Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications

Pál András Papp
pal.andras.papp@huawei.com
Huawei Zurich Research Center
Zurich, Switzerland

Georg Anegg
ganegg@ethz.ch
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

Albert-Jan N. Yzelman
albertjan.yzelman@huawei.com
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

ABSTRACT

We study the balanced $k$-way hypergraph partitioning problem, with a special focus on its practical applications to manycore scheduling. Given a hypergraph on $n$ nodes, our goal is to partition the node set into $k$ parts of size at most $(1 + \varepsilon) \cdot \frac{n}{k}$ each, while minimizing the cost of the partitioning, defined as the number of cut hyperedges, possibly also weighted by the number of partitions they intersect. We show that this problem cannot be approximated to within a $\frac{n}{k}^{1/\text{polyloglog} n}$ factor of the optimal solution in polynomial time if the Exponential Time Hypothesis holds, even for hypergraphs of maximal degree 2. We also study the hardness of the partitioning problem from a parameterized complexity perspective, and in the more general case when we have multiple balance constraints.

Furthermore, we consider two extensions of the partitioning problem that are motivated from practical considerations. Firstly, we introduce the concept of hyperDAGs to model precedence-constrained computations as hypergraphs, and we analyze the adaptation of the balanced partitioning problem to this case. Secondly, we study the hierarchical partitioning problem to model hierarchical NUMA (non-uniform memory access) effects in modern computer architectures, and we show that ignoring this hierarchical aspect of the communication cost can yield significantly weaker solutions.

CCS CONCEPTS

• Theory of computation → Approximation algorithms analysis; Parallel computing models; Problems, reductions and completeness.

KEYWORDS

Hypergraph; HyperDAG; Balanced partitioning; Parallel computing; Approximation; Hierarchical NUMA

ACM Reference Format:

Pál András Papp, Georg Anegg, and Albert-Jan N. Yzelman. 2023. Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications. In Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA ’23), June 17–19, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591087

1 INTRODUCTION

One of the most fundamental graph problems is to partition the node set of a graph into $k$ parts of similar size, while minimizing the number of cut edges. Recently, the focus of this balanced $k$-way partitioning problem has shifted from graphs to hypergraphs, where a hyperedge can contain not only two, but an arbitrary number of nodes.

A prominent application of this problem is finding the most efficient way to execute a complex computation in a parallel manner on $k$ processors. That is, we can use each node $v$ of a hypergraph to represent a specific step of a computation (e.g. a function call, or in a more fine-grained model, a single operation). A hyperedge $e$, on the other hand, represents a unit of data (e.g. an input or output variable) in this computation that is shared by a given subset of nodes; hence it would be desirable to execute the nodes of $e$ on the same processor in order to avoid data movement. In such a setting, the partitioning of the nodes into $k$ parts can be interpreted as an assignment of the computational steps to the $k$ available processors, and the balance constraint on the size of the parts ensures that the workload is indeed sufficiently parallelized. On the other hand, our objective is to cut as few hyperedges as possible, which corresponds to minimizing the total communication cost between the processors; this is indeed known to be the bottleneck in many real-world computations.

The main advantage of this hypergraph model is that it allows us to capture the communication cost accurately: if the hyperedge $e$ (representing the value of some variable) intersects $\lambda_e$ out of the $k$ parts, then it takes exactly $(\lambda_e - 1)$ data transfer operations to move this value from one of these processors (where it is initially stored) to all others (where it is needed). In contrast to this, if we try to model this connection between the same subset of nodes $e$ as a simple graph, then this will always result in an over- or underestimation of the real cost in some cases [25, 26].

In this paper, we present new hardness results for the balanced hypergraph partitioning problem, extending a $n^{1/\text{polyloglog} n}$-factor inapproximability bound which was only known for the similar bisection problem before. More importantly, we show that this hardness result already holds for hypergraphs of very small degree, thus pointing out the crucial role of heuristics in practice.

Besides this general hardness result, we also study two further aspects of the partitioning problem which are both strongly motivated by practical considerations. Firstly, we study the setting where we
also have dependencies (precedence relations) between the different computational steps in our hypergraph, e.g. when modelling the steps of an entire algorithm. We introduce the notion of hyperDAGs to capture the communication cost of a parallel execution in this case: this essentially combines the concept of computational DAGs with the more accurate hyperedge-based modeling of communication costs. We study the partitioning problem on hyperDAGs specifically.

Finally, we also define the hierarchical variant of the hypergraph partitioning problem. This deals with an oversimplification from our original model, namely, that the communication cost is assumed to be uniform between any pair of processors. The majority of modern computing architectures, however, are organized into a hierarchical tree structure: several cores are connected to the same CPU, several CPUs to the same RAM, and then we possibly have multiple such units connected on a network level. Due to this, these architectures exhibit highly non-uniform communication costs between different pairs of processing units: transferring data between two cores on the same processor only induces a small cost, whereas transferring it through multiple levels of the hierarchy is much more time-consuming. As such, for a realistic model of communication cost, it is essential to also incorporate this hierarchical structure into the partitioning problem.

Our main contributions are as follows:

- As our main result, we show that assuming the Exponential Time Hypothesis (ETH), there is no polynomial-time approximation algorithm of factor less than \( n^{1/\text{polyloglog} n} \) to the \( \epsilon \)-balanced hypergraph partitioning problem for any \( k \geq 2 \) or \( \epsilon \geq 0 \). Furthermore, this hardness result already applies for practically relevant cases: it already holds if our inputs are restricted to hyperDAGs of node degree at most 2.
- We define and analyze a special class of hypergraphs (hyperDAGs) that provide a more accurate model of capturing I/O cost in computations modelled by DAGs. We then study two natural techniques to develop more appropriate balance constraints for hyperDAGs. In case of layer-wise constraints, we show that the best solution cannot be approximated to an \( n^{1-\delta} \) factor (for any \( \delta > 0 \)). In case of schedule-based constraints, our observations show that a precise measurement of parallelization in hyperDAGs is not viable in practice.
- We discuss several hardness results for the natural extension of the partitioning problem where we have multiple independent balance constraints.
- Finally, we introduce the hierarchical partitioning problem to obtain a significantly more accurate model of I/O cost in today’s computing architectures. We show that ignoring this hierarchical aspect of the cost function can result in significantly weaker solutions.

