Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

An Embarrassingly Parallel Model Counter

Zhenghang Xu[1,2] , **Minghao Yin**[1,2] , **Jean-Marie Lagniez**[3]

[1]School of Information Science and Technology, Northeast Normal University, Changchun, China
[2]Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China
[3]Univ. Artois, CNRS, CRIL, F-62300 Lens, France
{xuzh121, ymh}@nenu.edu.cn, lagniez@cril.fr

Abstract

Model counting (also known as #SAT) is a fundamental problem in knowledge representation and reasoning, with applications ranging from probabilistic inference to formal verification. However, state-of-the-art model counters are limited by computational resources on a single machine. In this paper, we propose a novel distributed framework for model counting, exploiting the embarrassingly parallel nature of the problem. By decomposing the search space into independent subproblems and distributing them across different computation nodes, our approach achieves near-linear scalability on practical instances. Extensive experiments on standard benchmarks demonstrate both the effectiveness and efficiency of our framework.

1 Introduction

Model counting (MC) is the problem of computing the number of models (i.e., satisfying assignments) of a given propositional formula (typically in CNF) over its set of variables. Its direct extension, *weighted model counting* (WMC),[1] is of tremendous importance in a wide range of AI applications, including probabilistic inference (Sang, Beame, and Kautz 2005b; Chavira and Darwiche 2008; Dubray, Schaus, and Nijssen 2023), planning (Palacios et al. 2005; Domshlak and Hoffmann 2006), neural network verification (Baluta et al. 2019), and explainable AI (Izza et al. 2023). Model counting also finds key applications outside AI, notably in model checking and hardware testing (Heinz and Sachenbacher 2009; Feiten et al. 2012; Teuber and Weigl 2021; Dueñas-Osorio et al. 2017; Girol, Farinier, and Bardin 2021; Mei, Bonsangue, and Laarman 2024).

The computational power and difficulty of model counting are well-reflected by Toda's theorem, which shows that the entire polynomial hierarchy (PH) is contained in $\mathsf{P}^{\#\mathsf{P}}$; that is, every problem in the polynomial hierarchy can be solved in polynomial time, given access to a #P oracle (Toda 1991). The centrality of the #SAT problem has thus motivated significant research in designing both exact and approximate model counting algorithms capable of handling ever-larger instances, as highlighted in recent model counting competitions (https://mccompetition.org/).

A variety of (sequential, exact) model counters have been introduced and studied, including search-driven approaches such as `cachet` (Sang et al. 2004), `SharpSAT` (Thurley 2006), `SharpSAT-TD` (Korhonen and Järvisalo 2021), and `ganak` (Sharma et al. 2019), as well as compilation-based counters like `C2D` (Darwiche 2004), `SDD` (Oztok and Darwiche 2015), `dSharp` (Muise et al. 2012), `eadt` (Koriche et al. 2013), and `d4` (Lagniez and Marquis 2017a). However, model counting remains a #P-complete problem, typically much harder in practice than SAT, meaning that many real-world instances remain out of reach for current solvers.

To address more instances within reasonable time, a straightforward technological strategy is to run existing model counters on increasingly powerful processing units. Historically, improvements in processor speed were driven by raising clock frequencies through greater transistor densities on silicon chips, following Moore's Law. Moore's Law, articulated in the mid-1960s, states that the number of transistors in a dense integrated circuit doubles approximately every two years, a trend that held true for several decades. However, physical limitations, such as the atomic scale, are expected to eventually halt this progress, with most experts (including Gordon Moore himself) predicting the end of Moore's Law around 2025. Indeed, in recent years the pace of dimensional scaling (known as Dennard scaling) has noticeably slowed.

