) can connect an arbitrary number of vertices instead of exactly two. This allows a hypergraph to model any n-ary relationship. [3]

2.2 Graph Representations

In the following, we will compare two methods to store graphs using adjacency lists. An adjacency list is a list of vertex IDs indicating the neighbourhood of each vertex, i. e., all of the vertices it is connected to. In the undirected case, each edge is inserted symmetrically, meaning the target vertex has the source vertex in its adjacency list and vice versa.

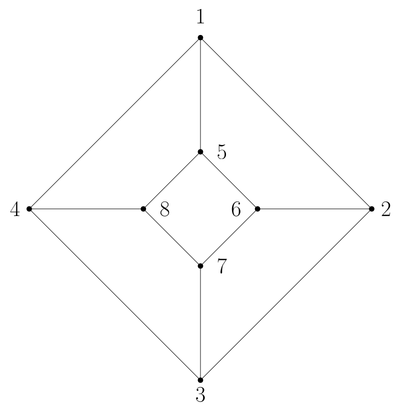

Figure 2.1: Graph Visualisation

The terms target and source vertex here can be assigned arbitrarily. In the directed case, however, only the target node of an edge is added to the adjacency list of the source node.

Nested Array Representation. The first way to store a graph using adjacency lists is a nested array representation (NAR). This means that the graph is stored in a single parent array. At the index of each vertex ID, this array contains the adjacency list of the respective vertex, which is an array of vertex IDs. Using a dynamic array data structure to achieve this task, the main advantage of this representation is its flexibility. Given that the array has an amortised insertion complexity of $\mathrm{O}(1)$, inserting an edge also takes constant time. Its main disadvantage is the additional storage which might be required. Usually, such an array has to store its current length, its capacity and its storage location. In the case of a C++ standard library [9] *vector*, this translates to 24 Bytes of memory per vector. In this representation, each node must store its own adjacency list. Thus, it must store a pointer to a vector, which typically takes up two bytes and it must store the vector itself, which takes up a total of 24 bytes as just established. Thus, each node requires 32 bytes of storage for the *empty* adjacency lists alone. This can become highly relevant when the graph has many vertices, and it can account for a big proportion of the storage size if adjacency lists are small.

Compressed Sparse Row Representation. The compressed sparse row representation (CSR) is a memory-efficient graph storage format where all adjacency lists are stored in a single array. A second array is utilised to store the offset, i. e., the beginning of each adjacency list. Thus, the total number of arrays stored is reduced to two for an arbitrarily large graph, given that the array can be large enough to contain all adjacency lists. Without sufficient gaps left between adjacency arrays, all subsequent nodes in the adjacency array have to be moved when an edge is added. Consequently, the insert performance on this data structure is $\mathrm{O}(m)$, where m denotes the number of edges. Note that in the undirected case, the constant factor 2 is applied due to the symmetry described above. If gaps are left at the end of each section, most of the memory efficiency is lost. Hence, this data structure is most suited if graphs do not change or when the sizes of the adjacency arrays are known beforehand. In this case, no capacity has to be stored; the end of one adjacency list is simply the beginning of the next one.

2.3 Bipartite Hypergraph Representation

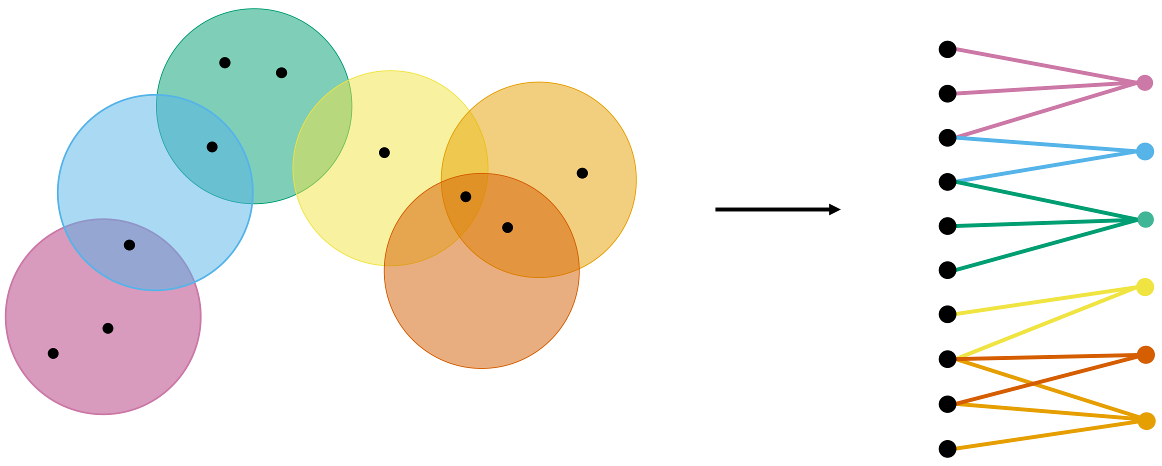

Figure 2.2: Bipartite Hypergraph Representation[1]

A hypergraph can be transformed into a bipartite graph by inserting an ‘artificial’ vertex for each hyperedge. To distinguish them from the original vertices (OV) that were already a part of the hypergraph, they are henceforth referred to as hyperedge vertices (HV). Afterwards, an edge is inserted to every vertex that the hyperedge is connected to. This is illustrated in Figure 5.4.

2.4 Encoding Techniques

This section provides a summary of the encoding techniques which will later be examined for their effectiveness on hypergraphs.

Delta Encoding. This encoding technique reduces the size of numbers stored in an array by replacing the absolute value of each number with the distance to its predecessor. While using signed integers would allow this scheme to be employed on unsorted arrays, it could lead to larger numbers having to be stored. Thus, the array must be sorted to minimise the size of the stored numbers [2]. For example, delta encoding would transform the sequence $\{1, 2, 5, 17, 18\}$ to $\{1, 1, 3, 12, 1\}$

[1]Figure inspired by ‘A hypergraph with six hyperedges’ [12], colour palette: Okabe-Ito [14]

Variable-Length Encoding. If numbers are stored in blocks, e. g., 32-bit integers, delta encoding does nothing to reduce the size of the array. Instead, it simply leads to more unused bits in each number block. This is where variable-length encoding comes in. Instead of storing numbers in blocks large enough to fit the biggest value, the size of each block is reduced to a minimum—usually one byte. The first bit of each block is then used to indicate if the number ends in the current block. In the example with one-byte blocks, this is the case if a number is 7 bits long or smaller. If it does not, the first bit is flipped, and the number is continued in the next block. This continues until there are enough bits to store the number. Since delta encoding alone does not reduce the size of adjacency lists, in the following, the term **gap encoding** will be used to refer to a combination of delta and variable-length encoding.

