Parallel Computation of Generalized Hypertree Decompositions*

Georg Gottlob,[1,2] Cem Okulmus,[1] and Reinhard Pichler[1]

[1]TU Wien, Austria, [2]University of Oxford, UK

1 Introduction

Answering Conjunctive Queries (CQs) and solving Constraint Satisfaction Problems (CSPs) are arguably among the most important tasks in Computer Science. They are classical NP-complete problems. However, they have tractable fragments for instances where the underlying hypergraphs are acyclic. There has been active research for several decades to generalize acyclicity with several notions of decompositions and width [4]. Here we focus on generalized hypertree decompositions (GHDs) and generalized hypertree width (*ghw*).

Deciding if a given CQ or CSP (strictly speaking, the underlying hypergraph H) has $ghw \leq k$ is NP-complete even for $k = 2$ [3]. However, it was also shown in [3] that the problem of deciding if a given hypergraph has $ghw(H) \leq k$ becomes tractable for fixed k under realistic restrictions. One such restriction is the Bounded Intersection Property (BIP): a hypergraph H has *intersection width* $\leq i$ (denoted as $iwidth(H) \leq i$), if the intersection of any two edges in H has at most i vertices. A class of hypergraphs satisfies the BIP, if there exists a constant i, such that every hypergraph $H \in \mathcal{C}$ has $iwidth(H) \leq i$.

In [2], three different algorithms for checking $ghw(H) \leq k$ (and, if so, computing a concrete GHD of width $\leq k$) were implemented and tested on the HyperBench benchmark [2] comprising over 3,000 hypergraphs derived from CQs and CSPs from various sources. All the algorithms thus implemented rely on the observation that hypergraphs of CQs or CSPs stemming from applications tend to have low *iwidth*. These GHD-computations were purely sequential, even though one of the algorithms seems to lend itself naturally to parallel processing: more precisely, this GHD-algorithm is based on so-called "balanced separators"; at each step of the top-down construction of a GHD, this algorithm searches for a set of at most k edges $\{e_1, \ldots, e_k\}$ (= a "balanced separator"), such that the vertex set $e_1 \cup \cdots \cup e_k$ splits the hypergraph into components whose size (measured in terms of the edges that intersect with each component) is at most half the size of the component processed by the parent node in the GHD.

Given that many of the experiments reported in [2] had high run times or were even stopped due to a time out (with default value 3,600 seconds), we have to look for a different computation strategy. The goal of this work is to provide a parallel computation of GHDs and to move the GHD-computation to a powerful cluster. To this end, we adopt the aforementioned GHD-algorithm

* This work was supported by the Austrian Science Fund (FWF):P30930-N35.

based on balanced separators (referred to as "b-seps", for short, in the sequel) and implement it in the Go programming language [1]. Developed at Google in 2009, it has already seen widespread use by a number of companies such as Dropbox, CloudFare, Netflix and by Google itself.

Below, we describe the challenges that we have faced in designing a parallel implementation of the b-seps approach:

- Finding a good design of the main targets for parallelization, namely the search for the next balanced separator and the recursive calls of the GHD-computation for the resulting components;
- Finding a way to partition the work space equally among CPUs;
- Designing parallel search to support efficient backtracking;
- Splitting resources equally among the search space during recursive calls;
- Introducing *subedges* of the edges in the given hypergraph (which is needed to exploit the low *iwidth*(H)) while avoiding unneeded combinatorial explosion.

Below, we summarize how we have dealt with the above challenges and we report on first, promising experimental results. We will show that the parallel approach is indeed able to significantly speed up the expensive GHD-computations and that it allowed us to solve some cases which were out of reach with the previous, purely sequential GHD-implementations in [2].

2 Preliminaries

We assume the reader to be familiar with basic notions such as conjunctive queries (CQs) and their corresponding hypergraphs. 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 $\lambda(n) \subseteq E(H)$. We denote with $B(\lambda(n))$ the set $\{v \in V(H) \mid v \in e, e \in \lambda(n)\}$. The functions χ and λ have to satisfy the following conditions:

1. For each edge $e \in E(H)$ there exists a node $n \in N$ such that $e \subseteq \chi(n)$.
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* (ghw) is the size of the largest label for λ. It was shown in [2] that for a class of hypergraphs enjoying the BIP, one can add polynomially many subedges of edges in $E(H)$ to ensure $B(\lambda(n)) = \chi(n)$ for every $n \in N$.

For a set of edges $S \subseteq E(H)$, we say two vertices $v_1, v_2 \in V(H)$ are $[S]$-*connected* if there is a path of edges $e_1, \ldots, e_n \in E(H) \backslash S$ such that $v_1 \in e_1$ and $v_2 \in e_n$ and for each pair in the path $e_i, e_{i+1}, i \leq n-1$ we have that e_i and e_{i+1} share a common vertex. We define an $[S]$-*component* to be a maximal set of $[S]$-connected vertices. 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* (b-sep) if all $[S]$-components of H have a size of $\frac{|E(H)|}{2}$ or less .

graph	sequential	parallel	speedup
Dubois-016.xml.hg	668.097 ms	45.39 ms	14.71
rand-25-10-25-87-24.xml.hg	5936.83 ms	879.99 ms	7.84
Nonogram-012-table.xml.hg	73976.08 ms	11057.68 ms	6.69
Pi-20-10-20-30-17.xml.hg	1439.71 ms	200.54 ms	7.17

Table 1. Running times of b-seps algorithm for width 3.

It was shown in [2] 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: if no such separator exists, then clearly there can be no GHD of H of width k.