To cope with the constraints on single-core performance and manage power dissipation, processor manufacturers have shifted towards multi-core chip designs. While this architectural change requires software to be adapted for parallel execution, multi-threaded parallelism has been successfully leveraged in some model counters. For example, the parallel model counter `CountAntom` (Burchard, Schubert, and Becker 2015) uses multiple threads to compute the model count concurrently, sharing both learned conflict clauses and cached sub-formula results. `CountAntom`'s "laissez-faire caching" scheme allows cores to access a shared cache, ensuring that sub-formula model counts are accurate, even though clause learning during tree search generally complicates parallel cache correctness.

However, the same physical integration limits that restrict clock speed increases also constrain the number of cores that

[1]In WMC, each literal is assigned a real-valued weight. The weight of an interpretation is the product of the weights of the literals set to true, and the weight of the formula is the sum of the weights of its models. WMC generalizes ordinary model counting, which corresponds to the case where all literal weights are 1.

can be packed onto a single chip. In practice, commercially available many-core processors can feature hundreds, but not thousands, of cores. Therefore, extending parallelism across multiple (possibly heterogeneous) networked computers, known as *distributed parallelism*, is the next logical progression. Unlike shared-memory multi-threaded systems, distributed environments lack efficient global communication, requiring message passing between processes.

A landmark in the evolution of distributed model counting is `dCountAntom` (Burchard, Schubert, and Becker 2016), which augments `CountAntom` with a master node that orchestrates the distribution of subproblems to worker nodes using message passing. However, the hierarchical structure of `dCountAntom` restricts flexibility, as worker nodes are unable to share work directly with each other. This limitation is overcome by `dmc` (Lagniez, Marquis, and Szczepanski 2018), which employs a work stealing paradigm: tasks are balanced dynamically among peers, with idle workers able to request additional work. To control communication overhead, `dmc` restricts message number and size, avoiding explicit transfers of learnt clauses or subproblems.

Nevertheless, work stealing strategies are often *intrusive*, as they typically require substantial changes to the core model counting architecture, posing technical challenges and limiting general applicability. This challenge motivates the need for simpler, more broadly applicable distributed solutions. In this paper, we present an embarrassingly parallel approach to model counting, called `DisCount` (Distributed Counting), enabling efficient distributed computation while requiring minimal changes to existing solvers. Our method decomposes the original formula into a large number of independent subproblems by selecting a subset of variables, enumerating all consistent assignments to these variables (validated using a SAT solver), and then distributing each subproblem to an available worker (such as a processor core or cluster node). The decomposition process leverages the variable selection method guided by tree decompositions as proposed in (Bannach and Hecher 2024b). This approach yields high scalability and efficiency, our experiments on recent model counting competition benchmarks demonstrate that generating around thirty subproblems per worker achieves notable speedup. Furthermore, we show that our solution performs competitively with work stealing approaches like `dmc`, often solving more instances. Importantly, our technique allows users to harness parallelism with virtually no modifications to model counters and without the need to develop parallel code.

The remainder of the paper is structured as follows. Section 2 lays out the formal background. Section 4 presents the related works. Section 3 details the proposed `DisCount` approach. Section 5 reports our experimental findings, and Section 6 concludes with a summary and perspectives for future work.

2 Preliminaries

Let $\mathcal{L}$ represent the propositional language formed from a finite set $\mathcal{P}$ of propositional variables, using the standard logical connectives ($\neg$, $\vee$, $\wedge$) and the Boolean constants $\top$ and $\bot$. A literal ℓ is either a variable $x \in \mathcal{P}$ or its negation $\neg x$. A term, also called cube, is a conjunction of literals, and a clause is a disjunction of literals. Terms and clauses may, when convenient, be regarded as sets of literals. A CNF formula is a conjunction of clauses, also viewed as the set of its clauses.

