

Fast and Parallel Decomposition of Constraint Satisfaction Problems

Georg Gottlob^{1,2}, Cem Okulmus² and Reinhard Pichler²

¹University of Oxford, UK

²TU Wien, Austria

georg.gottlob@cs.ox.ac.uk, {cokulmus,pichler}@dbai.tuwien.ac.at

Abstract

Constraint Satisfaction Problems (CSP) are notoriously hard. Consequently, powerful decomposition methods have been developed to overcome this complexity. However, this poses the challenge of actually computing such a decomposition for a given CSP instance, and previous algorithms have shown their limitations in doing so. In this paper, we present a number of key algorithmic improvements and parallelisation techniques to compute so-called Generalized Hypertree Decompositions (GHDs) faster. We thus advance the ability to compute optimal (i.e., minimal-width) GHDs for a significantly wider range of CSP instances on modern machines. This lays the foundation for more systems and applications in evaluating CSPs and related problems (such as Conjunctive Query answering) based on their structural properties.

1 Introduction

Many real-life tasks can be effectively modelled as CSPs, giving them a vital importance in many areas of Computer Science. As solving CSPs is a classical NP-complete problem, there is a large body of research to find tractable fragments. One such line of research focuses on the underlying *hypergraph structure* of a CSP instance. A key result in this area is that CSP instances whose underlying hypergraph is acyclic, can be solved in polynomial time [Yannakakis, 1981]. Several generalisations of acyclicity have been identified by defining various forms of hypergraph *decompositions*, each associated with a specific notion of *width* [Gottlob *et al.*, 2000; Cohen *et al.*, 2008]. Intuitively, the width measures how far away a hypergraph is from being acyclic, with a width of 1 describing the acyclic hypergraphs.

In this work, we focus on Generalized Hypertree Decompositions (GHD) [Gottlob *et al.*, 2009], and generalized hypertree width (*ghw*). The computation of GHDs is itself intractable in the general case, already for width 2 [Fischl *et al.*, 2018]. However, for (hypergraphs of) CSPs with realistic restrictions, this problem becomes tractable. One such restriction is the bounded intersection property (BIP), which requires that any two constraints in a CSP only share a bounded

number of variables [Fischl *et al.*, 2018]. Indeed, by examining a large number of CSPs from various benchmarks and real-life applications, it has been verified that this intersection of variables tends to be small in practice [Fischl *et al.*, 2019]. In that work, over 3,000 instances of hypergraphs of CSPs and also of Conjunctive Queries (CQs) were examined and made publicly available in the HyperBench benchmark at <http://hyperbench.dbai.tuwien.ac.at>

The use of such decompositions can speed up the solving of CSPs and also the answering of CQs significantly. In fact, in [Aberger *et al.*, 2016] a speed-up up to a factor of 2,500 was reported for the CQs studied there. Structural decompositions are therefore already being used in commercial products and research prototypes, both in the CSP area as well as in database systems [Aberger *et al.*, 2016; Aref *et al.*, 2015; Amroun *et al.*, 2016; Habbas *et al.*, 2015; Lalou *et al.*, 2009]. However, previous decomposition algorithms are limited in that they fail to find optimal decompositions (i.e., decompositions of minimal width) even for low widths. This is also the case for various GHD computation methods proposed in [Fischl *et al.*, 2019; Fichte *et al.*, 2018]. The overall aim of our work is therefore to advance the art of computing hypergraph decompositions and to make the use of GHDs for solving CSPs applicable to a significantly wider range of CSP instances than previous methods. More specifically, we derive the following research goals:

Main Goal: Provide major improvements for computing hypergraph decompositions.

As part of this main goal, we define in particular:

Sub-goal 1: Design novel parallel algorithms for structural decompositions, in particular GHDs, and

Sub-goal 2: put all this to work, by implementing and extensively evaluating these improvements.

As a first step in pursuing the first goal, we aim at *generally applicable* simplifications of hypergraphs to speed up the decomposition of hypergraphs. Here, “general applicability” means that these simplifications can be incorporated into any decomposition algorithms such as the ones presented in [Fichte *et al.*, 2018; Fischl *et al.*, 2019] and also earlier work such as [Gottlob and Samer, 2008]. Moreover, we aim at heuristics for guiding the decomposition algorithms to explore more promising parts of the big search space first.

However, it will turn out that these simplifications and heuristics are not sufficient to overcome a principal shortcoming of existing decomposition algorithms, namely their sequential nature. Modern computing devices consist of multi-core architectures, and we can observe that single-core performance has mostly stagnated since the mid-2000s. So to produce programs which run optimally on modern machines, one must find a way of designing them to run efficiently in parallel. However, utilising multi-core systems is a non-trivial task, which poses several challenges. In our design of parallel GHD-algorithms, we focus on three key issues:

- i minimising synchronisation delay as much as possible,
- ii finding a way to partition the search space equally among CPUs, and thus utilising the resources optimally,
- iii supporting efficient backtracking, a key element of all structural decomposition algorithms presented so far.