3 Results

The Go programming language [1] has a model of concurrency that is based on Hoare's Communicating Sequential Processes [5]. The smallest concurrent unit of computation is called a "goroutine", essentially a light-weight thread.

For the b-seps algorithm, we looked at two key areas of parallelization: the search for b-seps and the recursive calls. For the search, while testing out a number of configurations, we settled ultimately on using two types of goroutines: A number of workers, spawned via a central master goroutine, which starts them and waits on exactly two conditions (whichever happens first): either one of the workers finds a b-sep, or none of them do and they all terminate on their own. In the first case it makes sure all workers terminate and continues the main computation. For the recursive calls, we so far only spawn a single goroutine for each of them and wait for each call to return its result (a GHD of the corresponding subhypergraph).

To split the work equally among the workers during the search, a simple work balancer (each worker receiving jobs from the central master goroutine) would address this nicely. However, we found that this introduces a considerable synchronization delay when compared with splitting up the work beforehand. We assume here that the choice of edges has only small influence on the time needed to check if they form a b-sep. Thus we split the search space of size $M = \binom{N}{k}$ by the number of workers W (and if there is some remainder, we simply increase the workload of the first ($M \mod W$) workers by 1). Each worker has its own iterator, thus minimizing the need for communication during the search.

Our algorithm must support backtracking, i.e., restarting the search and finding another solution (if it exists). If we do not keep track carefully where each worker left off when the parallel search halted earlier, we could be forced to repeat work. We address this by saving the above mentioned iterators, used by each worker, in the main thread so as to reuse them during backtracking.

Another challenge lies in parallelizing the recursive calls in such a way that the resources (CPU cores) are split among them, to reduce unnecessary synchro-

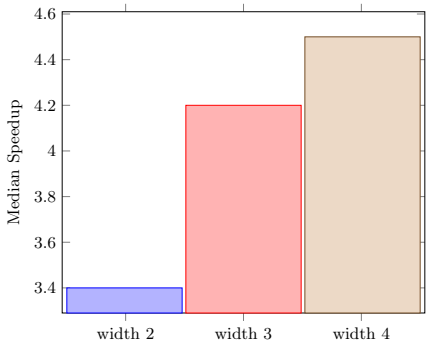

Figure 1. Median speedups for CSP Application instances from [3]

nization delays and not focus too much on one branch of the search tree. We have not yet found a satisfactory solution for this. One idea would be to go back to load balancing, and see if it helps with parallelizing the recursive calls, at the cost of taking away resources from the parallel search of b-seps.

The algorithm needs to compute *subedges* and consider them as possible choices for b-seps, as it would otherwise not be complete. The first and obvious choice is to add all possible subedges in the beginning (adding them globally). This leads to a combinatorial explosion, and more crucially also leads to many useless combinations, such as considering multiple subedges from the same original edge at once. We address this by adopting the local subedge variant from [3] and making it more restricted by first finding a balanced separator among the "original" edges of $E(H)$, and if this leads to a reject case, considering subedge variants of its edges. We also make sure not to repeat the same subedges by caching the vertex sets that generate them.

We used our parallel implementation to look at further instances that can be determined negatively (proving that no GHD of a certain width can exist) or positively (actually producing a GHD of that width). Our test setup was a cluster with 11 machines, each with two 12-core Intel Xeon CPU E5-2650 v4, clocked at 2.20GHz. For the tests, each job ran exclusively on a machine spawning up to 24 goroutines.

Possible speedups of four sample hypergraphs from HyberBench when compared to a sequential execution[1] are showcased in Table 1, where speedup is simply the sequential time divided by the parallel one. Furthermore, when comparing the execution times of the purely sequential version with parallelizing both the search for b-seps and parallelizing the recursive calls, we observe an increase in speedups at higher widths, seen in Figure 1. Additionally, we could produce new results for some previously unsolved instances of the HyperBench benchmark, thus determining their exact *ghw*. We are positive that with further

[1] 'Sequential execution' here refers to running the same general algorithm, but rewritten so that it does not use any goroutines. This version therefore only uses a single CPU core.

improvements, we will be able to "fill out the blank spots" on many hypergraphs of the HyperBench benchmark from [2] and in the process produce a more robust tool to compute GHDs.

4 Outlook

This paper is about work in progress. The next step will be to incorporate further optimizations into our Go implementation such as the following: (1) Applying various heuristics for ordering the hyperedges (to find b-seps faster), (2) caching of previous computations, and (3) looking at hybrid solutions which first apply the recursive splitting into subproblems via the b-seps approach and then (for sufficiently small subproblems) switch to one of the other (sequential) GHD-algorithms in [2]. The ultimate goal is, at least for all hypergraphs in the HyperBench benchmark where an upper bound on *ghw* of at most 6 is known (i.e., slightly more than 1,500 instances), to be able to compute the precise value of *ghw*. Currently, for about half of these instances, the exact *ghw* is still open.

References

1. Golang.org (Feb 2019), https://golang.org/
2. Fischl, W., Gottlob, G., Longo, D.M., Pichler, R.: Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In: PODS 2019 (to appear) (2019)
3. Fischl, W., Gottlob, G., Pichler, R.: General and fractional hypertree decompositions: Hard and easy cases. In: Proc. PODS 2018. pp. 17–32 (2018)
4. Gottlob, G., Greco, G., Leone, N., Scarcello, F.: Hypertree decompositions: Questions and answers. In: Proc. PODS 2016. pp. 57–74 (2016)
5. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)