2 RELATED WORK

Both the graph- and hypergraph partitioning problem is known to be NP-hard already for \( k = 2 \), and for any (non-trivial) \( \epsilon \geq 0 \) [22]. The problems have a wide range of applications, including parallel computing, VLSI design, and scientific computing [35].

There is a long line of work on approximation algorithms for the case of \( \epsilon = 0 \) and \( k = 2 \), also known as the bisection problem, culminating in an \( O(\log n) \)-approximation by Räcke [16, 30, 40]. The variant of the problem without a balance constraint has also been studied [23, 43], as well as lower bounds for the case when \( k \) is a variable part of the input [2, 17].

Many further works on approximating the partitioning problem have focused on \((\alpha, \beta)\)-bi-criteria approximations of the bisection problem [4, 17, 31, 32], where the cost is at most \( \alpha \) times that of the optimal bisection, and every partition has at most \( \beta \cdot \frac{n}{k} \) nodes (i.e. the strict balance constraint can be violated by a factor \( \beta \)). However, this is a significantly different concept from approximating the \( \epsilon \)-balanced partitioning problem, because bi-criteria approximations compare each solution only to the optimal bisection cost, and this optimal bisection cost can be a factor \( \Theta(n) \) larger than the optimum for \( \epsilon \)-balanced partitioning. As such, in applications where our goal is to find an \( \epsilon \)-balanced solution of low cost, the guarantees of the bi-criteria approach might not be meaningful: even if a low-cost \( \epsilon \)-balanced solution exists, the bi-criteria approximations may return a solution that only approximates the (possibly much higher) optimal bisection cost.

A hierarchical version of the partitioning problem has also been studied on simple graphs, mainly through similar bi-criteria approximations where \( \beta \) also depends on the height of the hierarchy [24, 42].

In recent years, the attention in partitioning problems has shifted to hypergraphs. The case of hypergraph partitioning without a balance constraint has been analyzed [9, 10]. As for the constrained case, the work of Räcke, Schwartz and Stotz [41] again focuses on the bisection problem: they present an approximation algorithm of factor \( O(\sqrt{n}) \), as well as several lower bounds for approximability, and they also show that tree-based methods (which provide some of the best approximations for graph partitioning) are not viable for hypergraphs.

The closest result to our main theorem (also from [41]) is a similar inapproximability bound of \( n^{1/\text{polyloglog} n} \) for the bisection problem. Our result is different from this in two ways. First, we show this bound for the balanced partitioning problem with \( \epsilon > 0 \), which is an easier problem than bisection: there is a simple reduction from balanced partitioning to bisection via adding isolated nodes (see the full version of the paper [36]), but the other direction is not straightforward. Second, we show that the bound already holds in hypergraphs (or even hyperDAGs) of very small degree.

We note that some of these related works consider the natural extension of the problem with node or edge weights; our hardness results also carry over to these more general settings.

We also point out that a similar (slightly more general) notion to hyperDAGs has already been discussed in the work of [39], noting that it provides a more accurate model of communication cost in computational DAGs; however, this work does not study the topic (either hyperDAGs or the partitioning problem) from a theoretical perspective.

Finally, due to the wide applicability of hypergraph partitioning, finding efficient and scalable solutions in practice has also been an active area of research for a long time. This includes sophisticated heuristics [7, 8, 27, 44, 46] as well as optimized exact algorithms [29, 38].
3 PRELIMINARIES

3.1 Hypergraphs and partitioning

A hypergraph $G(V, E)$ consists of a set of nodes $V$ and hyperedges $E \subseteq 2^V$, where $2^V$ denotes the power set of $V$. We denote the number of nodes by $n := |V|$, the total number of pins by $\rho := \sum_{e \in E} |e|$, and the maximal node degree by $\Delta := \max_{v \in V} \{|e \in E \mid v \in e\}$. We also use $|\ell|$ as shorthand notation for the set of integers $\{1, \ldots, \ell\}$.

A $k$-way partitioning $P$ of $G$ is a disjoint partitioning $P_1, \ldots, P_k$ of the nodes $V$. For a hyperedge $e \in E$, we define $\lambda_e := |\{i \in [k] \mid e \cap P_i \neq \emptyset\}|$ as the number of partitions intersecting $e$, and we say that $e$ is cut if $\lambda_e > 1$. There are two popular cost metrics for a $k$-way partitioning $P$: the cut-net metric $\{|e \in E \mid \lambda_e > 1\}$, and the connectivity metric $\sum_{e \in E} (\lambda_e - 1)$. Our hardness results apply to both of these cost metrics (unless one of the metrics is explicitly specified). Note that for the simplest case of $k = 2$, the two metrics are identical. In this case, we will also refer to the nodes in $P_1$ and $P_2$ as red and blue nodes for simplicity.

Given a balance constraint parameter $\epsilon > 0$, we say that $P$ is $\epsilon$-balanced if for all $i \in [k]$ we have $|P_i| \leq (1+\epsilon) \cdot \frac{\rho}{k}$. For convenience, this is sometimes relaxed to $|P_i| \leq \left\lceil (1+\epsilon) \cdot \frac{\rho}{k} \right\rceil$ to ensure that a balanced $P$ always exists. We also implicitly assume $\epsilon < k - 1$, i.e. the balance constraint ensures that no part $P_i$ can contain the entire set $V$.

Definition 3.1. In the $\epsilon$-balanced $k$-way hypergraph partitioning problem (or simply partitioning problem), we have an input hypergraph $G(V, E)$, and our goal is to find an $\epsilon$-balanced partitioning of $V$ with minimal cost (with respect to either the cut-net or the connectivity metric). In the decision version of the problem, the input also contains an $L \in \mathbb{Z}$, and we need to decide if there is an $\epsilon$-balanced partitioning of cost at most $L$.

Note that both $k \geq 2$ and $\epsilon \geq 0$ are fixed constants, i.e. parameters of the problem and not part of the input. The special case of $k = 2$, $\epsilon = 0$ is known as the bisection problem.