In order to evaluate our algorithmic improvements and our new parallel GHD-algorithms, we have implemented them and tested them on the publicly available HyperBench benchmark mentioned above. For our implementation, we decided to use the programming language Go proposed by Google [Donovan and Kernighan, 2015], which is based on the classical Communication Sequential Processes pattern by [Hoare, 1978], since it reduces the need for explicit synchronisation.

To summarise the **main results** of this work:

- We have developed three parallel algorithms for computing GHDs, where the first two are loosely based on the *balanced separator* methods from [Akotov, 2010; Fischl *et al.*, 2019]. Moreover, we have designed a hybrid approach, which combines the strengths of parallel and sequential algorithms. This hybrid approach ultimately proved to be the best.
- In addition to designing parallel algorithms, we propose several algorithmic improvements such as applying multiple pre-processing steps on the input hypergraphs and using various heuristics to guide the search for a decomposition. Moreover, for the hybrid approach, we have explored when to best switch from one approach to the other.
- We have implemented the parallel algorithms together with all algorithmic improvements and heuristics presented here. The source code of the program is available under <https://github.com/cem-okulmus/BalancedGo>. With our new algorithms and their implementation, dramatically more instances from HyperBench could be solved compared with previous algorithms. More specifically, we could extend the number of hypergraphs with exact *ghw* known by over 50%. In total, this means that for over 75% of all instances of HyperBench, the exact *ghw* is now known. If we leave aside the randomly-generated CSPs, and focus on the those from real world applications, we can show an increase of close to 100%, almost doubling the number of instances solved.

Our work therefore makes it possible to compute GHDs efficiently on modern machines for a wide range of CSPs. It enables the fast recognition of low widths for many instances encountered in practice (as represented by HyperBench) and thus lays the foundation for more systems and applications in evaluating CSPs and CQs based on their structural properties.

The remainder of this paper is structured as follows: In Section 2, we provide the needed terminology and recall previous approaches. In Section 3, we present our main algorithmic contributions. This is followed by presenting experimental evaluations in Section 4. In Section 5, we summarise our main results and highlight directions for future work.

2 Preliminaries

CSPs & Hypergraphs. A *constraint satisfaction problem* (CSP) P is a set of *constraints* (S_i, R_i) , where S_i is a set of variables and R_i a constraint relation which contains the valid combination of values that variables in S_i can take. A solution to P is a mapping of variables to values, such that for each constraint we map the variables to a corresponding combination in its constraint relation. A *hypergraph* H is a tuple $(V(H), E(H))$, consisting of a set of vertices $V(H)$ and a set of hyperedges (synonymously, simply referred to as “edges”) $E(H) \subseteq 2^{V(H)}$. To get the hypergraph of a CSP P , we consider $V(H)$ to be the set of all variables in P , and each S_i to be one hyperedge. A hypergraph H is said to have the bounded intersection property (BIP), if there exists a constant c such that for any two edges $e_1, e_2 \in E(H)$, $e_1 \cap e_2 \leq c$.

Decompositions. A GHD of a hypergraph $H = (V(H), E(H))$ is a tuple $\langle T, \chi, \lambda \rangle$, where $T = (N, E(T))$ is a tree, and χ and λ are labelling functions, which map to each node $n \in N$ two sets, $\chi(n) \subseteq V(H)$ and $\lambda(n) \subseteq E(H)$. For a node n we call $\chi(n)$ the *bag*, and $\lambda(n)$ the *edge cover* of n . We denote with $B(\lambda(n))$ the set $\{v \in V(H) \mid v \in e, e \in \lambda(n)\}$, i.e., the set of vertices “covered” by $\lambda(n)$. The functions χ and λ have to satisfy the following conditions:

1. For each $e \in E(H)$, there is a node $n \in N$ s.t. $e \subseteq \chi(n)$.
2. For each vertex $v \in V(H)$, $\{n \in N \mid v \in \chi(n)\}$ is a connected subtree of T .
3. For each node $n \in N$, we have that $\chi(n) \subseteq B(\lambda(n))$.

The *width of a GHD* is defined as $\max\{|\lambda(n)| : n \in N\}$. The generalized hypertree width (*ghw*) of a hypergraph is the smallest width of any of its GHDs. Deciding if $ghw(H) \leq k$ for a hypergraph H and fixed k is NP-complete, as one needs to consider exponentially many possible choices for the bag $\chi(n)$ for a given edge cover $\lambda(n)$.

It was shown in [Fischl *et al.*, 2019] that for any class of hypergraphs enjoying the BIP, one only needs to consider a polynomial set of subsets of hyperedges (called *subedges*) to compute their *ghw*.