Formulas are interpreted in the classical way: an interpretation ω is a mapping from $\mathcal{P}$ to $\{0,1\}$. An interpretation ω is a model of a formula Σ if Σ evaluates to 1 (true) under ω. Interpretations can equivalently be represented by the set of literals they satisfy. We use $\models$ to denote logical entailment and $\equiv$ for logical equivalence. For any formula $\Sigma \in \mathcal{L}$, we let $Var(\Sigma)$ denote the set of variables from $\mathcal{P}$ occurring in Σ. Boolean Constraint Propagation (BCP) is a fundamental procedure used in many preprocessors and solvers. $\mathrm{BCP}(\Sigma)$ returns $\{\emptyset\}$ if unit propagation on the clauses of the CNF formula Σ derives the empty clause (that is, a unit refutation), and otherwise returns the set of literals (from unit clauses) deduced from Σ by unit propagation.

Example 1. *Consider the CNF formula Σ over variables $Var(\Sigma) = \{x_1, \ldots, x_6\}$ with clauses: $\{\neg x_2 \vee x_3, x_3 \vee \neg x_6, x_5 \vee x_6, x_1 \vee \neg x_2 \vee x_5, x_1 \vee \neg x_4\}$. Given the term $\tau = \neg x_1, x_6$, applying BCP to the formula Σ augmented with the assignments in τ, i.e., to $\Sigma \wedge \tau$, results in $\mathrm{BCP}(\Sigma \wedge \tau) = \{\neg x_4, x_3\}$.*

The SAT problem asks whether a given CNF is satisfiable, that is, whether there exists a model of the formula. Conflict-Driven Clause Learning (CDCL) SAT solvers are state-of-the-art algorithms for solving SAT (Marques-Silva, Lynce, and Malik 2021). CDCL incrementally builds a variable assignment and applies BCP. Upon encountering a conflict, the solver performs conflict analysis to identify the cause and learns a new clause that prevents the same conflict from arising again. This learnt clause is added to the formula, effectively pruning the search space. During conflict analysis, the variables (literals) involved have their activity scores "bumped," increasing their likelihood of being chosen in subsequent branching decisions. This dynamic variable ordering focuses the search on the most relevant parts of the formula. The solver then backtracks non-chronologically, to a previous decision level, and this process repeats until the formula is determined to be satisfiable or unsatisfiable.

The notation $\|\Sigma\|$ denotes the number of models (i.e., satisfying assignments) of the formula Σ over its variables $Var(\Sigma)$. The *#SAT problem* asks for the total number of such models for a given Boolean formula, and is the canonical example of a #P-complete problem.

Example 2 (Example 1 cont'ed)**.** *The number of models of Σ, as defined in Example 1, is $\|\Sigma\| = 20$.*

Since model counting is closely related to the SAT problem, one of the most effective strategies builds upon DPLL-based SAT solvers (Gomes, Sabharwal, and Selman 2021). These solvers are extended to not only determine satisfiability but also compute the exact number of satisfying assignments. Specifically, model counters based on this approach explore the search space recursively by branching on variables and simplifying the formula at each step (mainly using BCP). However, unlike SAT solvers, which terminate upon finding a single satisfying assignment, model counters must

exhaustively enumerate all satisfying assignments, making the problem substantially more challenging.

A key enhancement in DPLL-based model counting is the use of connected component decomposition: after simplifying the formula in the current branch, the remaining formula may split into independent subformulas (connected components) that do not share variables. Since these subproblems are independent, their model counts can be computed separately and multiplied, significantly reducing redundant work (Sang et al. 2004).

Another crucial optimization is component caching: since the same subproblem may arise multiple times in different parts of the search tree, model counters store the counts of previously solved components in a cache (Sang et al. 2004). When the same component reappears, the solver can simply retrieve the cached result instead of recomputing it. These two techniques, decomposing the problem and caching partial results, are essential for scaling model counting to large, structured instances, and form the basis of state-of-the-art exact model counters (Sang et al. 2004; Thurley 2006; Korhonen and Järvisalo 2021; Sharma et al. 2019; Lagniez and Marquis 2017a).