Some of our hardness results are built on different complexity assumptions (we discuss these in the full version in more detail). Most important among these is the Exponential Time Hypothesis (ETH); intuitively, this states that $3$-SAT cannot be solved in subexponential time. We also occasionally use stronger variants of this hypothesis (such as SETH or Gap-ETH).

We also assume some familiarity with the parameterized complexity classes $\text{W}[1]$, XP, and para-NP [18]. Intuitively, a problem with some parameter $L$ is in XP if it can be solved in $n^f(L)$ time; it is para-NP-hard if it is already NP-hard for a fixed $L \in O(1)$.

3.2 HyperDAGs

General hypergraphs are indeed the appropriate way to model large computations when we can execute the computational steps in arbitrary order; for example, they are often used to model the parallelization of large SpMV (sparse matrix-dense vector) multiplications [29].

However, in other cases, our goal is e.g. to model the steps of a complex algorithm, where we also have dependencies between the different computational steps; as such, we clearly cannot execute them in any desired order. Such a setting can be modelled as a directed acyclic graph (DAG), where the nodes again represent specific computational steps (intermediate values to compute), and the directed edges represent precedence relations between these computations: the edge $(u, v)$ implies that the output value of computation $u$ is an input to computation $v$. This computational DAG model has been studied extensively in terms of scheduling, time-memory trade-off and many other perspectives [12, 13, 34].

Note, however, that if we directly apply computational DAGs to capture communication costs, we face the same problem as in simple graphs: the number of cut edges does not directly describe the number of values transferred between processors, and as such, it can provide a very inaccurate metric for the cost. In the extreme case, it can happen that a red node $u$ has an edge to $(n - 1)$ distinct blue successors; while this implies $(n - 1)$ cut edges between the two parts, in reality, we only need to transfer a single value once: the output of computation $u$ from the red to the blue processor. As such, to obtain an accurate model of communicate costs in computational DAGs, we introduce the notion of hyperDAGs.

Definition 3.2. For a given a computational DAG $G(V, E)$, the corresponding hyperDAG $G'(V', E')$ is defined by $V' := V$ and 

$$E' := \{\{u\} \cup S_u \mid u \in V\},$$

where $S_u := \{v \in V \mid (u, v) \in E\}$ is the set of immediate successors of $u \in V$.

Given this hypergraph representation of the computational DAG (illustrated in Figure 1), the hypergraph partitioning problem now provides the correct metric for the communication cost: if an intermediate value $u$ (represented by hyperedge $e = \{u\} \cup S_u$) is computed and stored on some processor $p_u \in [k]$, then we need $(\lambda_e - 1)$ transfer operations to make this value available for all the
other processors that compute a successor of \( u \). As such, using hyperDAGs instead of computational DAGs allows us to also capture the communication cost correctly when modelling a computation with precedence constraints.

We point out that many works on computational DAGs assume that the indegrees of nodes are bounded by a small constant \([1, 6]\). This directly translates to a small constant degree \( \Delta \langle \text{computational} \rangle \text{DAG} \). On the other hand, if we are only given a \( \Delta \), this directly translates to a small constant degree \( \Delta \langle \text{hyperDAG} \rangle \). Therefore, if the indegrees of nodes are bounded by a small constant \([1, 6]\), then our hyperDAG will have \( \Delta \leq 3 \). As such, hyperDAGs with a small constant degree are of particular interest in practice.

If we have a description of our hyperDAG that specifies for each hyperedge the node from which it was generated, then one can easily verify whether this hyperDAG corresponds to a valid computational DAG. On the other hand, if we are only given a general hypergraph \( G \) (without the generator nodes specified), it is not trivial to decide whether \( G \) is actually a hyperDAG, i.e. if it corresponds to the hyperDAG representation of some DAG. For example, the triangle in Figure 2 is a simple hypergraph that cannot be obtained as a hyperDAG from any original DAG: for instance, it does not have a node of degree 1 that could correspond to a source of the original DAG.

This shows that hyperDAGs are only a subclass of general hypergraphs, and hence understanding their properties is an important question: hyperDAGs could have some structural properties that make the partitioning problem easier on them. As such, even though this is not closely related to our main focus, we also provide a brief analysis of the fundamental properties of hyperDAGs in the full version of the paper [36].

- Firstly, we develop a complete characterization of hyperDAGs: we show that a hyperDAG is a hypergraph if and only if a specific property holds for all of its subgraphs.
- Using this characterization, we then also show that it can be decided in linear time whether a given hypergraph is a hyperDAG.
- Finally, for the sake of completeness, we prove that the partitioning problem still remains NP-hard if restricted to hyperDAG inputs.

4 INAPPROXIMABILITY RESULT

In this section we discuss our main theorem, which extends the previously known hardness result from the bisection case to the partitioning problem for general \( \epsilon \geq 0 \). More importantly, we also show that this hardness result already holds in heavily restricted cases (hyperDAGs of degree at most 2), suggesting that the problem is not even approximable in practically relevant settings.

**Theorem 4.1.** Assuming ETH, it is not possible to approximate the optimum of the partitioning problem to an \( n^{1/\log \log n} \)-factor in polynomial time (for some constant \( \delta > 0 \)). This holds for any \( k \geq 2 \) and \( \epsilon \geq 0 \), even if the input is restricted to hyperDAGs with \( \Delta = 2 \).

**Proof (sketch).** We use a reduction from the well-studied Smallest \( p \)-Edge Subgraph (SpES) problem: given a graph \( G(V, E) \), we need to find a subset \( V_0 \subseteq V \) such that the subgraph induced by \( V_0 \) has at least \( p \) edges, and \( |V_0| \) is minimized. It is known that if ETH holds, then there is a \( \delta > 0 \) such that no polynomial-time \( n^{1/\log \log n} \)-factor approximation exists to this problem [33].

Given an instance of the SpES problem, the main idea is to convert it into a hypergraph that mostly consists of blocks: groups of nodes which are so densely interconnected by hyperedges that they all need to receive the same color, otherwise we end up with an unreasonably high cost. These blocks are also used as a fundamental ingredient in several other constructions throughout the paper. In our current construction, we begin by creating two very large blocks \( A, A' \), and enforcing (through the balance constraint) that they must obtain different colors; let us assume w.l.o.g. that \( A \) is colored blue, and \( A' \) is colored red. We also create a smaller block for each \( v \in E \), and we carefully select the size of blocks such that at least \( p \) of these edge blocks must be colored red in order to satisfy the balance constraint. Finally, for each \( v \in V \), we create a hyperedge which contains (i) a node from the block of every \( e \) that is incident to \( v \), and (ii) a further node that is forced to be blue (due to further hyperedges connecting it to \( A \)). The construction is illustrated in Figure 3.