Components & Separators. Consider a set of vertices $W \subseteq V(H)$. A set C of vertices with $C \subseteq V(H) \setminus W$ is $[W]$ -*connected* if for any two distinct vertices $v, w \in C$ there exists a sequence of vertices $v = v_0, \dots, v_h = w$ and a sequence of edges e_0, \dots, e_{h-1} ($h \geq 0$) such that $\{v_i, v_{i+1}\} \subseteq (e_i \setminus W)$, for each $i \in \{0, \dots, h-1\}$. A set $C \subseteq V(H)$ is a $[W]$ -*component*, if C is maximal $[W]$ -connected. For a set of edges $S \subseteq E(H)$, we say that C is “[S]-connected” or an “[S]-component” as a short-cut for C is “[W]-connected” or a “[W]-component”, respectively, with $W = \bigcup_{e \in S} e$. We also call S a *separator* in this context. The

size of an $[S]$ -component C is defined as the number of edges $e \in E(H)$ such that $e \cap C \neq \emptyset$. For a hypergraph H and a set of edges $S \subseteq E(H)$, we say that S is a *balanced separator* if all $[S]$ -components of H have a size $\leq \frac{|E(H)|}{2}$.

It was shown in [Adler *et al.*, 2007] that, for every GHD $\langle T, \chi, \lambda \rangle$ of a hypergraph H , there exists a node $n \in N$ such that $\lambda(n)$ is a balanced separator of H . This property can be made use of when searching for a GHD of size k of a hypergraph H as we shall recall in Section 2.2 below.

2.1 Computing Hypertree Decompositions (HDs)

We briefly recall the basic principles of the Det-K-Decomp program from [Gottlob and Samer, 2008] for computing Hypertree Decompositions (HDs), which was the first implementation of the original HD algorithm from [Gottlob *et al.*, 2002]. HDs are GHDs with an additional condition to make their computation tractable in a way explained next.

For fixed $k \geq 1$, Det-K-Decomp tries to construct an HD of a hypergraph H in a top-down manner. It thus maintains a set C of vertices, which is initialised to $C := V(H)$. For a node n in the HD (initially, this is the root of the HD), it “guesses” an edge cover $\lambda(n)$, i.e., $\lambda(n) \subseteq E(H)$ and $|\lambda(n)| \leq k$. For fixed k , there are only polynomially many possible values $\lambda(n)$. Det-K-Decomp then proceeds by determining all $[\lambda(n)]$ -components C_i with $C_i \subseteq C$. The additional condition imposed on HDs (compared with GHDs) restricts the possible choices for $\chi(n)$ and thus guarantees that the $[\lambda(n)]$ -components inside C and the $[\chi(n)]$ -components inside C coincide. This is the crucial property for ensuring polynomial time complexity of HD-computation – at the price of possibly missing GHDs with a lower width.

Now let C_1, \dots, C_ℓ denote the $[\lambda(n)]$ -components with $C_i \subseteq C$ and let E_1, \dots, E_ℓ denote subsets of $E(H)$ with $E_i = \{e \mid e \cap C_i \neq \emptyset\}$ for every i . By the maximality of components, these sets E_i are pairwise disjoint. Moreover, it was shown in [Gottlob *et al.*, 2002] that if H has an HD of width $\leq k$, then it also has an HD of width $\leq k$ such that the edges in each E_i are “covered” in different subtrees below n . More precisely, this means that n has ℓ child nodes n_1, \dots, n_ℓ , such that for every i and every $e \in E_i$, there exists a node n_e in the subtree rooted at n_i with $e \subseteq \chi(n_e)$. Hence, Det-K-Decomp recursively searches for an HD of the hypergraphs H_i with $E(H_i) = E_i$ and $V(H_i) = \bigcup E_i$ with the slight extra feature that also edges from $E \setminus E_i$ are allowed to be used in the λ -labels of these HDs.

2.2 Computing GHDs

It was shown in [Fischl *et al.*, 2018] that, even for fixed $k = 2$, deciding if $ghw(H) \leq k$ holds for a hypergraph H is NP-complete. However, it was also shown there that if a class of hypergraphs satisfies the BIP, then the problem becomes tractable. The main reason for the NP-completeness in the general case is that, for a given edge cover $\lambda(n)$, there can be exponentially many bags $\chi(n)$ satisfying condition 3 of GHDs, i.e., $\chi(n) \subseteq B(\lambda(n))$. Now let $\lambda(n) = \{e_{i_1}, \dots, e_{i_\ell}\}$ with $\ell \leq k$. Of course, if we restrict each e_{i_j} to the subedge $e'_{i_j} = e_{i_j} \cap \chi(n)$ and define $\lambda'(n) = \{e'_{i_1}, \dots, e'_{i_\ell}\}$, then we get $\chi(n) = B(\lambda'(n))$. The key to the tractability shown

in [Fischl *et al.*, 2018] in case of the BIP (say, the intersection of any two distinct edges is bounded by a constant b) is that each e'_{i_j} is either equal to e_{i_j} or a subset of e_{i_j} with $|e'_{i_j}| \leq k \cdot b$. Hence, there are only polynomially many choices of subedges e'_{i_j} and also of $\chi(n)$. In [Fischl *et al.*, 2019], this property was used to design a program for GHD computation as a straightforward extension of Det-K-Decomp by adding the relevant (polynomially many!) subedges.