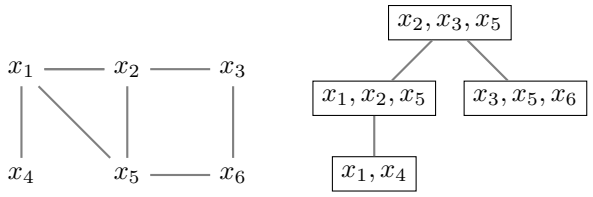
Despite these optimizations, the choice of which variable to branch on next remains critical to practical performance. The structure of the formula, as captured by its primal graph, can provide valuable insights for designing effective heuristics. In particular, leveraging tree decompositions of the primal graph enables the solver to exploit the problem's structure, potentially reducing the effective search space and leading to more efficient model counting.

A graph $G = (V, E)$ consists of a set of vertices $V = V(G)$ and a set of edges $E = E(G)$. A *tree* is a connected graph T such that $|E(T)| = |V(T)| - 1$. The *primal graph* of a CNF formula Σ is the undirected graph whose vertices correspond to the variables of Σ, with an edge between any two variables that appear together in some clause of Σ. Each connected component of the primal graph (i.e., a maximal set of vertices that are pairwise connected by paths) determines a subset of clauses of Σ, which we refer to as a connected component of the formula Σ.

Given an undirected graph $G = (V, E)$, a *tree decomposition* (Robertson and Seymour 1986; Bodlaender 2005) of G is a pair $(T, \{B_t\}_{t \in V(T)})$, where T is a tree and each node $t \in V(T)$ is associated with a subset (called a *bag*) $B_t \subseteq V$, such that:

1. For every vertex $v \in V$, there is at least one node $t \in V(T)$ with $v \in B_t$.
2. For every edge $\{u, v\} \in E$, there is at least one node $t \in V(T)$ with $\{u, v\} \subseteq B_t$.
3. For every $v \in V$, the set $\{t \in V(T) \mid v \in B_t\}$ induces a connected subtree of T.

The *width* of a tree decomposition T is defined as $w(T) = \max_{t \in V(T)} |B_t| - 1$. The *treewidth* of a graph G is the minimum width among all possible tree decompositions of G. Without loss of generality, we may assume that one node of T is designated as the *root* of the decomposition, chosen arbitrarily. For any node $t \in V(T)$, we denote by $d_T(t)$ the distance from the root to t, that is, the depth of t in T.



(a) Primal graph (b) Tree decomposition

Figure 1: The primal graph of the CNF formula given in Example 1 (left), and one of tree decompositions of the primal graph (right).

Example 3 (Example 1 cont'ed)**.** *Figure 1a illustrates the primal graph of the CNF formula presented in Example 1, while Figure 1b shows one possible tree decomposition of this primal graph.*

The notion of tree decomposition plays a central role in model counting, as it enables dynamic programming algorithms whose complexity is exponential in the tree-width k of the decomposition, rather than in the total number of variables. Specifically, for a tree decomposition of width k, model counting can be performed in time $\mathcal{O}(2^k \cdot n)$, where n is the input size. This makes the problem fixed-parameter tractable with respect to tree-width. Thus, for formulas whose associated graphs have small tree-width, model counting can be carried out efficiently, in contrast to the general #P-completeness of the task.

However, in many practical cases, the tree-width of the problem instance is too large for tree-decomposition-based algorithms alone to be effective. As a result, state-of-the-art model counters leverage tree decomposition information in a more nuanced way, for example, by adapting variable selection heuristics based on a variable's position in the tree decomposition.

In particular, (Korhonen and Järvisalo 2021) demonstrate experimentally that incorporating hybrid scores, which combine structural information from tree decompositions with traditional activity and frequency measures, can improve performance (even though this approach may not always realize the theoretical complexity bounds). They propose the following modification of the VSADS variable selection heuristic to include tree decomposition guidance:

$$\text{score}(x) = \text{act}(x) + \text{freq}(x) - C \min_{\{t \mid x \in T[t]\}} d_T(t)$$

where $\text{act}(x)$ is the activity of variable x, $\text{freq}(x)$ is its frequency, $T[t]$ is the bag at node t in the tree decomposition, $d_T(t)$ is a normalized distance metric (with values between 0 and 1), and C is a positive constant chosen per instance. By default, they use $C = 100 \times \frac{\exp(\frac{n}{w})}{n}$, where n is the number of variables and w is the width of the tree decomposition.

