Parallel Computations of Transversal Hypergraphs

Adam Murdock Gale[1 [0009−0008−1850−6006]] (✉) and Marc Moreno Maza[1 [0000−0003−3011−0756]]

Ontario Research Center for Computer Algebra, University of Western Ontario, London, Ontario, Canada
{agale9, mmorenom}@uwo.ca

1 Introduction

Hypergraph theory is a generalization of graph theory in which an edge can be any non-empty subset of the vertex set, and thus may contain more than two vertices. Given a hypergraph H, with vertex set V, a non-empty subset S of V is transversal to H if S has a non-empty intersection with every edge of H. The transversal hypergraph of H, denoted by $\mathrm{Tr}(H)$, is the set of all subsets of V which are transversal to H and inclusion-minimal with that property.

Over the past two decades, hypergraphs have gained significant attention and are used in areas like artificial intelligence [1,12,13,24], chemistry [18], computer vision [5], integrated circuits and VLSI [16], Physics [22], HPC and scheduling [6,9], data mining [15], social sciences [14,20,23], database theory [10], model-based diagnosis [10], and even philosophy [21], to name a few. In particular, the computation of transversal hypergraphs is commonly used in these applications of hypergraph theory.

Traditionally, $\mathrm{Tr}(H)$ is computed by a divide-and-conquer algorithm based on a formula due to Claude Berge [2]. This approach has been improved in terms of algebraic complexity [17] and cache complexity with parallelism in the fork-join model [19]. Recent studies suggest that the running time of Berge's algorithm is on average quasi-linear in the size of $\mathrm{Tr}(H)$ [8].

The size of $\mathrm{Tr}(H)$ can be exponential in the size of H, thus no algorithm computing $\mathrm{Tr}(H)$ can run in polynomial time. However, the question of whether $\mathrm{Tr}(H)$ can be computed in output-polynomial total time (i.e., in time polynomial in the combined sizes of the input and the output) is an open problem. Studying this latter problem has lead to another line of research, in particular by Thomas Eiter and Georg Gottlob [10], based on a completion algorithm for computing $\mathrm{Tr}(H)$. Eiter and Gottlob show that when the rank of H (i.e., the maximum edge size) is bounded by a constant k, $\mathrm{Tr}(H)$ can be computed in incremental-polynomial time, yielding an output-polynomial algorithm for this important subclass.

Furthermore, Eiter and Gottlob show that when H is β-acyclic (in the sense of Fagin [11]), $\mathrm{Tr}(H)$ can be computed in output-polynomial time by a divide-and-conquer algorithm (which we call BetaTr) that exploits the ear-decomposition structure of β-acyclic hypergraphs. This structural approach is particularly appealing because checking β-acyclicity can be done in polynomial time.

In this paper, we discuss these lines of research and present parallel and optimized C implementations of: (i) the divide-and-conquer algorithm based on Berge's formula (following [19]), (ii) the bounded-rank completion algorithm of Eiter and Gottlob [10], and (iii) the BetaTr algorithm for β-acyclic hypergraphs [10]. All implementations use bitvector edge encoding for efficient set operations and employ OpenCilk for task and data parallelism. The source code is available at `www.bpaslib.org`.

2 Transversal Hypergraphs

This section is a brief overview of the basic notions of hypergraph theory, with a focus on transversal hypergraphs and the main algorithms to compute them. For a detailed exposition we refer to the books of Claude Berge [3] and Alain Bretto [4]. We assume that the reader is familiar with the basic concepts of graph theory. In this paper, all hypergraphs are undirected and have no duplicate hyperedges. We start with the following definition.

A *hypergraph* is a pair $H = (V, E)$ where V is a non-empty finite set, the elements of which are called the *vertices* of H, and E is a set of non-empty subsets of V, called the *hyperedges* of H. The *degree* of a vertex $v \in V$ is the number of hyperedges e containing v. If a vertex has degree zero, then it is said to be *isolated*. The *rank* (resp. *anti-rank*) of H is the maximum (resp. minimum) cardinality of a hyperedge of H. A subset S of V is said *transversal* to H, whenever S has a non-empty intersection with every hyperedge of H.