In [Fischl *et al.*, 2019], yet another GHD algorithm was presented. It is based on the use of *balanced separators* and extends ideas from [Akutov, 2010]. The motivation of this approach comes from the observation that there is no useful upper bound on the size of the subproblems that have to be solved by the recursive calls of the Det-K-Decomp algorithm. In fact, for some node n with corresponding component C and edge set E , let C_1, \dots, C_ℓ denote the $[\lambda(n)]$ -components with $C_i \subseteq C$. Then there may exist an i such that $E_i = \{e \mid e \cap C_i \neq \emptyset\}$ is “almost” as big as E . In other words, in the worst case, the recursion depth of Det-K-Decomp may be linear in the number of edges.

The Balanced Separator approach from [Fischl *et al.*, 2019] uses the fact that every GHD must contain a node whose λ -label is a balanced separator. Hence, in each recursive decomposition step for some subset E_C of the edges of H , the algorithm “guesses” a node n' such that $\lambda(n')$ is a balanced separator of the hypergraph with edges E_C . Of course, this node n' is not necessarily a child node n_i of the current node n but may lie somewhere inside the subtree T_i below n . However, since GHDs can be arbitrarily rooted, one may first compute this subtree T_i with n' as the root and with n_i as a leaf node. This subtree is then (when returning from the recursion) connected to node n by rerooting T_i at n_i and turning n_i into a child node of n . The definition of balanced separators guarantees that the recursion depth is logarithmically bounded. This makes the Balanced Separator algorithm a good starting point for our parallel algorithm to be presented in Section 3.2.

3 Algorithmic Improvements & Parallelism

In this section, we first present a list of algorithmic improvements and then describe our strategy for designing parallel decomposition algorithms. Finally, we will show how to obtain a good hybrid strategy, which combines the strengths of the parallel and sequential approaches.

3.1 Algorithmic Improvements

Hypergraph preprocessing

An important consideration for speeding up decomposition algorithms is the simplification of the input hypergraph. Our considered reductions are:

- applying the GYO reduction from [Graham, 1979; Yu and Özsoyoğlu, 1979], which repeatedly allows a) the removal of vertices occurring in a single edge, as well as b) edges contained in other edges. Note that the application of one rule may enable the application of the other rule, so their combination may lead to a greater simplification compared to just any one rule alone.

- the removal of all but one vertex out of a set of vertices of the same *type*. Here the *type* of a vertex v is defined as the set of edges e which contain v .

It is important to note that all these reductions neither add nor lose solutions. Formally, we get the following property:

Theorem 1 *Preprocessing an input hypergraph by exhaustive application of the GYO reduction and the elimination of vertices of the same type as another vertex in the hypergraph, is sound and complete. More specifically, let H' be the result of applying this preprocessing to a hypergraph H . Then, for any $k \geq 1$, we have $ghw(H') \leq k$ if and only if $ghw(H) \leq k$. Moreover, any GHD of H' can be efficiently transformed into a GHD of H of the same width.*

Finding balanced separators fast

It has already been observed in [Gottlob and Samer, 2008] that the ordering in which edges are considered is vital for finding an appropriate edge cover $\lambda(n)$ for the current node n in the decomposition fast. However, the ordering used in [Gottlob and Samer, 2008] for Det-K-Decomp, (which was called MCSO, i.e., maximal cardinality search ordering) turned out to be a poor fit for finding balanced separators. A natural alternative was to consider, for each edge e , all possible paths between vertices in the hypergraph H , and how much these paths increase after removal of e . This provides a weight for each edge, based on which we can define the *maximal separator ordering*. In our tests, this proved to be a very effective heuristic. Unfortunately, computing the maximal separator ordering requires solving the all-pairs shortest path problem. Using the well-known Floyd-Warshall algorithm [Floyd, 1962; Warshall, 1962] as a subroutine, this leads to a complexity of $O(|E(H)| * |V(H)|^3)$, which is prohibitively expensive for practical instances. We thus explored two other, computationally simpler, heuristics, which order the edges in descending order of the following measures:

- The *vertex degree* of an edge e is defined as $\sum_{v \in e} deg(v)$, where $deg(v)$ denotes the degree of a vertex v , i.e., the number of edges containing v .
- The *edge degree* of an edge e is $|\{f : e \cap f\}|$, i.e., the number of edges e has a non-empty intersection with.

In our empirical evaluation, we found both of these to be useful compromises between speeding up the computation and complexity of computing the ordering itself, with the vertex degree ordering yielding the best results, i.e., compute $\lambda(n)$ by first trying to select edges with higher vertex degree.

Reducing the search space

The GHD algorithm based on balanced separators presented in [Fischl *et al.*, 2019] searches through all ℓ -tuples of edges (with $\ell \leq k$) to find the next balanced separator. The number of edge-combinations thus checked is $\sum_{i=1}^k \binom{N}{i}$, where N denotes the number of edges. Note that this number of edges is actually higher than in the input hypergraph due to the subedges that have to be added for the tractability of GHD computation (see Section 2.2).