This hybrid approach allows modern model counters to harness the advantages of structural decomposition even in cases where the tree-width is not small, by strategically prioritizing variables based on their structural relevance.

3 `DisCount` Distributed Counting

Our exact model counter, `DisCount` is a distributed algorithm that computes the number of models $\|\Sigma\|$ for a

given input CNF formula Σ. It leverages a fixed, potentially large number of processing units, which may be distributed across a computer network. Unlike traditional parallel model counters that rely on multithreading or shared memory, `DisCount` assumes no shared memory between processing units, ensuring true distribution and scalability.

`DisCount` is founded on the embarrassingly parallel paradigm (Wilkinson and Allen 2005), which entails decomposing a problem into independent parallel tasks that can be executed concurrently. Within the SAT community, this paradigm is epitomized by the cube-and-conquer framework (Heule et al. 2011). Cube-and-conquer was specifically designed to address challenging SAT instances by partitioning the search space into so-called cubes (i.e., partial assignments) using a lookahead solver, and then independently solving each cube using an incremental CDCL solver. The central insight underlying cube-and-conquer is that CDCL solvers are particularly effective at quickly resolving these smaller, well-structured subproblems ("cubes").

This approach confers multiple practical benefits for distributed computation: (i) it requires minimal communication, synchronization, or mutual exclusion between workers, resulting in high efficiency; (ii) it allows for a straightforward implementation since both the master and worker processes rely on standard, unmodified solvers; and (iii) it achieves inherent scalability by distributing independent subproblems across the available computational resources.

Within `DisCount` the computation of $\|\Sigma\|$ for a given CNF formula Σ is coordinated across $n+1$ processing units, where $n \geq 2$: a single master node m and n worker nodes $\mathcal{W} = w_1, \ldots, w_n$. Each worker w_i can dynamically assume multiple roles throughout the computation, including those of SAT solver, model counter, and tree decomposition engine. The specific responsibilities and mode transitions of each worker will be described in detail in the following sections. In general, each worker waits to be assigned a role, performs the corresponding tasks, and then either transitions to a new role or terminates its execution.

Algorithm 1 computes the number of models of a CNF formula Σ in parallel using a cube-and-conquer approach. It takes as input the CNF formula Σ, the number of cubes to be generated ($nbCubes$), and the set of workers $\mathcal{W}$. The output is $count$, the total number of models of Σ.

First, the formula is preprocessed to simplify subsequent computations (line 1), and the simplified formula is broadcast to all workers using the function `broadcast`, which takes as parameters the formula Σ and the set of workers $\mathcal{W}$ (line 2). Next, the formula is partitioned into $nbCubes \times |\mathcal{W}|$ disjoint cubes, enabling independent subproblems that can be processed in parallel (line 3) through a call to `generateCubes`. The details of this function are provided in Section 3.1. Each worker keeps a status flag (*idle*) indicating its availability for new tasks (line 4). All workers are then initialized in counter mode by calling `setWorkerCounterMode` on $\mathcal{W}$ (line 5).

The main loop assigns available cubes to idle workers, marking them as busy when a cube is dispatched (lines 7–10). Specifically, whenever an available worker $w \in \mathcal{W}$ is found, a cube $\tau \in \mathcal{C}$ is selected (line 8) and sent to w using the function `sendCubeToSolve`, which will be detailed in Section 3.2. After the task is assigned, the worker's status flag is updated to indicate it is no longer idle (line 10). When a worker completes its current task, this is detected by the function `free` (lines 10–11), which returns true if the worker has finished its task. The worker then becomes idle again and can be immediately assigned another cube, if any remain.