Interval Encoding. Given a sequence of integers $7, 8, ..., n$, the idea of interval encoding is to store only the starting and end points $7 - n$ or the starting point and length of the sequence instead of every number in it [2]. However, when iterating over an interval-encoded sequence, there has to be a way to distinguish between sequences and regular number values. This can be executed by inserting a special marker that indicates the start of an interval. However, this marker requires memory. Thus, in practice, only intervals which are greater than a certain threshold are considered to prevent unnecessary overhead from encoding small sequences as an interval. [16]

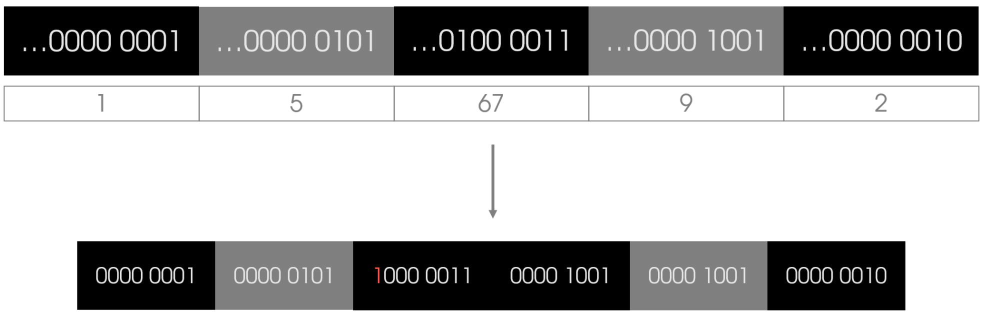

Figure 2.3: Illustration of Variable Length Encoding

CHAPTER 3

Related Work

3.1 Mt-KaHyPar

The Multi-Threaded Karlsruhe Graph and Hypergraph Partitioner (Mt-KaHyPar) is a shared-memory partitioner for graphs and hypergraphs. It is designed to efficiently handle large instances with high solution quality and scalable performance. In benchmarks comparing it against leading partitioners such as KaHyPar [19] and KaFFPa [17], it consistently matches or surpasses their quality while delivering substantial runtime improvements. This is especially true in parallel environments, and it can be scaled to a high number of threads without adversely affecting the solution quality. [6]

In addition to its parallel scalability, Mt-KaHyPar integrates a comprehensive set of algorithmic techniques that have proven central to high-quality hypergraph partitioning. Following the multilevel paradigm, it combines advanced coarsening schemes with diverse initial partitioning strategies to generate strong starting solutions. During uncoarsening, several refinement methods—such as flow-based refinements, localised FM-style heuristics, and evolutionary recombination operators—are applied to iteratively improve solution quality. This combination of complementary algorithms balances cut size and load distribution, thereby matching or surpassing the quality of the best sequential partitioners while fully exploiting parallelism. The framework is also highly modular and flexible, making it a well-suited candidate for further research and optimisation. One of its most notable limitations is that it cannot scale past system RAM [4]. Hence, reducing the memory footprint of the graphs directly extends the range of instances that can be processed on a given machine. A successful compression implementation that does not decrease solution quality can thus further increase the applicability of Mt-KaHyPar.

3.2 KaMinPar

KaMinPar is a scalable, parallel multilevel graph partitioner built on the deep paradigm [5]. The traditional approach to Multilevel Graph Partitioning (MGP) is to shrink the graph down into smaller, more manageable instances and solve the partition problem there. This 'shrinking' is referred to as graph coarsening. The idea is to reduce the size and complexity of a large graph while preserving its most important structural properties. After the coarsened graph has been partitioned, it is gradually brought back to its original size while the solution is refined at each uncoarsening step.

Deep MGP pushes the coarsening much further. The graphs are shrunk down to a multiple of the processing units or a small constant. Subsequently, the small coarse graph is not partitioned into many blocks at once (k-way partitioning). Instead, deep MGP solves the problem by recursive bipartitioning. This means the graph is repeatedly divided into two blocks during uncoarsening until the desired number of blocks is reached. Nevertheless, KaMinPar also integrates aspects of classical direct [5].

During coarsening and refinement, size-constrained label propagation is used as the clustering algorithm. Cluster size is restricted by an upper bound tied to block sizes, which ensures feasible and balanced partitions. Since the introduction of edge sparsification, there are also linear runtime guarantees [7].

In his thesis, Salwasser [16] examines strategies for effectively reducing the memory requirements of KaMinPar. He develops a compressed graph representation that utilises the compression techniques highlighted above. In addition to that, he develops a Two-Level Cluster Weight Vector. Cluster weights track the total weight of nodes in each cluster. They are stored to ensure constraints during graph partitioning. To avoid using a full 8 bytes for each weight, the first level has a definable restricted storage capacity, e. g., one or four bytes. Should the capacity be exceeded, the excess weight is stored in a dynamically growing hash table. This reduces memory consumption without significant performance losses.

Through the outlined methods and other integrated memory optimisations, Salwasser achieves an $8.6\times$ reduction in peak memory usage. His work demonstrates that careful memory optimisation and compression can extend the applicability of in-memory graph partitioners to large graph instances without sacrificing speed or solution quality. The proposed techniques are integrated into the framework, enabling it to partition some of the largest real-world graphs on a single machine with limited RAM.

3.3 Ligra and Hygra

In 2013, Ligra was introduced as a lightweight shared-memory graph processing framework which is optimised for parallel execution on multicore machines. It was later extended to support Hypergraphs, thus creating Hygra. Ligra has one routine to map over edges and one to map over vertices and aims to facilitate writing graph traversals [21]. However, in a more recent paper, one of the authors establishes a lack of high-performance hypergraph algorithms despite advancements in parallel graph processing. [20] Thus, the Ligra graph processing framework is extended to support hypergraphs, resulting in the Hygra framework. Hygra uses a bipartite graph representation to store hypergraphs and leverages techniques such as direction optimisation, edge-aware parallelisation, bucketing for load balancing, and compression. The paper demonstrates which high-performance parallel hypergraph processing tasks can become feasible on a single multicore machine through optimisation and compression. Furthermore, the proposed algorithms enable the analysis of hypergraphs that are too large to be used in existing frameworks.

3.4 HyperCSA

Recently, there has been novel approaches to compressing Hypergraphs in works like HyperCSA, further highlighting the need for effective compression solutions. Its approach is based on compressed suffix arrays (CSAs), which was inspired by text indexing. The entire hypergraph is transformed into a string and, by sorting nodes and edges, encoded into a CSA. This achieves a compact, query-efficient representation that outperforms query times of existing solutions. Additionally, it enables file size reductions by 26% up to 79%. [1]

Despite this, HyperCSA is not immediately suitable for implementation into Mt-KaHyPar. While the achieved file-based compression ratios are substantial, Mt-KaHyPar relies on fast in-memory access during the partitioning process. At first glance, the in-memory querying times of HyperCSA should still make it a promising solution. However, Mt-KaHyPar stores not only hyperedges, but also pre-computes the incidence lists of all nodes. Naturally, dropping these pre-computed lists would greatly reduce memory consumption, but significantly slow down incidence queries. HyperCSA, on the other hand, stores neither edges nor incidence lists as directly accessible arrays, as it fully relies on the CSA. Thus, even if the entire string representation is kept in memory, iterating through a neighbourhood requires complex index operations, which will cause a significant slowdown compared to constant-time array access. Consequently, the CSA approach is not trivially applicable to Mt-KaHyPar.