In the resulting hypergraph, we need to select a subset of (at least) \( p \) edge gadgets that we color red. However, if a node \( v \in V \) is incident to any of these \( p \) edges, then the hyperedge corresponding to \( v \) will be cut, since it contains both a red and a blue node. Altogether, the cost of a solution will be exactly the number of nodes covered by the \( p \) chosen (red) edges of \( G \); as such, approximating the minimum cost would also allow us to approximate the SpES problem to the same factor.

The more technical part is to extend the reduction first to hypergraphs with \( \Delta = 2 \), and then to hyperDAGs. For the extension to \( \Delta = 2 \), we essentially replace all the blocks by “grid gadgets”: these are gadgets which are 2-regular (each node has degree 2), but they still ensure that cutting off a significant portion of the nodes from the gadget induces an unacceptably high cost. We then discuss how to connect these grid gadgets to each other in a way such that it essentially exhibits the same properties as the original construction with blocks. Finally, to convert the construction into a hyperDAG, we add further auxiliary nodes that cannot affect the optimal partitioning, and then we show that there is an injective assignment from hyperedges to generating nodes, i.e. there indeed exists a computational DAG that corresponds to this hypergraph. \( \square \)

We note that our hardness results also carry over to the special class of 2-regular hypergraphs recently studied by [29] for modelling SpMV problems.
Furthermore, note that Theorem 4.1 was expressed in terms of ETH, which is a rather standard complexity assumption. There are several stronger inapproximability results for SpES based on less standard assumptions; these also provide stronger hardness results for the partition problem.

**Corollary 4.2.** Our reduction method also shows the inapproximability of the partitioning problem to the following factors based on stronger complexity conjectures:

- \( n^{f(n)} \) for any function \( f(n) = o(1) \), if Gap-ETH holds [33],
- \( n^\delta \) for a given \( \delta > 0 \), if specific one-way functions exist [3],
- \( n^{\frac{1}{2} - \delta} \) for any \( \delta > 0 \), if the Hypergraph Dense vs. Random Conjecture holds [11].

Besides approximation algorithms, it is also interesting to study the hardness of the problem from a parameterized complexity perspective, in terms of the allowed cost \( L \). It follows easily from the W[1]-hardness of SpES that the partitioning problem is also W[1]-hard. On the other hand, one can show that the problem is in XP, i.e., it can be solved in time \( n^{f(L)} \) for some function \( f \). Intuitively, the main idea is to try all possible combinations of cut hyperedges that can result in a total cost of at most \( L \); this means that at most \( L \) hyperedges are cut, so there are only \( n^{f(L)} \) such cases. Then in each of these cases, we can essentially remove these cut hyperedges (converting them into constraints) to obtain a delicate packing problem that can be solved by a dynamic programming approach.

**Lemma 4.3.** In terms of the allowed cost \( L \) as a parameter, the partitioning problem is \( W[1] \)-hard (already for hyperDAGs of degree \( \leq 2 \)), but it is in XP.

## 5 BALANCE CONSTRAINTS FOR HYPERDAGS

If our hypergraph models an application where the computational steps have no inter-dependence, i.e., they can be executed in any order, then the balance constraint already ensures that the computational workload on the \( k \) processors is evenly distributed. However, if we have a hyperDAG which was obtained from a computational DAG with precedence constraints, then a simple balance constraint may fail to ensure any amount of parallel execution.

Indeed, if for example our DAG \( G \) is a serial concatenation of two DAGs \( G_1 \) and \( G_2 \) of the same size (as sketched in Figure 4), then an assignment where \( G_1 \) and \( G_2 \) are assigned to the red and blue processors, respectively, is perfectly balanced. Yet, the blue processor will need to wait for all the computations on the red processor to finish before it can begin the computation of any blue nodes at all. Thus even though this partitioning satisfies the balance constraint, we are in fact unable to parallelize the workload between the red and blue processors at all. This suggests that in order to ensure parallel execution in hyperDAGs, we require a more refined approach for our balance constraint.

### 5.1 Layer-wise constraints

One natural idea is to divide a given algorithm into "phases", and ensure that the workload is balanced in each phase separately. In the simplest case, this corresponds to dividing the nodes of a hyperDAG into layers, i.e., disjoint sets \( V_1, \ldots, V_p \) such that \( \ell \) is the length of the longest path in the DAG, and for each directed edge \((u, v)\) with \( u \in V_i, v \in V_j \), we have \( i < j \). We illustrate such a layering in Figure 5. We can then define a layer-wise version of the partitioning problem where the balance constraint needs to hold in each layer separately (while our goal is still to minimize the cost of the cut as before).
we first need to select a valid layering of the DAG as discussed
in the DAG, and then these are discussed in detail in the full version of the paper. The
main ideas behind the proof are as follows:

Definition 5.1. In the layer-wise balanced hyperDAG partitioning problem, a partitioning is only feasible if each layer is balanced, i.e. if for all \( j \in [\ell], i \in [k] \), we have \( |P_i \cap V_j| \leq (1 + \epsilon) \cdot \frac{|V_j|}{k} \).

In the simplest case, we can create layers by sorting every node into the earliest possible layer: \( V_1 \) contains the source nodes of the DAG, and then \( V_i \) (for \( i \geq 2 \)) is the set of nodes that have all their predecessors contained in \( \bigcup_{j=1}^{i-1} V_j \); this indeed divides the DAG into layers \( V_1, \ldots, V_\ell \). However, in general, there are multiple different ways to divide the DAG into layers; for example, in the DAG in Figure 5, the lowermost node can be sorted either into layer \( V_2, V_3 \) or \( V_4 \). Hence we can also define a more general, flexible layering version of the partitioning problem, where our goal is twofold: we first need to select a valid layering of the DAG as discussed above, and we then need to find a layer-wise balanced partitioning according to these layers, with the final goal of minimizing the cost.