This process repeats until all cubes have been processed (lines 6–12). Finally, the master process collects the partial model counts from each worker using `getSumCount`, which takes a worker as parameter and returns the total number of models computed by w for its assigned cubes (lines 13–14). These counts are aggregated, and all workers are terminated by calling `stop` for each worker (line 15). The final result, the total number of models of Σ, is returned at line 16.

Algorithm 1: `DisCount`

Data:
- Σ: a CNF formula;
- $nbCubes$: number of worker cubes (integer);
- $\mathcal{W}$: the set of workers.

Result: $count$: an integer representing $\|\Sigma\|$.

```
   // Step 1. Preprocessing.
1  Σ ← preprocessing(Σ);
2  broadcast(Σ, W);
   // Step 2. Cube Generation.
3  C ← generateCubes(Σ, nbCubes × |W|, W);
   // Step 3. Parallel Counting.
4  idle[w] = 1 for each w ∈ W;
5  setWorkerCounterMode(W);
6  while ∃ τ ∈ C do
7      if ∃ w ∈ W s.t. idle[w] = 1 then
8          C ← C \ {τ};
9          sendCubeToSolve(τ);
10         idle[w] = 0;
11     while ∃ w ∈ W s.t. free(w) and idle[w] = 0 do
12         idle[w] = 1;
   // Step 4. Result Collection.
13 count ← 0;
14 for w ∈ W do count ← count + getSumCount(w);
   // Step 5. Terminate all workers.
15 for w ∈ W do stop(w);
16 return count;
```

3.1 Cube Generation

The `generateCubes` procedure, detailed in Algorithm 2, divides the search space of a CNF formula Σ into multiple smaller subproblems called cubes. Given a target number of cubes (*nbCubes*) and a set of workers $\mathcal{W}$, the algorithm strives to generate up to *nbCubes* cubes that are suitable for parallel solving.

Algorithm 2: `generateCubes`

Data:
- Σ: a CNF formula;
- $nbCubes$: number of cubes to generate (integer);
- $\mathcal{W}$: the set of workers.

Result: $\mathcal{Q}$: the set of generated cubes, or $\emptyset$ if $\Sigma \equiv \bot$.

```
// Initialization
1  Q ← {∅};
2  T ← ∅;
3  idle[w] = 1 for each w ∈ W;
4  map[w] = ∅ for each w ∈ W;
5  initBranchingHeuristic(Σ, W);
6  setWorkerSATMode(W);

   // Main cube generation loop
7  while 0 < |Q| < nbCubes or ∃w s.t. idle[w] = 0 do
       // Generate new tasks
8      if Q ≠ ∅ and |Q| + |T| < nbCubes then
           // Select a cube to extend.
9          choose τ ∈ Q and remove it from Q;
10         v ← selectVariable(Σ, τ);
           // Split the cubes.
11         if BCP(Σ ∧ τ ∧ v) ≠ {∅} then
12             T ← T ∪ {BCP(Σ ∧ τ ∧ v)};
13         if BCP(Σ ∧ τ ∧ ¬v) ≠ {∅} then
14             T ← T ∪ {BCP(Σ ∧ τ ∧ ¬v)};
       // Assign a new task
15     if ∃w ∈ W s.t. idle[w] = 1 and T ≠ ∅ then
16         choose τ ∈ T and remove it from T;
17         map[w] ← τ;
18         idle[w] ← 0;
19         sendCube(w, τ);
       // Process a completed task
20     if ∃w ∈ W s.t. done(w) then
21         if getResult(w) is SAT then
22             Q ← Q ∪ {map[w]};
23         idle[w] ← 1;
24 return Q;
```

At the outset (lines 1–6), the initialization phase prepares all necessary data structures. Specifically, the set of finalized cubes $\mathcal{Q}$ is initialized with the empty assignment, representing the entire search space (line 1), while the set of pending tasks $\mathcal{T}$ is initialized as empty (line 2). For every worker $w \in \mathcal{W}$, the algorithm marks the worker as idle (line 3) and clears any previous task assignments (line 4). Additionally, it initializes the branching heuristic based on Σ and the set of workers (line 5); this routine will be detailed later. Finally, the SAT solvers assigned to each worker are configured for search mode, preparing them to process cubes as they become available (line 6).