We can reduce this significantly by shifting our focus on the actual bags $\chi(n)$ generated from each $\lambda(n)$ thus computed. Therefore, we initially only look for balanced sepa-

rators of size k , checking $\binom{N}{k}$ many initial choices of $\lambda(n)$. Only if a choice of $\lambda(n)$ and $\chi(n) = \bigcup \lambda(n)$ does not lead to a successful recursive call of the decomposition procedure, we also inspect subsets of $\chi(n)$ – strictly avoiding the computation of the same subset of $\chi(n)$ several times by inspecting different subedges of the original edge cover $\lambda(n)$. We thus also do not add subedges to the hypergraph upfront but only as they are needed as part of the backtracking when the original edge cover $\lambda(n)$ did not succeed. Separators consisting of fewer edges are implicitly considered by allowing also the empty set as a possible subedge.

Summary. Our initial focus was to speed up existing decomposition algorithms via improvements as described above. However, even though these algorithmic improvements showed some positive effect, it turned out that a more fundamental change is needed. We thus turned our attention to parallelisation, which will be the topic of the next section.

3.2 Parallelisation Strategy

As described in more detail below, we use a divide and conquer method, based on the balanced separator approach. This method divides a hypergraph into smaller hypergraphs, called *subcomponents*. Our method proceeds to work on these subcomponents in parallel, with each round reducing the size of the hypergraphs (i.e., the number of edges in each subcomponent) to at most half their size. Thus after logarithmically many rounds, the method will have decomposed the entire hypergraph, if a decomposition of width K exists. For the computation we use the modern programming language Go [Donovan and Kernighan, 2015], which has a model of concurrency based on [Hoare, 1978].

In Go, a *goroutine* is a sequential process. Multiple goroutines may run concurrently. In the pseudocode provided, these are spawned using the keyword **go**, as can be seen in Algorithm 1, line 15. They communicate over *channels*. Using a channel ch is indicated by $\leftarrow ch$ for *receiving* from a channel, and by $ch \leftarrow$ for *sending* to ch .

Overview

Algorithm 1 contains the full decomposition procedure, whereas Algorithm 2 details the parallel search for separators, as it is a key subtask for parallelisation. To emphasise the core ideas of our parallel algorithm, we present it as a decision procedure, which takes as input a hypergraph H and a parameter K , and returns as output either *Accept* if $ghw(H) \leq K$ or *Reject* otherwise. Please note, however, that our actual implementation also produces a GHD of width $\leq K$ in case of an accepting run.

The parallel Balanced Separator algorithm begins with an initial call to the procedure **Decomp**, as seen in line 1 of Algorithm 1. The procedure **Decomp** takes two arguments, a hypergraph H' for the current subcomponent considered, and a set S_p of “special edges”. These special edges indicate the balanced separators encountered so far, as can be seen in line 15, where the current separator *subSep* is added to the argument on the recursive call, combining all its vertices into a new special edge. The special edges are needed to ensure that the decompositions of subcomponents can be combined to an overall decomposition, and are thus considered as additional

edges. The recursive procedure **Decomp** has its base case in lines 3 to 4, when the size of H' and Sp together is less than or equal to 2. The remainder of **Decomp** consists of two loops, the Separator Loop, from lines 6 to 23, which iterates over all balanced separators, and within it the SubEdge Loop, running from lines 11 to 22, which iterates over all subedge variants of any balanced separator. New balanced separators are produced with the subprocedure **FindBalSep**, used in line 7 of Algorithm 1, and detailed in Algorithm 2. After a separator is found, **Decomp** computes the new subcomponents in line 12. Then goroutines are started using recursive calls of **Decomp** for all found subcomponents. If any of these calls returns **Reject**, seen in line 18, then the procedure starts checking for subedges. If they have been exhausted, the procedure checks for another separator. If all balanced separators have been tried without success, then **Decomp** rejects in line 24.

We proceed to detail the parallelisation strategy of the two key subtasks: the search for new separators and the recursive calls over the subcomponents created from chosen separators.

Parallel search for balanced separators

For the search, while testing out a number of configurations, we settled ultimately on the following scenario, shown in Algorithm 2: The function **FindBalSep** first starts a number of Worker goroutines, seen in lines 3 to 4. These are also passed a channel, which they will use to send back any balanced separators they find. The worker procedure iterates over all candidate separators in its assigned search space, and sends back the first balanced separator it finds over the channel. Then **FindBalSep** waits for one of two conditions (whichever happens first): either one of the workers finds a balanced separator, lines 6 to 7, or none of them do and they all terminate on their own once they have exhausted the search space. Then the empty set is returned, as seen in line 8, indicating that no further balanced separators exist.

This design reduces the need for synchronisation: each worker is responsible for a share of the search space, and the only time a worker is stopped is when either it has found a separator which is balanced, or when another worker has done so. The number of worker goroutines scales with the number of available processors, thus allowing the search to optimally use the available resources. Our design addresses backtracking in this way: the workers employ a simple data structure, called M_i in Algorithm 2, to store their current progress, allowing them to generate the next separator to consider, and this data structure is stored in **Decomp**, seen in line 5 of Algorithm 1, even after the search is over. In case of backtracking, this allows to quickly continue the search exactly where it left off, without losing any work. If multiple workers find a balanced separator, one of them “wins”, and during backtracking, the other workers can immediately send their found separators to **FindBalSep** again.