With these layer-wise constraints, the partitioning problem turns out to be even harder: it is already NP-hard to distinguish between an optimal cost of 0 and \( n^{1-\delta} \) (for any \( \delta > 0 \)).

Theorem 5.2. It is NP-hard to approximate the layer-wise balanced partitioning problem to any finite factor, both in the fixed and in the flexible layering case.

Proof (sketch). The proof consists of several technical steps; these are discussed in detail in the full version of the paper. The main ideas behind the proof are as follows:

- Our DAG construction consists of several connected components, each having a carefully designed number of nodes in each layer. In order to obtain a partitioning of cost 0, all of these components need to be monochromatic.
- Besides the main components, we also add \( k \) "control components", and use auxiliary layers at the end of the DAG to ensure that these all receive different colors. These control components are then used to add a desired number of nodes of fixed colors to any layer, and hence, intuitively, to ensure that specific layers must contain at least/at most a specific number of nodes of given colors.
- At the core of our construction, there is a reduction from the well-known graph coloring problem, using the above tools to convert the coloring problem to this multi-constraint partitioning setting.

In the resulting DAG, a layer-wise partitioning of cost 0 exists if and only if the original graph has a valid 3-coloring. Moreover, our DAG is designed to allow only one possible layering, so this settles the proof for both the fixed and the flexible layering case. \( \square \)

A slightly different version of this proof also shows that in the flexible layering case, this hardness result already applies separately to the subproblem of finding the best layering of the DAG, i.e. the layering where the optimum cost is smallest. In other words, even if we have an oracle that returns the optimal partitioning for a specific fixed layering of the DAG, the optimum is still not approximable to any finite factor.

5.2 Schedule-based constraints

We have seen that layer-wise constraints (instead of a single constraint) allow us to exclude solutions where computations in a hyperDAG are in fact not parallelized. Unfortunately, the layer-wise approach can run into a different problem: it may impose a condition that is too strict, also excluding some solutions that are in fact perfectly parallelized.

In particular, consider a DAG with two distinct paths of length 3 from a single source node to a single sink node, and then let us split both the first node in the upper path and the second node in the lower path into a larger set of \( b \) nodes, as sketched in Figure 6. With layer-wise constraints (and a sufficiently small \( \epsilon \)), we are forced to partition both of these sets in an (almost) balanced way, since they contain almost all the nodes in the given layer. Hence whichever color we choose for the successor of these sets, we will have a cost of \( \Theta(b) \). In contrast to this, if we were to simply color the upper branch red and the lower branch blue (and the source and sink node with an arbitrarily chosen color), then we have near-perfect parallelization, and a cut cost of only 2 altogether.

In general, the only straightforward way to develop an exact metric of parallelization is to consider a concrete scheduling of the DAG, which assigns the nodes not only to processors, but also to time steps. A detailed discussion of scheduling problems is beyond the scope of this paper; in the rest of the section, we briefly show how scheduling can be used to develop a more accurate balance constraint for partitioning, and we also discuss the limits of this approach.
Definition 5.3. Given a DAG and a fixed constant $k$, a scheduling is an assignment of the nodes to processors $p : V \to [k]$ and to time steps $t : V \to \mathbb{Z}^+$ such that
- for all $u, v \in V$ we have either $p(u) \neq p(v)$ or $t(u) \neq t(v)$ (the scheduling is correct),
- for all $(u, v) \in E$, we have $t(u) < t(v)$ (the precedence constraints are satisfied).

The goal is then to minimize the makespan $\max_{v \in V} t(v)$ of the scheduling, i.e. to execute the computations as fast as possible (without considering communication costs).

When compared to $n$, this optimal makespan essentially allows us to measure the parallelizability of the DAG. For example, if our DAG is simply a directed path, then the best makespan is $n$ (the DAG is not parallelizable at all); on the other hand, if it consists of $k$ disjoint DAG components of equal size, then the best makespan is $\frac{n}{k}$ (the DAG is perfectly parallelizable). As such, it is a natural idea to use this metric to define a more sophisticated, schedule-based balance constraint on our hyperDAGs, where a given partitioning is feasible if it can be relatively well parallelized compared to the best possible parallelization of the DAG.

More formally, given a DAG, let $\mu$ denote the minimal makespan in general (i.e. over all $p', t'$ such that $(p', t')$ is a valid schedule), and let $\mu_p$ denote the minimal makespan for a fixed partitioning $p$ (i.e. over all $t'$ such that $(p, t')$ is a valid schedule).

Definition 5.4. In a schedule-based balance constraint, we say that a partitioning $p : V \to [k]$ is feasible if $\mu_p \leq (1 + \epsilon) \cdot \mu$.

This schedule-based constraint provides a much more sophisticated condition of sufficient parallelization in hyperDAGs. On the other hand, the approach has strong limitations in practice. In particular, the DAG scheduling problem (computing $\mu$) is not known to be polynomially solvable except for a few special cases, such as for $k = 2$, or for special classes of DAGs such as out-trees, level-order DAGs or bounded-height DAGs [12, 14, 15, 19, 21, 45]. Moreover, maybe more surprisingly, we show that evaluating the quality of a given partitioning (computing $\mu_p$) is an even harder problem which remains NP-hard even in these very special cases.

Theorem 5.5. Computing $\mu_p$ is already NP-hard for $k = 2$, even if the inputs are restricted to out-trees, level-order DAGs or bounded-height DAGs.

This provides an unusual situation in these special cases: we can efficiently compute the parallelizability of the DAG in general, but we cannot compute how parallelizable our own solution is, and hence we cannot verify if it satisfies a schedule-based balance constraint. This suggests that such a schedule-based constraint is not a viable approach in practice, even for the simplest case of $k = 2$.

6 MULTI-CONSTRAINT PARTITIONING

The layer-wise balance constraints in our hyperDAGs are in fact a special case of a natural generalization of the partitioning problem, where instead of having only a single balance constraint for the whole set $V$, we have separate balance constraints for smaller subsets of nodes.

This problem might be of independent interest in several applications. For instance, as another (more practical) approach to ensure sufficient parallelization in hyperDAGs, one might decide to heuristically decompose the hyperDAG into relatively independent “regions” (preferably larger than layers but smaller than the entire graph, such as e.g. the sets $G_1$ and $G_2$ in Figure 4), and enforce a balance constraint on each region separately. To our knowledge, similar multi-constraint problems have only been studied on simple graphs [28] or in particular applications [37] before.