For any hypergraph $H = (V, E)$, there are hypergraphs which we associate to H:

- The *complement hypergraph* of H is the hypergraph $\mathsf{Compl}(H) = (V, F)$ where $F = \{V \setminus e \mid e \in E\}$. We remark that this is only possible if H does not have V as a hyperedge since otherwise $\mathsf{Compl}(H)$ would have an empty hyperedge.
- The *minimal hypergraph* of H is the hypergraph $\mathsf{Min}(H) = (V, F)$ where F consists of all hyperedges of H which are minimal with respect to inclusion.
- The *maximal hypergraph* of H is the hypergraph $\mathsf{Max}(H) = (V, F)$ where F consists of all hyperedges of H which are maximal with respect to inclusion.
- The *transversal hypergraph* of H is is the hypergraph $\mathsf{Tr}(H) = (V, Z)$, where Z consists of all transversals of H that are minimal with respect to inclusion. It is easy to prove that the hypergraphs $\mathsf{Min}(H)$ and $\mathsf{Tr}(\mathsf{Tr}(H))$ are equal.

We illustrate the above notions with a small example.

Example 1. Consider the hypergraph $H = (V, \mathcal{E})$ with $V = \{a, b, c, d, e\}$ and $\mathcal{E} = \{E_1, E_2, E_3, E_4\}$ where $E_1 = \{a, b, c\}$, $E_2 = \{a, b\}$, $E_3 = \{b, c, d\}$, $E_4 = \{d, e\}$, depicted on the top-left of Figure 1. Since $E_2 \subset E_1$, H is not simple. The hypergraph $\mathsf{Max}(H)$ loses E_2 since it is contained in E_1; similarly $\mathsf{Min}(H)$ loses E_1 since it contains E_2. The transversal hypergraph $\mathsf{Tr}(H)$ is shown in the bottom center; every hyperedge of $\mathsf{Tr}(H)$ has a non-empty intersection with every edge of H and is inclusion-minimal with that property. The hypergraph H' (bottom-right) is isomorphic to H (via $a \leftrightarrow c$) but not equal to H.

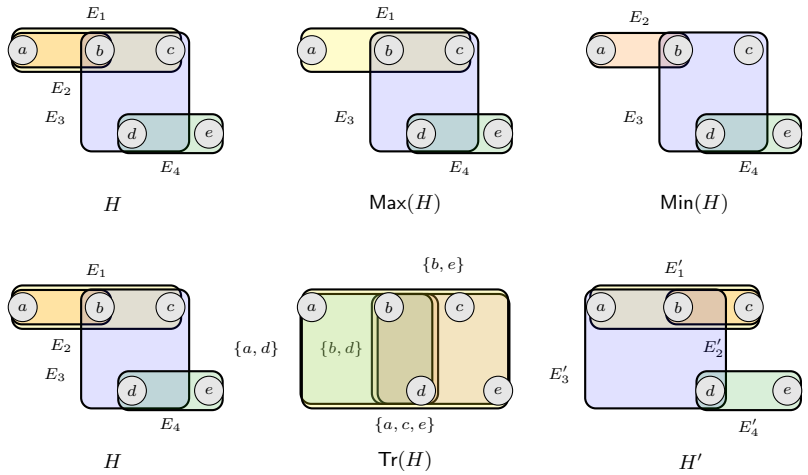

Fig. (1) Top row: H (left), $\mathsf{Max}(H)$ (middle), $\mathsf{Min}(H)$ (right). Bottom row: H (left), $\mathsf{Tr}(H)$ (middle), H' isomorphic to H via $a \leftrightarrow c$ (right).

Computing $\mathsf{Min}(H)$ and $\mathsf{Max}(H)$ is a well-studied question in the more general context of partially ordered sets (posets). For a poset $\mathcal{P} = (P, \prec)$ with $n = |P|$ elements and maximum anti-chain length ℓ, $\mathsf{Min}(\mathcal{P})$ can be computed in $O(\ell n)$ time [7]. A cache-optimal parallel algorithm for $\mathsf{Min}(\mathcal{P})$ in the fork-join model is presented in [19]. A poset is *ranked* if there exists a rank function $\mathsf{rank}(\cdot)$ such that $\mathsf{rank}(x) < \mathsf{rank}(y)$ implies $y \not\prec x$. The poset $(2^V, \subset)$ is ranked by set cardinality, and the algorithm of [19] can be further optimized for ranked posets when rank comparisons are cheaper than inclusion tests.