CHAPTER 4

Compression Algorithm and Implementation

Before tackling the implementation of hypergraph compression in a project with concrete requirements and restrictions, we will develop a memory-efficient strategy to stream and store hypergraphs.

4.1 Graph Transformation

Despite the advantages that hypergraphs have, an extensive literature exists on graph algorithms that concern themselves with and run on traditional graphs. To apply these algorithms—and libraries which implement them—on hypergraphs, it is first necessary to convert them into a regular graph using the bipartite representation outlined in Section 2.3. For a representation using nested arrays, Algorithm 1 achieves the described transformation.

4.2 Graph Representation

To systematically assess the ideal compression approach, we will sketch the maximally compressed lossless representation of a hypergraph given the techniques outlined in chapter 2. Then, we reintroduce concrete requirements as we weigh memory savings against runtime penalties.

As the set of hyperedges is sufficient to completely describebe a hypergraph, a maximally compressed representation would not be bipartite. The advantage of the bipartite transformation is that the entire incidence list of a node— meaning the list of all hyperedges it is connected to—is already stored in memory and can be accessed immedi-

Algorithm 1: Hypergraph to Bipartite Representation Conversion

```
Procedure ConvertToBipartite(Hypergraph H)
    $Adj \leftarrow$ NAR of size $(H.NumVertices + H.NumHyperedges)$;
    $i \leftarrow H.NumVertices$;
    for each $HE \in H.Hyperedges$ do
        $i \leftarrow i + 1$;
        $Adj[i] \leftarrow HE$;
        for each $j \in H.Edges$ do
            $E[j] \leftarrow i$;
    return $\langle V, E \rangle$;
```

ately. Nevertheless, this information is still present in the hyperedges: The incidence list of a given node can be generated by simply iterating through all hyperedges and checking whether the node ID is present. If the hyperedges are sorted, this can be done in $\mathrm{O}(n_{hyperedges} \cdot \log MaxHyperedgeSize)$.

To avoid storing the size and capacity of each hyperedge, a CSR representation is used. This means that all hyperedges are stored in a single array. A second offset array is required to store the starting index of each hyperedge. Then, varint and delta encoding are applied, which requires the node IDs in each hyperedge to be sorted in ascending order. The first entry of each hyperedge is not delta encoded. This allows the edge-wise decompression of the array. Finally, interval encoding is applied to further reduce the memory footprint.

Whether this maximally compressed representation is feasible or not heavily depends on the use case of the graph. If node incidence lists are not frequently needed, and most queries iterate through node IDs in a hyperedge, this representation is very economical and might have tolerable runtime penalties. In this case, HyperCSA [1] is likely the best fit. It evaluates many queries faster than the sketched data structure and likely has better compression, especially on large instances.

Moreover, this non-bipartite representation can only be used when the application that consumes the graph is tailored to hypergraphs, as it does not describe any binary edges. For cases where fast incidence queries are important, such as during hypergraph partitioning, a bipartite representation is superior, even though it doubles the memory usage of the graph.

One requirement that renders both CSR arrays and HyperCSA infeasible is flexibility. Frequent updates are expensive in HyperCSA, as the index and the suffix array might have to be completely rebuilt. As for the CSR array, adding a single node to any hyperedge requires moving the entire array after the position of the node, as its adjacency list has to be updated. Thus, neither approach is suitable for dynamic use cases.

In a nutshell, HyperCSA is the best fit for a static representation with infrequent incidence

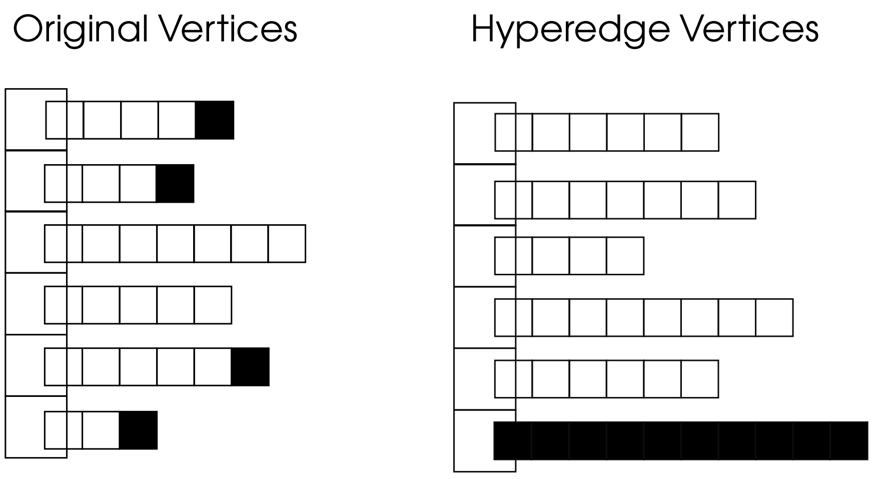

Figure 4.1: Insertion of Hyperedges

queries. A gap-encoded CSR array is the most memory-efficient way to store a bipartite graph representation, but it are infeasible for dynamic applications. NARs are most suited for dynamic use cases, although they require 16 additional bytes of memory for each node and hyperedge.

4.3 Streaming Algorithm

In the following, we will assume that a given hypergraph fits into memory when fully compressed. The ideal scenario when reading it from a file is that it is directly converted into a fully compressed format while it is still being read. This would mean that the memory consumption would gradually increase during the construction process until the hypergraph is fully read. Necessarily, this would mean that the entire streaming operation would never exceed the memory capacity of the machine. Otherwise, the assumption that the compressed graph fits into memory would be violated.

Therefore, it is necessary to assess which compression techniques can be applied during the construction phase. The main problem that arises is the OV. For each vertex ID in a hyperedge, two adjacency lists have to be updated: The hyperedge vertex ID has to be inserted into the original vertex's list, and it, in turn, has to be inserted into the newly created hyperedge's list. As a consequence, compressing the adjacency lists of the HV is straightforward: Since they are artificially inserted, they are guaranteed not to be connected to any other HV. Thus, the adjacency list of each HV is exactly equal to the hyperedge it represents, will not have to be altered later and can be fully compressed directly after being read. The OV, on the other hand, have to be updated constantly. Therefore, while the CSR is more storage efficient, the NAR is initially used for the OV to achieve constant-time insertions.

As mentioned above, gap encoding requires numbers to be sorted to achieve the best outcome. While the OV have to be updated constantly, the construction of the bipartite representation grants a property which can be exploited here. Since the IDs of the HV are simply integers starting from the number of OV + 1, they are parsed in ascending order. This means that, once a given ID n has been inserted into any given adjacency list of an OV, there will never be any ID $m < n$ which has to be inserted after it. Consequently, the adjacency arrays can be immediately stored in a compressed format. Note that the first node ID of each array is not gap encoded. If this were the case, the entire previous array would have to be iterated through to calculate the absolute value of the starting vertex. Thus, each starting vertex is stored as-is. In the end, the graph is converted into the CSR. This procedure is outlined in Algorithm 2.