Definition 6.1. In the multi-constraint partitioning problem, our input also contains disjoint subsets $V_1, \ldots, V_c \subseteq V$. We say that a partitioning $P = \{P_1, \ldots, P_k\}$ of $V$ is feasible if it satisfies the balance constraint for all subsets, i.e. for all $j \in [c], i \in [k]$, we have $|P_i \cap V_j| \leq (1 + \epsilon) \cdot \frac{|V_j|}{k}$.

The simplest cases to analyze in terms of hardness are the two extremes of $c$. That is, when we only have $c = O(1)$ constraints, then the problem still remains in XP, and a simple reduction allows us to carry over some of the known approximation algorithms for standard partitioning to this multi-constraint case, although in a significantly weaker form. On the other hand, when we have $c \geq n^\delta$ for some constant $\delta > 0$, the problem becomes significantly harder both in terms of approximability and parameterized complexity.

Lemma 6.2. For $c \in O(1)$ constraints, there exists a reduction from multi-constraint bisection to the standard bisection problem, and the partitioning problem is still in XP (with respect to $L$).

Lemma 6.3. If we have $c \geq n^\delta$ constraints for some constant $\delta > 0$, then no polynomial-time approximation exists for the partitioning problem to any finite factor, and the problem is para-NP-hard (with respect to $L$).

Between the two cases when $O(1) < c < n^\delta$, the question is not so straightforward. However, with a stronger complexity assumption (the Strong Exponential Time Hypothesis, SETH), we can also show a hardness result here for any algorithm running in sub-quadratic time. This is indeed a relevant observation, since having quadratic running time is often already prohibitive in practice.
Theorem 6.4. For multi-constraint partitioning with \( c = \omega(\log n) \), no finite factor approximation algorithm is possible in subquadratic time (i.e., \( n^{2-\delta} \) for some \( \delta > 0 \)) if SETH holds.

Proof (Sketch). The proof uses a reduction from the so-called Orthogonal Vectors Problem (OVP): given a set of \( m \) binary vectors \( a_1, \ldots, a_m \), the goal is to decide whether any two of these vectors are orthogonal (i.e., their dot product is 0). It is known that for vectors of dimension \( D = \omega(\log m) \), this cannot be decided in \( O(m^{2-\delta}) \) time unless SETH is falsified [20].

The main idea of the construction is to have a separate node \( v_i^{(j)} \) representing the \( j \)-th dimension of the \( i \)-th vector for each \( i \in [m] \), \( j \in [D] \), and a further “anchor” node \( u_i \) for each vector. For each fixed \( j_0 \in [m] \), we add a hyperedge containing the node \( u_{i_0} \), and all nodes \( v_{i_0}^{(j)} \) such that the \( j \)-th coordinate of \( a_i \) is 1.

Then through a series of technical steps, we create balance constraints that fulfill the following properties. Firstly, for each fixed \( j_0 \in [D] \), we add a dimension-wise balance constraint which ensures that at most one of the nodes \( v_{i_0}^{(j)} \) can be red. Furthermore, we also add a single balance constraint on the anchor nodes, ensuring that at least two of the nodes \( u_i \) need to be red.

Assume we want to find a valid multi-constraint partitioning of cost 0 in the resulting construction. This requires us to color two of the anchor nodes \( u_{i_0} \) and \( u_{i_1} \) red; however, then for all entries that are 1 in the chosen vectors, the corresponding nodes \( v_{i_0}^{(j)} \) must also be red. However, such a solution can only satisfy the dimension-wise balance constraints if there is no dimension \( j \) where both vectors have an entry of 1, i.e. if they are orthogonal. □

7 HIERARCHICAL COST FUNCTION

The simplicity of the partitioning problem makes it a very popular model to analyze the parallel execution of computations. On the other hand, due to this simplicity, the model cannot capture one of the most prominent characteristics of modern computing architectures, namely non-uniform memory access: transferring data between different pairs of processing units can have very different costs in practice. This is usually due to the hierarchical structure of these architectures: we often have several cores within the same processor, several processors attached to the same RAM, and then several of these computers connected over a network. Modern architectures even expose such hierarchical structure within single processors. In such an architecture, the communication cost between two cores heavily depends on the highest level of the hierarchy that the data has to cross: sending data between two cores on the same processor is a relatively fast operation, while sending data to another core through the top-level network connection is drastically slower.

As such, it is a natural goal to extend our analysis of partitioning problems to such a hierarchical setting. Formally, we will model these architectures by a rooted tree of depth \( d \), with the leaves of the tree corresponding to the compute units. We assume that each level of this tree has a fixed branching factor \( b_l \), i.e. every node on the \( i \)-th level (from the top) has exactly \( b_l \) children; this implies that we partition our hypergraph into \( k = \prod_{i=1}^{d} b_l \) sets. Furthermore, assume we have a set of constant cost parameters \( g_1, \ldots, g_d \) such that if two computing units have their lowest common ancestor in level \( i \) of the tree, then transferring a variable between the parts has a cost of \( g_i \) (as illustrated in Figure 7).

Figure 7: Illustration of communication costs in the hierarchical setting: the cost of transferring a variable depends on the level of the hierarchy that the data has to cross.

\[ \sum_{i=1}^{d} g_i \cdot (\lambda_e^{(i)} - \lambda_e^{(i-1)}) \]

and the total cost of a partitioning is again the sum of this cost over all hyperedges \( e \in E \).

For example, if \( e \) intersects all the \( k = 4 \) parts in a 2-level hierarchy with \( b_1 = b_2 = 2 \), then regardless of which part the variable is stored in, the cost of transferring it to the other three parts is \( g_1 + 2 \cdot g_2 = g_1 + 2 \); we need to move the variable once over the top level, and twice over the bottom level of the hierarchy. The formula indeed equals to this for \( \lambda_e^{(1)} = 2 \) and \( \lambda_e^{(2)} = 4 \). Note that the standard partitioning problem is obtained as a special case of this setting when our hierarchy has depth \( d = 1 \).