Parallel recursive calls.

For the recursive calls on subcomponents encountered, we create for each such call its own goroutine, as explained in the overview. This can be seen in Algorithm 1, line 15, where the output is then sent back via the channel ch . Given enough resources, this allows for each call to be run completely sep-

Algorithm 1: Parallel Balanced Separator algorithm

Input: H : Hypergraph
Parameter: K : width parameter
Output: **Accept** if ghw of $H \leq K$, else **Reject**

```

1 return Decomp( $H, \emptyset$ ) ▷ initial call
2 function Decomp( $H'$ : Graph,  $Sp$ : Set of Special Edges)
3   if  $|H' \cup Sp| \leq 2$  then ▷ Base Case
4     return Accept
5    $M :=$  a set of  $K$ -tuples
6   repeat ▷ Separator Loop
7      $sep :=$  FindBalSep( $H', M$ )
8     if  $sep = \emptyset$  then
9       break
10     $subSep := sep$ 
11    repeat ▷ SubEdge Loop
12       $comp :=$  getComponents( $subSep, H', Sp$ )
13       $ch :=$  a channel
14      for  $c \in comp$  do
15        go  $ch \leftarrow$  Decomp( $c.H, c.Sp \cup \bigcup subSep$ )
16      while any recursive call still running do
17         $out \leftarrow ch$ : ▷ Wait on channel
18        if  $out = \text{Reject}$  then
19           $subSep =$  NextSubedgeSep( $subSep$ )
20          continue SubEdge Loop
21        return Accept ▷ Found decomposition
22    until  $subSep = \emptyset$ 
23  until  $sep = \emptyset$ 
24  return Reject ▷ Exhausted Search Space

```

arately, thus easily fulfilling the need to minimise synchronisation. This fact - that all recursive calls can be worked on in parallel - is also in itself a major performance boost: in the sequential case we execute all recursive calls in a loop, but in the parallel case - see lines 14 to 15 in Algorithm 1 - we can execute these calls simultaneously. Thus, if one parallel call rejects, we can stop all the other calls early, and thus potentially save a lot of time. It is easy to imagine a case where in the sequential execution, a rejecting call is encountered only near the end of the loop.

It is important to note that this parallel algorithm is still a correct decomposition procedure. More formally, we state the following property:

Theorem 2 *The algorithm for checking the ghw of a hypergraph given in Algorithm 1 is sound and complete. More specifically, Algorithm 1 with input H and parameter K will accept if and only if there exists a GHD of H with width $\leq K$. Moreover, by materialising the decompositions implicitly constructed in the recursive calls of the **DECOMP** function, a GHD of width $\leq K$ can be constructed efficiently in case the algorithm returns **Accept**.*

Based on this parallelisation scheme, we produced a parallel implementation of the Balanced Separator algorithm, with the improvements mentioned in Section 3.1. We already saw some promising results, but we noticed that for many in-

Algorithm 2: Parallel Search for Separators

```
1 function FindBalSep( $H'$ : Graph,  $M$ : Set of  $K$ -tuples)
2    $ch :=$  a channel
3   for  $M_i \in M$  do
4     go WORKER( $M_i, ch$ )
5   while any worker still running do
6      $out \leftarrow ch$ ;  $\triangleright$  Wait on channel
7     return out
8   return empty set  $\triangleright$  Exhausted Search Space
9 function Worker( $M_i$ :  $K$ -tuple of integers,  $ch$ : Channel)
10  for  $sep \in \text{NextSeparator}(M_i)$  do
11    if IsBalanced( $sep, H'$ ) then
12       $ch \leftarrow sep$   $\triangleright$  Send  $sep$  to Master
13  return  $\triangleright$  No separator found within  $M_i$ 
```

stances, this purely parallel approach was not fast enough. We thus continued to explore a more nuanced approach, mixing both parallel and sequential algorithms.

3.3 Hybrid Approach - Best of Both Worlds

We now present a novel combination of parallel and sequential decomposition algorithms. It contains all the improvements mentioned in Section 3.1 and combines the Balanced Separator algorithm from Section 3.2 and Det-K-Decomp recalled in Section 2.1.

This combination is motivated by two observations: The Balanced Separator algorithm is very effective at splitting larger hypergraphs into smaller ones and in negative cases, where it can quickly stop the computation if no balanced separator for a given subcomponent exists. It is slower for smaller instances where the computational overhead to find balanced separators at every step slows things down. Furthermore, for technical reasons, it is also less effective at making use of caching. Det-K-Decomp, on the other hand, with proper heuristics, is very efficient for small instances and it allows for effective caching, thus avoiding repetition of work.

The Hybrid approach proceeds as follows: For a fixed number m of rounds, the algorithm tries to find balanced separators. Each such round is guaranteed to half the number of hyperedges considered. Hence, after those m rounds, the number of hyperedges in the remaining subcomponents will be reduced to at most $\frac{|E(H)|}{2^m}$. The Hybrid algorithm then proceeds to finish the remaining subcomponents by using the Det-K-Decomp algorithm.