Algorithm 2: Streaming Algorithm

```
Procedure StreamAndCompress(Hypergraph H)
    $OV \leftarrow$ dynamic array of size $H.NumVertices$;
    $HV \leftarrow$ dynamic array of size $H.NumHyperedges$;
    $hyperedge\_id \leftarrow H.NumVertices + 1$;
    for each hyperedge $h$ read from disk do
        $hyperedge\_id \leftarrow hyperedge\_id + 1$;
        $HV[hyperedge\_id] \leftarrow$ gap_encode$(h)$;
        for each vertex $v$ in $h$ do
            gap $\leftarrow hyperedge\_id - OV[v].back()$;
            $OV[v] \leftarrow OV[v] \cup \{gap\}$;
    $VertexArray \leftarrow$ array of size $(H.NumVertices + H.NumHyperedges)$;
    $EdgeArray \leftarrow$ array;
    offset $\leftarrow 1$;
    for AdjacencyArray $a$ in $OV$ do
        $VertexArray \leftarrow VertexArray \cup \{$offset$\}$;
        $EdgeArray \leftarrow EdgeArray \cup a$;
        $offset \leftarrow$ offset $+ a.length()$;
    for each AdjacencyArray $a$ in $HV$ do
        $VertexArray \leftarrow VertexArray \cup \{$offset$\}$;
        $EdgeArray \leftarrow EdgeArray \cup a$;
        offset $\leftarrow$ offset $+ a.length()$;
```

4.3.1 Buffer

To account for cases where the graph is too large to fit into memory, partially compressed during IO, a buffer is added to the procedure. This buffer stores all HV. Once full, the adjacency lists are stored in a binary chunk file on disk. The properties outlined above ensure that these files will only have to be read again once in the final stage of the procedure. The OV accordingly remain in memory, as they are frequently used and chunking them on disk would greatly reduce performance. Once all of the hyperedges are read, first, all of the OV are converted into the CSR. This frees up a maximal amount of memory. Then, the chunk files are merged and appended to the edge array in order

A smaller buffer size implies a lower peak memory consumption, but also a longer IO time, as more chunks have to be written and merged. The following figure shows the relationship between the compression time and the buffer size.

4.4 Mt-KaHyPar Implementation

In this section, the insight gained through the stand-alone compression experiments is applied to the hypergraph partitioning library Mt-KaHyPar.

4.4.1 Data Structure

The current data structure in Mt-KaHyPar that the compressed representation will replace is the static hypergraph. As the compression techniques highlighted above rely on densely packed arrays, frequent rebuilding is computationally expensive. Since there are no gaps to account for edge and node insertions, the entire CSR array would have to be rebuilt or additional memory would have to be allocated, directly counteracting the compression.

Consequently, the compressed data structure will serve as a substitute for the static hypergraph. The representation in Mt-KaHyPar differs slightly from the bipartite approach outlined above. No artificial nodes are inserted to represent hyperedges. Instead, there is an explicit separation of hyperedges and node adjacency lists. Both are stored in a separate CSR array. However, the offset array does not merely contain the starting positions of nodes and edges in the respective list. Instead, it is an array of objects which store additional information:

Thus, for a hypergraph with n_{hn} hypernodes and n_{he} hyperedges, the storage requirement without any adjacency lists is already $25(n_{hn} + n_{he})$ bytes. Most of these values cannot be omitted, as they are crucial for the partitioning procedure. Thus, different strategies are applied for each of these properties.

To store the hyperedges and the incidence arrays of each node, two CSR arrays are created. A second array then stores the offset of each node and edge respectively. The ID serves as

Hypernode / Hyperedge	**Memory (Bytes)**
Degree / Size	8
Begin (start index in CSR array)	8
Weight	8
Valid (boolean for deactivation)	1
Total	**25**

Table 4.1: Hypernode/-edge Memory Consumption in the Static Hypergraph

the index in the array to allow constant-time lookups.

4.4.2 Compression Techniques

Start Index. The start index of each adjacency list is stored uncompressed. Other than reducing the index size to 32 bits—which would limit the maximum graph size—there is little room to reduce the size of the offset arrays. If any encoding were applied to the vector which contains the start offsets, additional computation would be required to decode them, thus increasing the time required to find an adjacency list.

Weights. On the other hand, weights do not have to be stored at all for unweighted instances. However, there is a critical restriction to this: During the partitioning process—specifically the contraction of a graph—weights are required even on unweighted instances. This is because the weight of affected nodes is aggregated during contraction and applied to the contracted node. To remedy this issue, the weight vector is initially empty and does not take up any memory. As long as this is the case, any weight query simply returns 1. As soon as any node or edge is assigned a weight other than 1, the respective array is initialised. This ensures optimal memory efficiency for both node and edge weights.

Degrees and Sizes. Node degrees and edge sizes are stored in the CSR arrays in the beginning of each chunk. This allows both constant time lookups and memory efficiency. Finding the degree (size) of a node (egde) simply requires looking up the start index in the offset array and decoding the first entry. Given that all degrees and sizes are at most a 64-bit integer, this requires decoding at most 9 blocks and thus, is in $O(1)$.

Validity Flag. To keep track of disabled nodes and edges while allowing constant time lookups and updates, two additional arrays are used. While the boolean in the Hypernode and Hyperedge objects requires 8 bits of storage, an efficient bit array can reduce this to a single bit. Consequently, the memory usage of the validity flag is greatly reduced.

4.4.3 Partitioning requirements

The partitioning step introduces additional requirements, such as edge removal and contraction. In the following, the implementation of these requirements is discussed.

Edge Removal and Iteration During graph preprocessing, edges are dropped and restored. In the static graph, dropped hyperedges are not only disabled, but also removed from all the adjacency lists of all nodes they connected. Dropping an edge is trivial; the validity flag is simply set to false. However, removing it from all adjacency lists presents a challenge. Firstly, all affected incident nets have to be decoded. Secondly, the respective edge has to be removed, and the degree must be updated. Then, all entries following the edge ID have to be shifted by one to close the gap. Lastly, the entire list has to be re-encoded. However, this operation creates gaps in the incident nets array and would result in a detrimental runtime if the entire CSR array were to be recreated. Therefore, the gaps are filled with padding bytes.

Two problems arise. First of all, the encoded degree of a given node might underflow upon edge removal. This can be solved by leaving the size the same and simply adding any empty block with the continuation bit set to true followed by the actual value.