This hierarchical cost function results in a more complex version of the partitioning problem, where the role of different parts is not symmetric anymore. We briefly discuss some key properties of this more realistic model below. Besides the details of these proofs, the full version also provides a brief discussion of two more questions that arise naturally regarding our results on this hierarchical model: (i) how they carry over to hyperDAGs and/or the multi-constraint setting from previous sections, and (ii) how they can be generalized to cost functions that are inspired not by a tree, but by a different (arbitrary) processor topology.

7.1 Recursive approach

A natural solution idea for partitioning is to recursively split \( G \) into smaller and smaller parts. Even in the regular \( k \)-way partitioning
problem, such a recursive approach is very commonly used in
heuristics: we can repeatedly split each part into two further parts,
until the number of parts reaches $k$.

This recursive method also provides a natural solution approach
for our hierarchical partitioning problem: we can first try to split
$G$ cleverly into $b_1$ parts, then split each of these into $b_2$ further
parts, and so on, forming the entire hierarchy of $k$ parts in such a
recursive way. Moreover, the approach is even more intuitive in
this hierarchical case: since the cuts on the highest level induce a
much larger cost, it seems reasonable to first minimize the number
of these cuts, and only then move on to the lower levels.

It has already been observed before that such a recursive ap-
proach is not always optimal $[5, 47]$; however, we can show in a
simple example that it can even be a linear factor away from the
optimum cost.

**Lemma 7.2.** The solution returned by recursive partitioning can be
a factor $\Theta(n)$ off the optimum cost, both for regular and hierarchical
partitioning, even if each of the recursive steps is optimal separately.

**Proof (sketch).** Let $b_1 = b_2 = 2$, and consider the hypergraph
sketched in Figure 8, consisting of 9 densely connected blocks and
only a few hyperedges between these blocks. Assume that the larger
blocks in the figure each consist of $\frac{n}{2}$ nodes, while the smaller blocks
each consist of $\frac{n}{12}$ nodes.

In an optimal recursive partitioning, the first step will split this
hypergraph into two parts of equal size along the vertical axis,
without cutting any hyperedges (see the left side of the figure).
However, in the next step, the recursive approach needs to split both
sides into two further parts; with a small enough $\epsilon$ in the balance
constraint, this forces us to split one of the densely connected blocks
on the left side, resulting in a cost of $\Theta(n)$.

In contrast to this, there exists a direct 4-way partitioning of the
hypergraph where only $O(1)$ hyperedges are cut (right side of the
figure). This solution has a factor $\Theta(n)$ smaller cost than the recur-
sive solution, not only under the regular (cut-net or connectivity)
cost metrics, but also according to our hierarchical cost function,
since the coefficients $g_i$ are constants.

### 7.2 Hierarchy-agnostic partitioning

Another natural idea in this setting is to apply a regular partitioning
algorithm that does not consider the underlying processor hierarchy
at all. More specifically, given an input hypergraph, we can use the
following two-step method to obtain a hierarchical partitioning:

(i) first find a good regular $k$-way partitioning of the hypergraph,
(ii) then assign these $k$ parts to the $k$ leaf positions in the hierarchy
in a clever way.

For the analysis of this two-step method, we will assume that both
steps happen optimally: we first find an optimal (regular) partition-
ing of the hypergraph, and then we also assign the $k$ parts to
hypergraph positions in an optimal way. This allows us to study a
fundamental question: what happens if we have a good partition-
ing algorithm, but we disregard the hierarchical nature of modern
computing architectures in the partitioning step?

On the one hand, it is easy to show that the optimal solution
with this two-step method is at most a factor $g_1$ worse than the
true optimum for hierarchical cost. Intuitively, this is because an
optimal algorithm for standard partitioning can only misjudge the
real (hierarchical) cost of each hyperedge by a $\frac{g_1}{g_1} = g_1$ factor.

**Lemma 7.3.** The two-step method is a $g_1$-approximation.

On the other hand, it turns out that in unfortunate cases, the
difference can indeed be in the magnitude of a factor $g_1$. This carries
an important conceptual message: if we ignore the fact that the true
nature of the cost function is hierarchical in practice, then even by
finding the optimal partitioning, we might still be a large constant
factor away from the actual optimum cost.

**Theorem 7.4.** The two-step method can be a factor $\frac{b_1 - 1}{b_1} \cdot g_1$ worse
than the optimum.

**Proof (sketch).** On a high level, the proof requires a star-shaped
construction where a large block $A$ is densely connected to $(k - 1)$
smaller blocks $B_i$, which can either all fit into a single part, or into
separate parts (sketched in Figure 9). The construction is carefully
designed such that placing each $B_i$ in a separate part has slightly
smaller standard cost, so this solution is preferred by the two-step

![Figure 8: Construction for Lemma 7.2. Large and small squares correspond to blocks of size $\frac{n}{2}$ and $\frac{n}{12}$, respectively. Recursive bipartitioning (left side) first makes an optimal split of cost $0$ along the vertical axis; however, in the next recursive step, it needs to split one of the blocks to fulfill the balance constraint, resulting in a cost of $\Theta(n)$. On the other hand, direct $k$-way partitioning (right side) can provide a solution of cost $O(1)$ only.](image-url)
we can make the following simple observations:

level, then the same connections only incur a cost of $B_i$. On the other hand, the problem does become significantly more technical in this hierarchical setting in practice; on the other hand, it does not affect the hardness of the partitioning problem. On the one hand, the problem still remains in the parameterized complexity class $\mathcal{XP}$ with respect to $b_i$. Note that with our assumption so far that $k \in O(1)$, the number of possible solutions to this problem is only a function of $k$, and hence also a constant; as such, the problem is trivial from a complexity-theoretic perspective. However, the problem becomes more interesting if we briefly explore the case when $k$ is a variable part of the input. This setting might be relevant in applications where the partitioning task is severely time-critical, and hence instead of using a fixed architecture, we e.g. decide to increase $k$ proportionally to the hypergraph size.

We conclude the paper by briefly analyzing the complexity of this hierarchy assignment problem in the simplest case of only $d = 2$ levels. In this case, one can essentially contract each of the $k$ partitions into a single node, and express the hierarchy assignment problem as a specific kind of partitioning task on the resulting contracted hypergraph (which might contain multiple copies of some hyperedges). Our results show that this two-level hierarchy assignment problem is polynomially solvable for $b_2 = 2$, but already NP-hard if $b_2 = 3$.