2.1 The Divide-and-Conquer Algorithm Based on Berge's Formula

Computing $\mathsf{Tr}(H)$ is another well-studied question. We start with the simple procedure proposed by Claude Berge in [2] by means of two elementary observations:

1. If the hypergraph $H = (V, E)$ has a single hyperedge e, then $\mathsf{Tr}(H) = (V, Z)$, where $Z = \{\{v\} \mid v \in e\}$.
2. If E and E' are two disjoint subsets of $2^V \setminus \emptyset$ then the transversal hypergraph of $H = (V, E \cup E')$ can be computed by the following formula

$$\mathsf{Tr}(H) \quad = \quad \mathsf{Min}(\ \mathsf{Tr}((V, E)) \vee \ \mathsf{Tr}((V, E'))), \tag{1}$$

where, for two hypergraphs $T = (V, E)$ and $T' = (V', E')$ the operation $T \vee T'$ is defined as follows:

$$T \vee T' \quad = \quad (V \cup V', \{e \cup e' \mid (e, e') \in E \times E'\}). \tag{2}$$

Together, these observations yield a divide-and-conquer algorithm. Given $H = (V, \mathcal{E})$ with $m = |\mathcal{E}|$ edges, we split $\mathcal{E}$ into halves $\mathcal{E}_L$ and $\mathcal{E}_R$, recursively compute their transversals, and combine via Equation (1). The base case $m = 1$ is handled by observation (1). When $m = 2$, the two edges E_1 and E_2 are handled directly: $\mathsf{Tr}((V, \{E_1, E_2\})) = \{\{x\} \mid x \in E_1 \cap E_2\} \cup \{\{x, y\} \mid x \in E_1 \setminus E_2,\ y \in E_2 \setminus E_1\}$. This binary splitting continues until one of these base cases is reached, producing a recursion tree of depth $O(\log m)$. Importantly, the two recursive calls at each level are independent, making them natural candidates for parallel execution.

2.2 The Completion Algorithm of Eiter and Gotlob

The completion algorithm of Eiter and Gotlob [10] provides an output-polynomial method for computing $\mathsf{Tr}(H)$ when the rank $r(H) = \max\{|E| : E \in H\}$ is bounded by a constant. The algorithm maintains a set $\mathcal{G}$ which is gradually extended until $\mathcal{G} = \mathsf{Tr}(H)$. At each step, $\mathcal{G} \subseteq \mathrm{Tr}(H)$ holds as an invariant.

The algorithm relies on the following characterization. Let H be a hypergraph with rank $r(H) \leq k$ for some constant $k \geq 0$. Then $\mathcal{G} = \mathrm{Tr}(H)$ if and only if $\mathbf{P} \wedge \mathbf{Q}$ holds, where:

- **P**: $\mathrm{Tr}(\mathcal{G})|_r \subseteq H$, meaning every edge of $\mathrm{Tr}(\mathcal{G})$ with cardinality at most r is an edge of H;
- **Q**: there exists no $\mathcal{G}' \subseteq \mathcal{G}$ with $|\mathcal{G}'| = r + 1$ such that for all $E \in \mathcal{G}$, $E \not\supseteq \{x \in V : d(x, \mathcal{G}') > 1\}$, where $d(x, \mathcal{G}')$ denotes the number of edges of $\mathcal{G}'$ containing x.

When either condition fails, a witness can be extracted and minimized to obtain a new $T_1 \in \mathrm{Tr}(H) \setminus \mathcal{G}$, which is added to $\mathcal{G}$. Each iteration runs in polynomial time and produces one additional minimal transversal, so $\mathrm{Tr}(H)$ is computed in incremental-polynomial time when $r(H) \leq k$.

In practice, the condition **P** corresponds to enumerating subsets of V of cardinality at most r and checking whether each is a transversal of $\mathcal{G}$ that is not already in H. This is the **C1** phase of the algorithm. The condition **Q** corresponds to searching over $(r+1)$-subsets of $\mathcal{G}$ and checking whether a certain vertex set is a transversal of H not covered by $\mathcal{G}$. This is the **C2** phase.