Secondly, the padding bytes need to be unambiguously identifiable and recognised by the iterators. Using the value -1 is not possible without sacrificing a bit in each block non-continuation block to store the sign of each value. Using the largest possible integer is not an option, because the varint encoding allows the storage of numbers only limited by the maximal array length. Thus, a trivial "maximum integer" does not exist and would also require indefinite blocks. However, by exploiting the properties of the CSR arrays, 0 can be used as a padding byte.

Since no duplicates are stored, a gap of “0" will never occur. Thus, when an iterator encounters the value 0, it immediately knows that a chunk is over. Notwithstanding this, there a two valid occurrences of the number 0. On the one hand, it can occur as the continuation of a varint that started in a previous block. This is trivially identified by checking if the previous block has the continuation bit set to *true*. If this is not the case, the iterator knows it has encountered a padding block. On the other hand, 0 might simply be indicating the node or edge with ID 0. On first glance, it appears that this can simply be detected by the position of the entry. If 0 is the first entry in a given chunk, it must be indicating a valid ID instead of a padding byte.

Yet, there is one edge case in which this behaviour fails. Say a given node is connected to the hyperedges 3 and 9. Then, both of these edges are removed. The adjacency list of this node is now empty, filled with only padding bytes. However, the iterator identifies the first padding byte as the valid edge ID “0”, since it is the first entry. Hence, anytime the iterator identifies 0 as a potential node (edge) ID, it has to fall back on another property of the node (edge): its degree (size). If the degree (size) is 0, it returns an empty list. If it is 1, it returns 0 as an ID. In any other case (> 1), the iterator would not consider first entry to be a padding byte in the first place.

When an iterator is initialised, it finds the last valid element of a chunk. To optimise the runtime of this operation, the iterator first jumps to the start of the next node (edge). This is possible in constant time as it merely requires a lookup in the offset array. The iterator then decrements its position until it finds a valid entry or reaches the start of the chunk. If no edge has been removed from the graph and there are no gaps, the last element is always valid and the iterator immediately knows where the chunk ends. Thus finding the end of a chunk is in $\mathrm{O}(1)$, as exactly two lookups have to be executed. If edges have been removed—which is only the case during partitioning—the worst case runtime is the max degree of the graph. For edges, this operation is always in $\mathrm{O}(1)$, as the CSR edge array is never altered and the last element in a chunk is always valid.

Proxy Object Access. To retain cross-compatibility with the structure of the static hypergraph, the Hypernode and Hyperedge interfaces are retained, but merely serve as proxy objects. They are not stored in memory unless they are accessed by index. In this case, the information of the weight array, validity array and the CSR array is combined to compute all of the properties of the respective object listed in table 4.4.1 and initialise a temporary proxy object.

In summary, these techniques reduce the memory consumption of each edge and node from **25 Bytes** down to $\approx$ **10 Bytes**[1]. This figure consists of **8 Bytes** for the offset, **1-9 Bytes** for the degree/size and **1 Bit** for the validity flag. It is important to note that all of the improvements except for the compression of the degree can also be trivially implemented in the static hypergraph without any additional overhead.

Contraction. Operating directly on compressed, varint gap–encoded CSR complicates random access and forbids cheap in-place edits. To avoid inflation and fragmentation, the algorithm rebuilds a new compressed hypergraph via streaming passes:

Representation challenges. (i) pins and incident nets are stored as varint headers plus strictly increasing gap sequences, so random access requires short decoding prefixes; (ii) structural edits would fragment arrays; (iii) detecting parallel coarse edges is costly if full pin lists are decoded and materialized.

Strategy. (1) For each enabled fine edge, decode its slice once, map pins to coarse nodes, then sort-unique; drop singletons. (2) Collect lightweight per-edge metadata in parallel (hash, size, 32-bit fingerprint, ID); parallel-sort only a permutation by these keys; deduplicate in one linear sweep, decoding candidates selectively on key matches and aggregating weights into a representative. (3) Compute exact encoded length per coarse edge

[1]This is a conservative estimate which assumes that the node degree and hyperedge size can fit into two varint bytes on average. Across the 488 files of the benchmark set, a [18], the average degree is $\lceil 25 \rceil$ and the average edge size is $\lceil 27 \rceil$. Both of these values fit into a single block.

(varint(|pins|) + gap varints), take a prefix sum, and write each edge directly into its disjoint incidence slice in parallel. (4) Build incident nets in two passes: first count degree and byte budget per node (including varint(degree) header and he-ordered gap costs); prefix-sum to assign disjoint slices; then write headers and gap-encoded incident edges without races.

Parallelism and Determinism. Thread-local buffers remove contention; disjoint slice assignment eliminates write races; an explicit ID tie-breaker in the sort key yields deterministic output. Weights are assigned in a final sequential pass to avoid contention in compact two-level vectors.

The purpose of these measures is to ensure: few linear passes over compressed data, no global uncompressed materialisation, minimal peak memory usage, scalable parallel throughput, and reproducible results.

4.4.4 Construction

The compressed hypergraph is streamed analogously to the stand-alone implementation. Usually, graphs in Mt-KaHyPar are created using a two-part process. In the first step, the entire graph with all of its edges is read into memory. Afterwards, this information is passed to a Factory which creates a graph object dependent on the application (static vs. dynamic, graph vs. hypergraph).

To minimise peak memory usage during the construction process, the compressed hypergraph is streamed directly from disk, meaning it is built while it is being read. As in the un-buffered stand-alone implementation, the compressed adjacency list is immediately created as a gap encoded CSR array. The node incidence lists are created as a NAR, but are also encoded directly exploiting the properties outlined in section 4.3.

CHAPTER 5

Experimental Evaluation

The following experiments assess the general efficacy of different compression techniques on hypergraphs and the effects of introducing compression into Mt-KaHyPar.

5.1 Setup

All Benchmarks hereafter were executed on a machine equipped with a 12th Gen Intel® Core™ i5-1235U processor. This CPU has a single socket with 6 physical cores and supports 12 hardware threads via simultaneous multithreading (2 threads per core). The processor operates at a base frequency of 1.3–4.4 GHz and supports both 32-bit and 64-bit modes with little-endian byte ordering. It provides 39-bit physical and 48-bit virtual addressing. The system is configured with 16 GiB of main memory. The cache hierarchy consists of 288 KiB L1 data cache, 192 KiB L1 instruction cache, 7.5 MiB of L2 cache distributed across cores, and a shared 12 MiB L3 cache. Hardware virtualisation (Intel VT-x) was enabled, with Microsoft Hyper-V providing full virtualisation support. and benchmarks were compiled under the Windows Subsystem for Linux and executed using CMake version 3.28.3.

For experiments involving Mt-KaHyPar, the default preset was used. Graphs were partitioned into 12 blocks using the km1 objective function with an ε of 0.03.

5.2 Gap vs. Gap-Interval Encoding

These results indicate that interval encoding can improve compression on the bipartite representation of hypergraphs, but its effect is noticeably diminished compared to real-world graphs. Surprisingly, we found that there are 54,850,115 intervals of length 3 or greater, but they have a mean length of 5.118. This means that the intervals that do exist are close to the cut-off where interval encoding is applied, explaining both the low number of graphs it affected at all and the lower effectiveness of the compression method.