This required quite extensive changes to Det-K-Decomp, since it must be able to deal with Special Edges. Formally, each call of Det-K-Decomp runs sequentially. However, since the m rounds can produce a number of components, many calls of Det-K-Decomp can actually run in parallel. In other words, our Hybrid approach also brings a certain level of parallelism to Det-K-Decomp.

4 Experimental Evaluation and Results

We performed our experiments on the HyperBench benchmark, with the goal to determine the exact generalized hyper-

tree width of all instances. We evaluated how our approach compares with existing attempts to compute the ghw , and we investigated how various heuristics and parameters prove beneficial for various instances. The detailed results of our experiments¹, in addition to the source code of our Go programs² are publicly available. Together with the benchmark instances, which are detailed below and also publicly available, this ensures the reproducibility of our experiments.

4.1 Benchmark Instances and Setting

HyperBench. The instances used in our experiments are taken from the benchmark HyperBench, collated from various sources in industry and the literature, which was released by [Fischl *et al.*, 2019] and made publicly available at <http://hyperbench.dbai.tuwien.ac.at>. It consists of 3071 hypergraphs from CQs and CSPs, where for many CSP instances the exact ghw was still undetermined.

Hardware and Software. We used Go 1.2 for our implementation, called *BalancedGo*. Our experiments ran on a cluster of 12 nodes, running Ubuntu 16.04.1 LTS with a 24 core Intel Xeon E5-2650v4 CPU, clocked at 2.20 GHz, each node with 160 GB of RAM. We disabled HyperThreading for the experiments.

Setup and Limits. For the experiments, we set a number of limits to test the efficiency of our solution. For each run, consisting of the input (i.e., hypergraph H and integer K) and a configuration of the decomposer, we set a one hour (3600 seconds) timeout and limit available RAM to 1 GB. These limits are justified by the fact that these are the exact same limits as were used in [Fischl *et al.*, 2019], thus ensuring the direct comparability of our test results. To enforce these limits and run the experiments, we used the *HTCondor* software [Thain *et al.*, 2005], originally named just Condor.

4.2 Results

The key results from our experiments are summarised in Table 1. Under “Decomposition Algorithms” we use (*ensemble*) to indicate that results from two algorithms are collected, i.e., results from the Hybrid algorithm and the parallel Balanced Separator algorithm. To also consider the performance of one of the individual approaches introduced in Section 3, namely the results of the Hybrid approach (from Section 3.3) is separately shown in a section of the table. As a reference point, we considered the *NewDetK* solver from [Fischl *et al.*, 2019]. For each of these three, we also listed the average time and the maximal time to compute a GHD of optimal-width for each group of instances of HyperBench, as well as the standard deviation. The minimal times are left out for brevity, since they are always near or equal to 0. Note that for HyperBench the instance groups “CSP App.” or “CQ App.”, listed in Table 1, are hypergraphs of (resp.) CSP or CQ instances from real world applications.

Across all experiments, out of the 3071 instances in HyperBench, our implementation *BalancedGo* solved over 2300, as seen in Table 1. By “solved” we mean that the precise

¹See: Raw data available under [Gottlob *et al.*, 2020]

²See: <https://github.com/cem-okumus/BalancedGo>

Instances		Decomposition Algorithms											
Group	#graphs	NewDetK [Fischl <i>et al.</i> , 2019]				Hybrid Approach				BalancedGo <i>ensemble</i>			
		#solved	avg	max	stdev	#solved	avg	max	stdev	#solved	avg	max	stdev
CSP App.	1090	386	150.82	2608.0	490.47	762	36.60	3293.73	229.25	763	42.21	3479.05	248.65
CSP Random	863	412	65.78	3240.0	379.12	557	30.47	3465.20	226.93	625	24.75	3465.20	194.30
CSP Other	82	27	126.43	2538.0	422.42	40	80.38	2376.52	331.53	42	72.89	2376.52	296.52
CQ App.	535	535	0.00	0.0	0.00	535	0.00	0.01	0.00	535	0.00	0.32	0.01
CQ Random	500	281	2.12	3350.0	21.14	325	16.41	3213.82	183.30	354	18.18	3213.82	176.01
Total	3071	1642	59.00	3240.0	325.03	1730	30.00	3465.20	216.64	2320	29.36	3479.05	209.72

Table 1: Overview on the number of instances solved and running times (in seconds) for producing optimal-width GHDs

ghw could be determined in these cases. We observed that the number of 2320 solved instances exceeds the sum of the solved cases by the two individual implementations of Hybrid algorithm and parallel Balanced Separator. This is due to the fact that the “ensemble” combines results from all individual approaches. We mean with “combine” that we use information gained from runs of all possible algorithms. For a hypergraph H and a width k , an accepting run gives us an upper bound (since the optimal $ghw(H)$ is then clearly $\leq k$), and a rejecting run gives us a lower bound (since then we know that $ghw(H) > k$). By pooling multiple algorithms, we can combine these produced upper and lower bounds to compute the optimal width (when both bounds meet) for more instances than any one algorithm could determine on its own. We note that the results for NewDetK from (Fischl *et al.*, 2019) are also such an “ensemble”, combining the results of three different GHD algorithms. It turned out that parallel Balanced Separator is particularly well suited for deriving lower bounds (i.e., detecting “Reject”-cases), while the Hybrid algorithm is best for deriving upper bounds (i.e., detecting “Accept”-cases).