2.3 The Divide-and-Conquer Algorithm of Eiter and Gotlob for β-Acyclic Hypergraphs

Several notions of acyclicity for hypergraphs have been proposed. Following Fagin [11], we consider α-, β-, γ-, and *Berge*-acyclicity, forming the proper inclusion hierarchy: Berge-acyclic $\Rightarrow$ γ-acyclic $\Rightarrow$ β-acyclic $\Rightarrow$ α-acyclic. A hypergraph H is α-acyclic if it can be reduced to the empty hypergraph by the Graham–Yu–Ozsoyoglu (GYO) reduction, which repeatedly (1) removes a vertex v occurring in only one edge, and (2) removes an edge E' contained in another edge E. A hypergraph is β-acyclic if every sub-hypergraph is α-acyclic.

The key structural concept is the *ear node*: a vertex v appearing in exactly one edge. Every simple β-acyclic hypergraph with a nonempty edge has an ear node [10], and β-acyclicity can be checked in polynomial time by repeatedly removing ear nodes and simplifying.

Eiter and Gotlob exploit this ear-decomposition structure to compute $\mathsf{Tr}(H)$ for β-acyclic hypergraphs via the following divide-and-conquer algorithm.

Algorithm 1: BetaTr(H)

Input: Simple β-acyclic hypergraph $H = (V, \mathcal{E})$
Output: $\mathrm{Tr}(H)$
if $\mathcal{E} = \emptyset$ **then**
  return $\{\emptyset\}$
if $\emptyset \in \mathcal{E}$ **then**
  return $\emptyset$
Find an ear node v appearing in exactly one edge $E \in \mathcal{E}$
$H_1 \leftarrow \mathsf{Min}(\{F \cap (V \setminus E) \mid F \in \mathcal{E}\} \setminus \{\emptyset\})$
$H_2 \leftarrow \mathsf{Min}((\mathcal{E} \setminus \{E\}) \cup \{E \setminus \{v\}\})$
$T_1 \leftarrow$ BetaTr(H_1)
$T_2 \leftarrow$ BetaTr(H_2)
return $\{\{v\} \cup T : T \in T_1\} \cup T_2$

Since β-acyclicity is preserved under sub-hypergraphs, H_1 and H_2 are again β-acyclic and the recursion is well-founded. The total number of recursive calls is $O(v_e(H) \cdot |\mathrm{Tr}(H)|)$, where $v_e(H)$ is the number of essential vertices, so the overall running time is bounded by $q(m, n) \cdot (|\mathrm{Tr}(H)|+1)$ for some polynomial q. Thus BetaTr computes $\mathrm{Tr}(H)$ in output-polynomial time for β-acyclic hypergraphs [10].

3 Parallel Algorithms for Transversal Hypergraph Computations

All our implementations encode hyperedges as bitvector arrays: bit i of word $\lfloor i/64 \rfloor$ records membership of vertex i, so that intersection, union, and subset tests reduce to word-level bitwise operations in $O(\lceil |V|/64 \rceil)$ time per edge. Parallelism is provided by OpenCilk, a lightweight fork-join framework with work-stealing scheduling.

3.1 Parallelizing Berge's Formula

Following [19], we implement the divide-and-conquer algorithm of Section 2 with two levels of parallelism.

Task Parallelism in the Recursion. When the edge set $\mathcal{E}$ has more than a threshold number of edges (we use 32 in practice), it is split into two halves $\mathcal{E}_L$ and $\mathcal{E}_R$. The recursive calls $\mathsf{Tr}((V, \mathcal{E}_L))$ and $\mathsf{Tr}((V, \mathcal{E}_R))$ are independent and are executed concurrently via `cilk_spawn` and `cilk_sync`. Below the threshold, a serial base case is used.

Parallelism in the Merge. The merge step forms all pairwise unions of transversals from the two subproblems and removes non-minimal elements. The computation of $\mathsf{Min}()$ is itself parallelized using the cache-optimal algorithm of [19], which uses a four-way divide-and-conquer with a 2+sync+2 parallelization pattern.