Table 5.1: Gap Encoding vs. Interval Encoding

	Dependent variable:	
	CompressionRatio	
	(1)	(2)
Interval	0.909^{***}	0.545^{***}
	(0.251)	(0.122)
Constant	2.435^{***}	2.008^{***}
	(0.177)	(0.086)
Observations	144	976
R^2	0.085	0.020
Adjusted R^2	0.078	0.019
Residual Std. Error	1.503 (df = 142)	1.903 (df = 974)
F Statistic	13.148^{***} (df = 1; 142)	20.019^{***} (df = 1; 974)
Note:		$^{*}p<0.1$; $^{**}p<0.05$; $^{***}p<0.01$

5.3 Comparison to KaMinPar

In direct comparison to KaMinPar, our implementation improved the geometric mean compression ratio by 19% and the running time by 77%. The latter is likely because the graphs were converted into their bipartite representation before being parsed by KaMinPar. This essentially doubles the number of elements that have to be read from disk, as each vertex in a hyperedge is added to two adjacency lists. Compression ratio measurements are unaffected by this, as our implementation also uses the memory consumption of the bipartite representation as the baseline.

5.4 Mt-KaHyPar

For the compressed hypergraph data structure, there are three critical objectives: The compression ratio should be as high as possible, the partition time should be as short as possible, and the partition quality should remain unaffected compared to the static hypergraph.

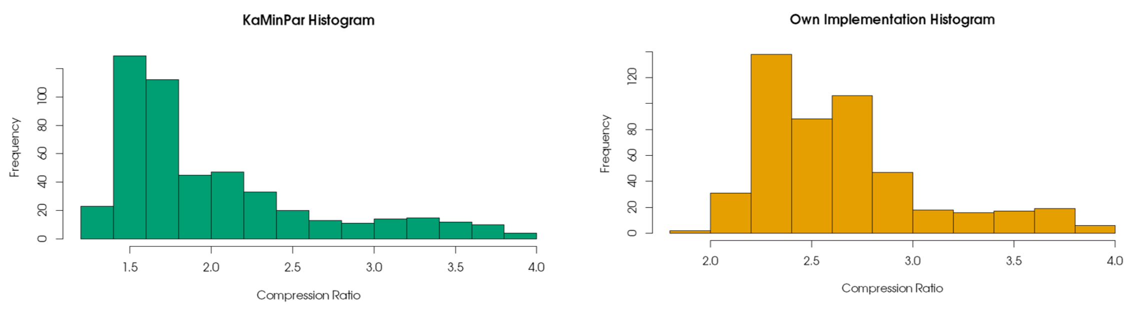

Figure 5.1: Compression Ratio Histograms

To assess these objectives, the same hypergraph files will be read and multilevel-partitioned, first using the static hypergraph and then using the compressed hypergraph. We expect to see a high compression ratio on the data structure. Nevertheless, it is important to point out that many of the applied optimisations can also be trivially implemented in the static hypergraph. Except for the compression of the adjacency arrays, the replacement of the Hypernode and Hyperedge objects with array and proxy object accessors changes very little about the functioning of the data structure. Notwithstanding that, the benchmarks are made against the static hypergraph as-is to gauge a realistic image of the improvements that can be expected due to this paper.

The partition time is expected to increase. While previously, accessing an arbitrary element in any given adjacency list was a constant-time operation, now, the lists have to be encoded first. This is especially significant if there are frequent random accesses within the edges. When the edge is not iterated through from start to finish, but the element at position x is accessed directly, all previous entries have to be decoded as well. This is because the values are delta-encoded. Thus, without decoding and accumulating all values from the start of the list until position x, the value cannot be computed.

Construction Time. The experiments indicate that the streamed construction is faster than the existing construction process that is employed by the static hypergraph. This result is not completely surprising, as the two-step process of loading everything into memory first and then constructing the graph is avoided. However, the current factory maps the file to the virtual address space (mmap), so peak memory during construction is decreased and, in some instances, the construction time is comparable.

Partition Time. On average, partition time increased by 0.436s. This constitutes a slowdown of $\approx 36\%$. This result suggests that the compressed hypergraph should only be used when memory constraints would make it impossible to process a hypergraph, or if the partitioning process becomes slow due to virtual memory usage.

Memory Usage. As expected, the memory usage of the graphs could be significantly reduced. The geometric mean compression ratio across all hypergraphs is ≈ 2.027. There

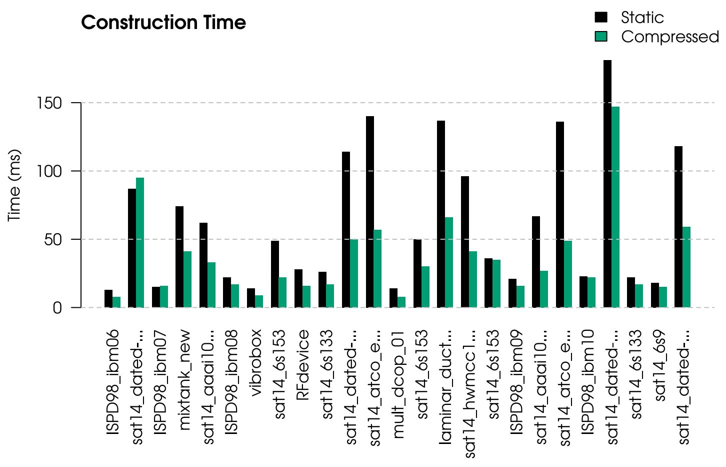

Figure 5.2: Hypergraph Construction Time

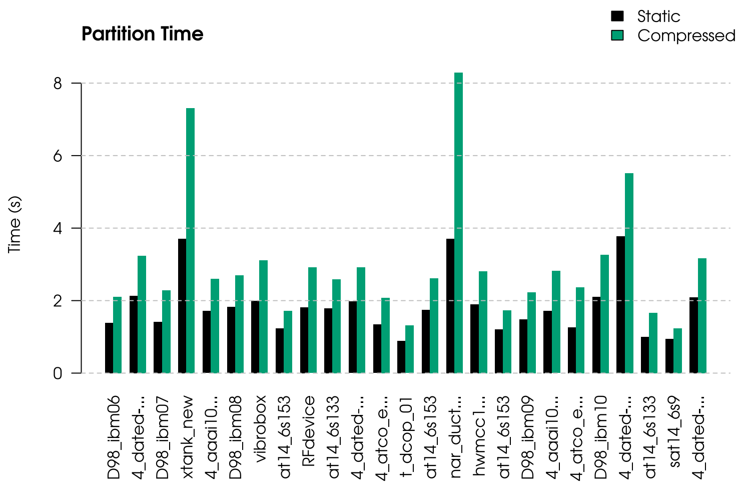

Figure 5.3: Hypergraph Partition Time

are several reasons why the compression ratio fluctuates. Firstly, when IDs in the adjacency lists are closer together, delta encoding works more efficiently. Secondly, the proxy object optimisations become more significant the more sparse the graph is. For example, lazy loading of the weight array is more effective the smaller the average node degree is, as the weight takes up proportionally more of the graph's memory. Table 5.3 summarises the estimated savings by the respective technique.