**Theorem 7.4.** Consider the hierarchy assignment problem with only $d = 2$ levels.

- for $b_2 = 2$, the problem is solvable in polynomial time.
- for $b_2 = 3$, the problem is already NP-hard.

The two-step method, on the other hand, raises a more interesting question if we consider its second step as a separate hierarchy assignment problem. That is, given an already fixed $k$-way partitioning of the hypergraph, our goal is to assign the $k$ fixed parts to the $k$ available positions in the hierarchy optimally, i.e. such that the total hierarchical cost is minimized.

Note that our assumption so far that $k \in O(1)$, the number of possible solutions to this problem is only a function of $k$, and hence also a constant; as such, the problem is trivial from a complexity-theoretic perspective. However, the problem becomes more interesting if we briefly explore the case when $k$ is a variable part of the input. This setting might be relevant in applications where the partitioning task is severely time-critical, and hence instead of using a fixed architecture, we e.g. decide to increase $k$ proportionally to the hypergraph size.

We conclude the paper by briefly analyzing the complexity of this hierarchy assignment problem in the simplest case of only $d = 2$ levels. In this case, one can essentially contract each of the $k$ partitions into a single node, and express the hierarchy assignment problem as a specific kind of partitioning task on the resulting contracted hypergraph (which might contain multiple copies of some hyperedges). Our results show that this two-level hierarchy assignment problem is polynomially solvable for $b_2 = 2$, but already NP-hard if $b_2 = 3$.

**Theorem 7.5.** Consider the hierarchy assignment problem with only $d = 2$ levels.

- for $b_2 = 2$, the problem is solvable in polynomial time.
- for $b_2 = 3$, the problem is already NP-hard.

The two-step method, on the other hand, raises a more interesting question if we consider its second step as a separate hierarchy assignment problem. That is, given an already fixed $k$-way partitioning of the hypergraph, our goal is to assign the $k$ fixed parts to the $k$ available positions in the hierarchy optimally, i.e. such that the total hierarchical cost is minimized.

Note that with our assumption so far that $k \in O(1)$, the number of possible solutions to this problem is only a function of $k$, and hence also a constant; as such, the problem is trivial from a complexity-theoretic perspective. However, the problem becomes more interesting if we briefly explore the case when $k$ is a variable part of the input. This setting might be relevant in applications where the partitioning task is severely time-critical, and hence instead of using a fixed architecture, we e.g. decide to increase $k$ proportionally to the hypergraph size.

We conclude the paper by briefly analyzing the complexity of this hierarchy assignment problem in the simplest case of only $d = 2$ levels. In this case, one can essentially contract each of the $k$ partitions into a single node, and express the hierarchy assignment problem as a specific kind of partitioning task on the resulting contracted hypergraph (which might contain multiple copies of some hyperedges). Our results show that this two-level hierarchy assignment problem is polynomially solvable for $b_2 = 2$, but already NP-hard if $b_2 = 3$.

**Theorem 7.5.** Consider the hierarchy assignment problem with only $d = 2$ levels.

- for $b_2 = 2$, the problem is solvable in polynomial time.
- for $b_2 = 3$, the problem is already NP-hard.

The two-step method, on the other hand, raises a more interesting question if we consider its second step as a separate hierarchy assignment problem. That is, given an already fixed $k$-way partitioning of the hypergraph, our goal is to assign the $k$ fixed parts to the $k$ available positions in the hierarchy optimally, i.e. such that the total hierarchical cost is minimized.

Note that with our assumption so far that $k \in O(1)$, the number of possible solutions to this problem is only a function of $k$, and hence also a constant; as such, the problem is trivial from a complexity-theoretic perspective. However, the problem becomes more interesting if we briefly explore the case when $k$ is a variable part of the input. This setting might be relevant in applications where the partitioning task is severely time-critical, and hence instead of using a fixed architecture, we e.g. decide to increase $k$ proportionally to the hypergraph size.

We conclude the paper by briefly analyzing the complexity of this hierarchy assignment problem in the simplest case of only $d = 2$ levels. In this case, one can essentially contract each of the $k$ partitions into a single node, and express the hierarchy assignment problem as a specific kind of partitioning task on the resulting contracted hypergraph (which might contain multiple copies of some hyperedges). Our results show that this two-level hierarchy assignment problem is polynomially solvable for $b_2 = 2$, but already NP-hard if $b_2 = 3$.

**Theorem 7.5.** Consider the hierarchy assignment problem with only $d = 2$ levels.

- for $b_2 = 2$, the problem is solvable in polynomial time.
- for $b_2 = 3$, the problem is already NP-hard.

The two-step method, on the other hand, raises a more interesting question if we consider its second step as a separate hierarchy assignment problem. That is, given an already fixed $k$-way partitioning of the hypergraph, our goal is to assign the $k$ fixed parts to the $k$ available positions in the hierarchy optimally, i.e. such that the total hierarchical cost is minimized.

Note that with our assumption so far that $k \in O(1)$, the number of possible solutions to this problem is only a function of $k$, and hence also a constant; as such, the problem is trivial from a complexity-theoretic perspective. However, the problem becomes more interesting if we briefly explore the case when $k$ is a variable part of the input. This setting might be relevant in applications where the partitioning task is severely time-critical, and hence instead of using a fixed architecture, we e.g. decide to increase $k$ proportionally to the hypergraph size.

We conclude the paper by briefly analyzing the complexity of this hierarchy assignment problem in the simplest case of only $d = 2$ levels. In this case, one can essentially contract each of the $k$ partitions into a single node, and express the hierarchy assignment problem as a specific kind of partitioning task on the resulting contracted hypergraph (which might contain multiple copies of some hyperedges). Our results show that this two-level hierarchy assignment problem is polynomially solvable for $b_2 = 2$, but already NP-hard if $b_2 = 3$.

**Theorem 7.5.** Consider the hierarchy assignment problem with only $d = 2$ levels.

- for $b_2 = 2$, the problem is solvable in polynomial time.
- for $b_2 = 3$, the problem is already NP-hard.