3.2 Parallelizing Eiter and Gotlob's Completion Algorithm

The completion algorithm of Section 2.2 has a serial main loop: each iteration either extends $\mathcal{G}$ by one minimal transversal or terminates. Within each iteration, however, both the C1 and C2 phases expose substantial data parallelism.

Parallel C1 (Subset Enumeration). The C1 phase enumerates all k-subsets of V (for $k = 1, \ldots, r$) and tests whether each is a transversal of $\mathcal{G}$ not already in H. We pre-generate the $\binom{|V|}{k}$ candidate subsets and distribute them across workers using `cilk_for`. Each worker tests candidates independently against a read-only snapshot of $\mathcal{G}$. An `std::atomic<bool>` flag provides early termination: once any worker finds a valid candidate, all others skip their remaining work.

Parallel C2 (Vertex-Subset Search). The C2 phase searches over subsets of $\mathcal{G}$ for a witness that **Q** fails. In our implementation, the search space is divided into chunks that are distributed via `cilk_for`. A resume bookmark persists across iterations so that later calls only scan the residual range, reducing chunk count and avoiding unnecessary spawn overhead. As in C1, the first valid witness is stored via atomic compare-and-exchange.

3.3 Parallelizing BetaTr

The BetaTr algorithm of Section 2.3 has a natural recursive structure: at each step, two independent subproblems H_1 and H_2 are constructed and solved recursively, analogously to the fork-join pattern in Berge's divide-and-conquer. We parallelize these recursive calls using `cilk_spawn` and `cilk_sync`.

However, the two subproblems are typically unbalanced. The subproblem H_1 (projecting edges onto $V \setminus E$) tends to be smaller, while H_2 (removing a single vertex from the ear edge) retains most of the structure of H. This imbalance limits the effective parallelism exposed by the recursion.

4 Experimentation

All experiments were conducted on an Intel i7-1185G7 (4 cores, 8 threads) running Linux, compiled with OpenCilk-enabled `clang++` at optimization level `-O2`. We compare three algorithms: the parallel divide-and-conquer based on Berge's formula (trvgen), the parallelized bounded-rank completion algorithm (Completion), and the BetaTr algorithm for β-acyclic inputs.

4.1 Data Sets

We use the following families of hypergraphs:

- $K_{n,k}$: the complete k-uniform hypergraph on n vertices, consisting of all $\binom{n}{k}$ subsets of size k. These have bounded rank and dense edge sets.
- L_d: the circuit hypergraph of a lattice matroid of dimension d, ordered in colexicographic order. These have moderate vertex counts but large edge sets.
- m_N: perfect matching hypergraphs with N vertices and $N/2$ disjoint edges of size 2. These are β-acyclic, sparse, and have $2^{N/2}$ minimal transversals.

4.2 Divide-and-Conquer vs. Completion

Table (1) trvgen (8 threads) vs. Completion, serial and parallel. Times in seconds.

Input	$\lvert V\rvert$	$\lvert\mathcal{E}\rvert$	trvgen	Compl. (serial)	Compl. (parallel)
$K_{14,7}$	14	3 432	0.003	12.37	5.14
L_6	21	1 237	0.029	15.19	8.14

For these inputs, trvgen is substantially faster than Completion. The parallel Completion achieves 2.4× speedup over the serial version on $K_{14,7}$ (C1-dominated) and 1.9× on L_6 (C2-dominated). However, candidates in the completion algorithm are enumerated serially (each iteration depends on the previous result), which limits overall parallel speedup.

4.3 Edge Ordering Sensitivity

The divide-and-conquer algorithm splits the edge list at the midpoint, so edge ordering strongly affects intermediate transversal sizes. Colexicographic (colex) ordering keeps related edges together, producing small intermediates. In contrast, the completion algorithm processes edges incrementally against the growing output $\mathcal{G}$, so its per-iteration cost depends on $|\mathcal{G}|$ and $|V|$ but not on edge order. Table 2 confirms this: on L_6, a single unlucky shuffle causes a $> 10^6\times$ slowdown for TRVGEN, while COMPLETION is completely unaffected. We therefore recommend a cheap $O(|\mathcal{E}| \log |\mathcal{E}|)$ colex sort as a preprocessing step before invoking TRVGEN.