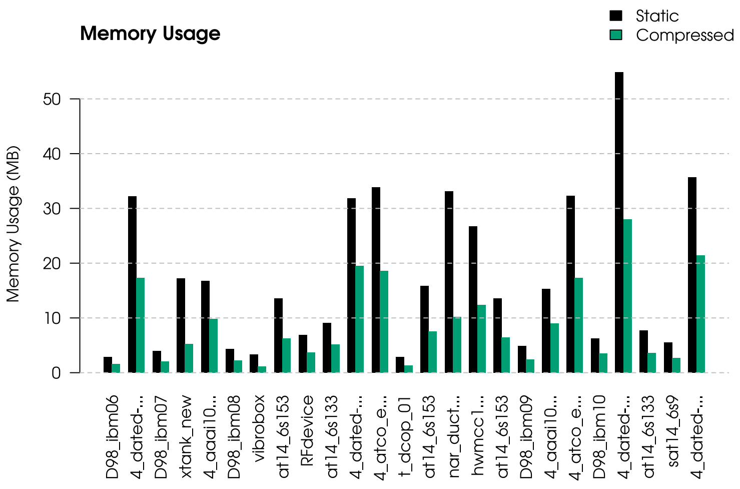

Figure 5.4: Hypergraph Compression Ratio

Table 5.2: Geometric Means of Savings Categories (KB)

Encoding	BitVectors	SizeDegree	Total
$2,273$	195	$1,564$	$7,313$

Peak Memory Usage. Figure 5.4 showcases the reduction in peak memory usage during partitioning. The mean compression ratio is 1.542.

Save for some random variation likely caused by the use of heuristics, the partition quality is identical between the two data structures. Overall, these experiments demonstrate that incorporating compression into Mt-KaHyPar enables substantial memory savings and increases partition time while preserving solution quality. This trade-off makes it a valuable technique for large-scale hypergraph partitioning where available memory is a critical bottleneck. However, due to the significance of the slowdown, it might be worth implementing only the optimisations that are largely performance-neutral, such as bit vectors and two-level weight vectors as used by Salwasser [16]. Table 5.3 summarises the estimated savings by each technique.

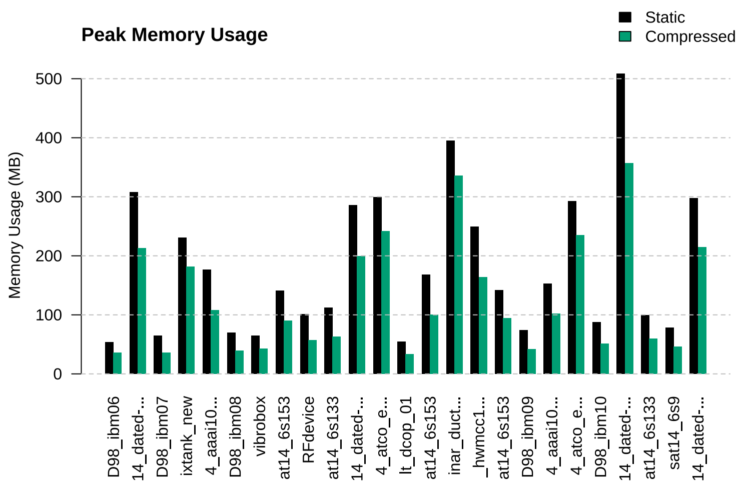

Figure 5.5: Hypergraph Partition Quality

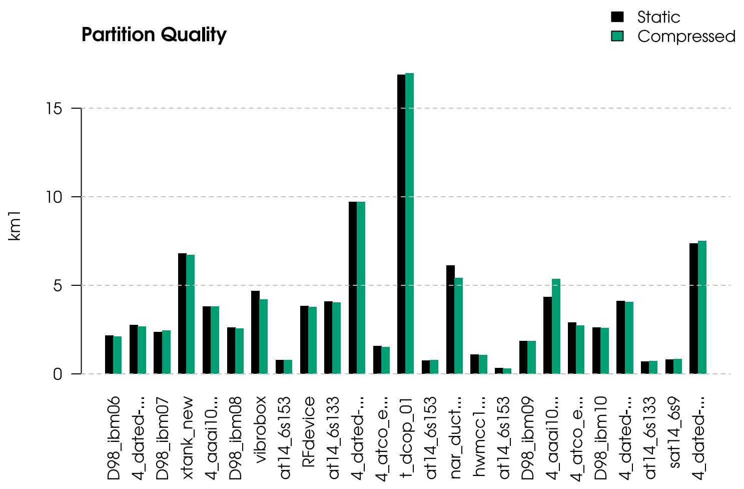

Figure 5.6: Hypergraph Partition Quality

Table 5.3: Geometric Means of Savings

Category	Mean Saved KB
IncidenceSaved	$1,081$
IncidentNetsSaved	$1,216$
WeightsSaved	916
HypernodeValidBitsSaved	80
HyperedgeValidBitsSaved	100
HypernodeDegreeHdrSaved	640
HyperedgePinHdrSaved	799
TotalSavedB	$5,528$

CHAPTER 6

Discussion

6.1 Conclusion

This project developed a strategy for the streamed compression of hypergraphs. Employing established compression methods, hypergraphs are converted into a gap-encoded bipartite CSR adjacency list. During the input process of the graph, properties of the bipartite construction are exploited to reduce peak memory consumption and immediately store vertex arrays in a compressed format.

In Mt-KaHyPar, a new compressed data structure is added. It mimics the existing static hypergraph but greatly reduces memory consumption. Since adjacency array access requires decoding steps, this data structure is partitioned 36% slower on average, but it matches the partition quality of the static hypergraph and achieves a mean compression ratio of ≈ 2.027.

6.2 Future Work

The memory savings in Mt-KaHyPar that the compressed hypergraph enables are accompanied by a significant runtime penalty. Future work might explore optimisations to the compressed data structure that restore runtime performance. This might be accomplished by sacrificing peak memory usage to cache decoded arrays or exploring techniques such as run-markers, SIMD-based Stream VByte or branch-light pointer decoding.

Implementation Details

Stand-Alone Interface

The graph interface provides several methods to interact with the static graph. By request, individual neighbourhoods are decompressed and can either be iterated over or returned as an array.

Table .1: Main Features and Methods of the Compressed Hypergraph Class

Method/Feature	**Description**
Constructor	Reads a metis file from a path
`num_nodes()`	Returns the number of nodes in the graph.
`num_edges()`	Returns the number of edges in the graph. This is useful as variable-length encoding makes it difficult to extract the number of edges in the array.
`num_hyperedges()`	Returns the number of hyperedges in the graph.
`operator[]`	Decompresses the neighbourhood of node `i` and caches the last decompressed neighbourhood for efficiency.
`for_each_neighbor()`	Applies a callback to each neighbour of node `i`, avoiding the allocation of a vector.
`compressed_graph()`	Returns a constant reference to the compressed adjacency list.
`used_memory()`	Estimates the memory usage of the compressed graph structure.