For the computationally most challenging instances of HyperBench, those of $ghw \geq 3$, this signifies an increase of over 70 % in solved instances when compared with [Fischl *et al.*, 2019]. In addition, when considering the CSP instances from real world applications, we managed to solve 763 instances, almost doubling the number from NewDetK. In total, we now know the exact ghw of around 70% of all hypergraphs from CSP instances and the exact ghw of around 75% of all hypergraphs of HyperBench.

We stress that, in the first place, our data is not about time, but rather about the number of instances solved within reasonable time constraints. And here we provide an improvement for these practical CSP instances of near 100% on the current state of the art; no such improvements have been achieved by other techniques recently. It is also noteworthy, that the Hybrid algorithm alone solved 1730 total cases, thus beating the total for NewDetK in [Fischl *et al.*, 2019], which, as mentioned, combines the results of three different GHD algorithms.

5 Conclusion and Outlook

We presented novel parallel algorithms in this paper, advancing the ability to compute GHDs of a significantly larger set of CSPs. This paves the way for more applications to use

them to speed up the evaluation of CSP instances.

For immediate future work, we want to look at parallel approaches to compute plain Hypertree Decompositions (HDs), since the Balanced Separator approach does not directly translate to them. Another interesting goal is harnessing Go’s excellent cloud computing capabilities to extend our results beyond the computation of GHDs to actually evaluating large real-life CSPs in the cloud.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF):P30930-N35. Georg Gottlob is a Royal Society Research Professor and acknowledges support by the Royal Society for the present work in the context of the project “RAISON DATA” (Project reference: RP\R1\201074). Gottlob is, moreover, grateful to Cristina Zoltan for having suggested Go as an appropriate language for the parallelisation of computing hypertree decompositions.

References

- [Aberger *et al.*, 2016] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 431–446, 2016.
- [Adler *et al.*, 2007] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007.
- [Akotov, 2010] Dmitri Akotov. *Exploiting parallelism in decomposition methods for constraint satisfaction*. PhD thesis, University of Oxford, UK, 2010.
- [Amroun *et al.*, 2016] Kamal Amroun, Zineb Habbas, and Wassila Aggoune-Mtalaa. A compressed generalized hypertree decomposition-based solving technique for non-binary constraint satisfaction problems. *AI Comm.*, 29(2):371–392, 2016.
- [Aref *et al.*, 2015] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on*

Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015, pages 1371–1382, 2015.

- [Cohen *et al.*, 2008] David A. Cohen, Peter Jeavons, and Marc Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *J. Comput. Syst. Sci.*, 74(5):721–743, 2008.
- [Donovan and Kernighan, 2015] Alan A. A. Donovan and Brian W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.
- [Fichte *et al.*, 2018] Johannes Klaus Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An SMT approach to fractional hypertree width. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, pages 109–127, 2018.
- [Fischl *et al.*, 2018] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and fractional hypertree decompositions: Hard and easy cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32, 2018.
- [Fischl *et al.*, 2019] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 464–480, 2019.
- [Floyd, 1962] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.
- [Gottlob and Samer, 2008] Georg Gottlob and Marko Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13, 2008.
- [Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.
- [Gottlob *et al.*, 2002] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, May 2002.
- [Gottlob *et al.*, 2009] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, September 2009.
- [Gottlob *et al.*, 2020] Georg Gottlob, Cem Okumus, and Reinhard Pichler. Raw Data on Experiments for BalancedGo. Zenodo, April 2020. doi:10.5281/zenodo.3767137.
- [Graham, 1979] M. H. Graham. On The Universal Relation. Technical report, University of Toronto, 1979.
- [Habbas *et al.*, 2015] Zineb Habbas, Kamal Amroun, and Daniel Singer. A forward-checking algorithm based on a generalised hypertree decomposition for solving non-binary constraint satisfaction problems. *J. Exp. Theor. Artif. Intell.*, 27(5):649–671, 2015.
- [Hoare, 1978] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
- [Lalou *et al.*, 2009] Mohammed Lalou, Zineb Habbas, and Kamal Amroun. Solving hypertree structured CSP: sequential and parallel approaches. In Marco Gavanelli and Toni Mancini, editors, *Proceedings of the 16th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA@AI*IA 2009, Reggio Emilia, Italy, December 11-12, 2009*, volume 589 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.
- [Thain *et al.*, 2005] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
- [Warshall, 1962] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
- [Yannakakis, 1981] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94, 1981.
- [Yu and Özsoyoğlu, 1979] C. T. Yu and M. Z. Özsoyoğlu. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, pages 306–312, 1979.