Table (2) Effect of edge ordering on running time (seconds).

Input	Ordering	TRVGEN	COMPLETION
L_5 ($\|V\|$=15)	Colex	0.002	0.150
	Reversed	0.002	0.134
	Shuffled (avg)	0.117	0.132
L_6 ($\|V\|$=21)	Colex	0.029	15.91
	Reversed	0.027	15.57
	Shuffled (worst)	79 152	15.19

4.4 BETATR on β-Acyclic Inputs

Table 3 compares BETATR with TRVGEN on perfect matching hypergraphs, which are β-acyclic.

Table (3) BETATR vs. TRVGEN on perfect matchings.

Input	$\|V\|$	$\|\mathcal{E}\|$	$\|\mathrm{Tr}\|$	TRVGEN	BETATR
m_{24}	24	12	4 096	0.051 s	0.007 s
m_{28}	28	14	16 384	0.531 s	0.028 s
m_{30}	30	15	32 768	2.23 s	0.060 s
m_{34}	34	17	131 072	35.6 s	0.234 s
m_{36}	36	18	262 144	>120 s	0.473 s
m_{40}	40	20	1 048 576	>120 s	1.04 s

BETATR achieves speedups of up to 253× over TRVGEN. The advantage grows with the number of vertices because BETATR runs in time polynomial in $|\mathrm{Tr}(H)|$ for β-acyclic inputs, while TRVGEN's running time depends on the sizes of intermediate results, which grow combinatorially. However, on dense inputs (where $|\mathcal{E}| \gg |V|$), the $O(|\mathcal{E}|^2)$ cost of the simplification step in BETATR can dominate, making TRVGEN preferable.

4.5 Dispatcher Strategy

Based on these experiments, we propose an algorithm dispatcher that selects the best method for a given input:

1. If H is simple, $|\mathcal{E}| \leq |V|$, and H is β-acyclic, use BETATR.
2. Otherwise, sort $\mathcal{E}$ into colexicographic order and use TRVGEN.

The β-acyclicity check runs in $O(|V|^2 \cdot |\mathcal{E}|)$ time, which is negligible compared to transversal computation.

5 Concluding Remarks

We have presented parallel and optimized C implementations of three complementary algorithms for computing transversal hypergraphs: (i) the divide-and-conquer algorithm based on Berge's

formula, with task parallelism via recursive fork-join and a cache-optimal parallel merge; (ii) the bounded-rank completion algorithm of Eiter and Gotlob, parallelized for the first time using data-parallel subset enumeration and vertex-subset search; and (iii) the BetaTr algorithm for β-acyclic hypergraphs, which exploits ear-node decomposition to achieve output-polynomial running time.

Our experiments show that each algorithm has a distinct performance profile: trvgen is the fastest general-purpose method when edges are well-ordered, but it is highly sensitive to edge ordering; the completion algorithm is immune to ordering effects but limited by serial candidate enumeration; and BetaTr achieves up to 253× speedup over trvgen on sparse β-acyclic inputs. A simple dispatcher that checks β-acyclicity and pre-sorts edges can select the best algorithm automatically.