Benchmark Sets

The following table summarises key details about the benchmark set used in during testing. It highlights the number of nodes, hyperedges, the max and average degree (Δ), and max and average edge size (ES).

Hypergraph	Nodes	Hyperedges	Max Δ	Avg Δ	Max RS	Avg ES
ISPD98_ibm06	32498	34826	91	3.94	35	3.68
sat14_dated-10-1...	629461	141860	10	2.27	26	10.08
ISPD98_ibm07	45926	48117	98	3.82	25	3.65
mixtank_new.mtx	29957	29957	154	66.60	154	66.60
sat14_aaai10-pla...	107838	308235	67	6.40	61	2.24
ISPD98_ibm08	51309	50513	1165	3.99	75	4.06
vibrobox.mtx	12328	12328	121	27.81	121	27.81
sat14_6s153.cnf....	85646	245440	2048	6.69	3	2.33
RFdevice.mtx	74104	74104	9	4.93	271	4.93
sat14_6s133.cnf	96430	140968	1456	3.41	3	2.33
sat14_dated-10-1...	141860	629461	26	10.08	10	2.27
sat14_atco_enc2...	112732	526872	517	18.61	11	3.98
mult_dcop_01.mtx	25187	25187	22774	7.67	22767	7.67
sat14_6s153.cnf	171292	245440	1024	3.34	3	2.33
laminar_duct3D.mtx	67173	67173	89	57.06	89	57.06
sat14_hwmcc10-ti...	163622	488120	423	6.96	3	2.33
sat14_6s153.cnf....	245440	85646	3	2.33	2048	6.69
ISPD98_ibm09	53395	60902	173	4.16	39	3.65
sat14_aaai10-pla...	53919	308235	73	12.81	61	2.24
sat14_atco_enc2...	56533	526872	1031	37.10	11	3.98
ISPD98_ibm10	69429	75196	137	4.29	41	3.96
sat14_dated-10-1...	1070757	229544	10	2.31	38	10.77
sat14_6s133.cnf....	140968	48215	3	2.33	2912	6.82
sat14_6s9.cnf.dual	100384	34317	3	2.33	2696	6.83
sat14_dated-10-1...	283720	629461	13	5.04	10	2.27

Table .2: Hypergraph Benchmark Statistics.

Zusammenfassung

Hypergraphen sind eine Verallgemeinerung der Graphdatenstruktur, bei der Kanten eine beliebige Anzahl von Knoten verbinden dürfen, anstatt genau zwei. Dadurch sind sie nicht auf die Modellierung binärer Beziehungen beschränkt und erweitern die Flexibilität von herkömmlichen Graphen. Trotzdem ist Ihre Anwendbarkeit durch Speichergrößen beschränkt. Diese Arbeit entwickelt eine komprimierte Hypergraph-Repräsentation und implementiert diese in der parallelen Partitionierungsbibliothek Mt-KaHyPar. In der eigenständigen Implementierung werden Hypergraphen in einen bipartiten Graphen transformiert, in dem Hyperkanten durch künstlich eingefügte Knoten repräsentiert werden, die dann mit allen Knoten verbunden sind, die sie im Originalgraphen verbunden haben. In Mt-KaHyPar ahmt die Implementierung die bestehende statische Hypergraph-Datenstruktur nach. Der komprimierte Hypergraph wird im Mittel um 36% langsamer partitioniert, weist jedoch eine identische Partitionierungsqualität auf und erreicht ein mittleres Kompressionsverhältnis von circa $2,027$ zur Speicherung des Graphen sowie $1,542$ während der Partitionierung.

Bibliography

[1] Enno Adler, Stefan Böttcher, and Rita Hartel. Compressing hypergraphs using suffix sorting. *arXiv preprint arXiv:2506.05023*, 2025.

[2] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602. ACM, 2004.

[3] Alain Bretto. Hypergraph theory. *An introduction. Mathematical Engineering. Cham: Springer*, 1:209–216, 2013.

[4] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms*, 20(1):9:1–9:54, 2024.

[5] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep multilevel graph partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 48:1–48:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[6] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. In *23rd Workshop on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 16–30. SIAM, 2021.

[7] Lars Gottesbüren, Nikolai Maas, Dominik Rosch, Peter Sanders, and Daniel Seemaier. Linear-time multilevel graph partitioning via edge sparsification, 2025.

[8] Katarzyna Grzesiak-Kopeć, Piotr Oramus, and Maciej Ogorzałek. Hypergraphs and extremal optimization in 3d integrated circuit design automation. *Advanced Engineering Informatics*, 33:491–501, 2017.

[9] ISO/IEC JTC1/SC22/WG21. Working draft — standard for programming language c++ (draft n5008). Web Draft, with HTML and PDF versions, 2025. Latest working draft as of March 15, 2025; includes the standard library definitions (e.g. std::vector).

[10] Abdul Majeed and Ibtisam Rauf. Graph theory: A comprehensive survey about graph theory applications in computer science and social networks. *Inventions*, 5(1):10, 2020.

[11] Sepideh Maleki, Udit Agarwal, Martin Burtscher, and Keshav Pingali. Bipart: a parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–174, 2021.

[12] MPI für Mathematik in den Naturwissenschaften. A hypergraph with six hyperedges. `https://www.mis.mpg.de/de/research/spectral-hypergraph-theory`, 2025. Accessed: 2025-09-26.

[13] Aditya Mukherjee, Akshat Sachan, and P Madhavan. Graph based navigation system. In *AIP Conference Proceedings*, volume 3075, page 020163. AIP Publishing LLC, 2024.

[14] Masataka Okabe and Kei Ito. Color universal design (cud): How to make figures and presentations that are friendly to colorblind people. `https://jfly.uni-koeln.de/color/#pallet`, 2008. Original work published 2002.

[15] Anandhan Prathik, K Uma, and J Anuradha. An overview of application of graph theory. *International Journal of ChemTech Research*, 9(2):242–248, 2016.

[16] Daniel Salwasser. Optimizing a Parallel Graph Partitioner for Memory Efficiency. 2024. Publisher: Karlsruhe Institute of Technology.

[17] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning, 2011.

[18] Sebastian Schlag and Tobias Heuer. Hypergraph benchmark set for "improving coarsening schemes for hypergraph partitioning by exploiting community structure", May 2025.

[19] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning, 2021.

[20] Julian Shun. Practical parallel hypergraph algorithms. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, pages 232–249. ACM, 2020.

[21] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[22] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 293–302. IEEE, 2016.

[23] William Thomas Tutte. *Graph theory*, volume 21. Cambridge university press, 2001.