Several directions for future work are worth pursuing. First, the three algorithms can potentially be composed: for a general hypergraph H, one can identify the β-acyclic components of H (if any exist), apply BetaTr to those components, and use trvgen or the completion algorithm for the remainder. Second, the parallelization of BetaTr can be improved by addressing the load imbalance that arises when the two subproblems H_1 and H_2 differ greatly in size. Third, it would be interesting to investigate whether the completion algorithm can be made more parallel by parallelizing the data enumeration within and across iterations.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bai, S., Zhang, F., Torr, P.H.: Hypergraph convolution and hypergraph attention. Pattern Recognition **110**, 107637 (2021). https://doi.org/10.1016/j.patcog.2020.107637
2. Berge, C.: Hypergraphes: combinatoire des ensembles finis. Dunod, Presses Universitaires de France, Paris (1987)
3. Berge, C.: Hypergraphs - combinatorics of finite sets, North-Holland mathematical library, vol. 45. North-Holland (1989)
4. Bretto, A.: Hypergraph theory. Springer (2013)
5. Bretto, A., Cherifi, H., Aboutajdine, D.: Hypergraph imaging: an overview. Pattern Recognition **35**(3), 651–658 (2002)
6. Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdağ, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing **69**(8), 711–724 (2009)
7. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S.J., Verbin, E.: Sorting and selection in posets. SIAM J. Comput. **40**(3), 597–622 (2011). https://doi.org/10.1137/070697720, https://doi.org/10.1137/070697720
8. David, J., Gholami, M., Lhote, L.: On the average-case complexity of Berge algorithm. Theoretical Computer Science **1064**, 115702 (2026). https://doi.org/10.1016/j.tcs.2025.115702
9. Devine, K.D., Boman, E.G., Heaphy, R.T., Bisseling, R.H., Catalyurek, U.V.: Parallel hypergraph partitioning for scientific computing. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. pp. 10–pp. IEEE (2006)
10. Eiter, T., Gottlob, G.: Identifying the minimal transversals of a hypergraph and related problems. SIAM Journal on Computing **24**(6), 1278–1304 (1995)
11. Fagin, R.: Degrees of acyclicity for hypergraphs and relational database schemes. Journal of the ACM **30**(3), 514–550 (1983). https://doi.org/10.1145/2402.322390
12. Feng, Y., You, H., Zhang, Z., Ji, R., Gao, Y.: Hypergraph neural networks. In: Proceedings of the AAAI conference on artificial intelligence. vol. 33, pp. 3558–3565 (2019)
13. Gao, Y., Zhang, Z., Lin, H., Zhao, X., Du, S., Zou, C.: Hypergraph learning: Methods and practices. IEEE Transactions on Pattern Analysis and Machine Intelligence **44**(5), 2548–2566 (2022)
14. Ghoshal, G., Zlatić, V., Caldarelli, G., Newman, M.E.: Random hypergraphs and their applications. Physical Review E **79**(6), 066118 (2009)
15. Gunopulos, D., Mannila, H., Khardon, R., Toivonen, H.: Data mining, hypergraph transversals, and machine learning. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. pp. 209–216 (1997)

16. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multilevel hypergraph partitioning: Application in vlsi domain. In: Proceedings of the 34th annual Design Automation Conference. pp. 526–529 (1997)
17. Kavvadias, D.J., Stavropoulos, E.C.: An efficient algorithm for the transversal hypergraph generation. J. Graph Algorithms Appl. **9**(2), 239–264 (2005). https://doi.org/10.7155/JGAA.00107, https://doi.org/10.7155/jgaa.00107
18. Konstantinova, E., Skorobogatov, V.: Application of hypergraph theory in chemistry. Discrete Mathematics **235**(1), 365–383 (2001). https://doi.org/10.1016/S0012-365X(00)00290-9, chech and Slovak 3
19. Leiserson, C.E., Moreno Maza, M., Li, L., Xie, Y.: Parallel computation of the minimal elements of a poset. In: Moreno Maza, M., Roch, J. (eds.) Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France. pp. 53–62. ACM (2010). https://doi.org/10.1145/1837210.1837221, https://doi.org/10.1145/1837210.1837221
20. Lin, Y.R., Sun, J., Castro, P., Konuru, R., Sundaram, H., Kelliher, A.: Metafac: community discovery via relational hypergraph factorization. In: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 527–536 (2009)
21. Nicholson, T.L.D.: Philosophical hypergraphics: some applications to philosophy of the theory of hypergraphs. Ph.D. thesis, Simon Fraser University (2007)
22. Rossi, M., Huber, M., Bruß, D., Macchiavello, C.: Quantum hypergraph states. New Journal of Physics **15**(11), 113022 (2013)
23. Zhang, Z.K., Liu, C.: A hypergraph model of social tagging networks. Journal of Statistical Mechanics: Theory and Experiment **2010**(10), P10005 (oct 2010)
24. Zhou, D., Huang, J., Schölkopf, B.: Learning with hypergraphs: Clustering, classification, and embedding. In: Schölkopf, B., Platt, J., Hoffman, T. (eds.) Advances in Neural Information Processing Systems. vol. 19. MIT Press (2006)