Following initialization, the algorithm enters its main loop, repeatedly generating and distributing cubes until the prescribed number is reached or further splitting is infeasible. If the original formula is unsatisfiable, the algorithm returns the empty set. The core of the method iteratively maintains four entities: the set $\mathcal{Q}$ of finalized cubes, the set $\mathcal{T}$ of cubes pending distribution to workers, the array of idle workers, and a mapping that associates each worker with the cube they are currently processing. This main loop can be conceptually divided into three phases:

1. Generating New Candidate Cubes (lines 8–15): During each iteration, the algorithm selects a cube τ from $\mathcal{Q}$ (line 9) and chooses a branching variable using the configured heuristic (line 10). It then splits τ into two child cubes by assigning the variable both true and false (lines 11–14). Each new subcube undergoes BCP. Only satisfiable subcubes (modulo BCP) are kept and added to the task set $\mathcal{T}$ (lines 12 and 14).

2. Assigning Tasks to Workers (lines 15–19): When there are idle workers and pending tasks, a cube τ from $\mathcal{T}$ is selected and assigned to an available worker (lines 16–18). The mapping map records that worker w is handling task τ (line 17), and w is marked as busy (line 18). The cube is then sent to the worker via the subroutine `sendCube`, which will be detailed later.

3. Processing Completed Tasks (lines 20–23): As workers complete their assigned cubes, the algorithm checks whether the cube is satisfiable (the function `done` return `true`). If so, the cube is added to $\mathcal{Q}$ (lines 21–22), and the corresponding worker is marked idle again (line 23).

This structured approach enables efficient partitioning of the search space and balanced workload distribution among workers. The use of the `sendCube` function facilitates parallel SAT solving on the generated cubes, thereby accelerating the cube generation process. The function `sendCube` assigns a selected cube to a specific worker w for satisfiability checking. Since all workers have local access to the underlying CNF formula Σ (see Algorithm 1, line 2), it is unnecessary to explicitly transmit Σ with each cube assignment. Upon receiving a new cube, worker w enters solving mode, causing `done`(w) to return `false`. The worker then invokes a SAT solver on the assigned cube. Once the solver finishes, the worker stores the satisfiability result, which can be retrieved using the `getResult` function, and updates its status so that `done`(w) subsequently returns `true`.

To conclude this section on cube generation, we discuss the branching heuristic, a critical component for determining the quality of generated cubes and the overall efficiency of `DisCount`. Selecting an effective branching heuristic is challenging, as little information is available during partitioning regarding how efficiently a model counter will solve the resulting subproblems. The heuristic must thus estimate subproblem difficulty, a task that is inherently nontrivial. Our approach takes two factors into account: which cube to extend and which variable to branch on. For cube selection, we prioritize extending the smallest cube, based on the intuition that cubes with more literals fixed are likely to be easier for the solver.

Regarding variable selection, we evaluated several classical SAT branching heuristics (such as MOM (Dubois

et al. 1993), JWTS (Jeroslow and Wang 1990), and VSIDS (Moskewicz et al. 2001)) as well as heuristics developed specifically for model counting (like DLCS and VSADS (Sang, Beame, and Kautz 2005a)). Since VSIDS and VSADS rely on conflict-driven variable scoring, we invoke a CDCL SAT solver during the initialization phase (line 5) to collect conflict data that guides subsequent branching decisions.

The best-performing heuristic for model counting often combines classical branching strategies with structural information extracted from a tree decomposition of the formula's primal graph. To this end, we also implemented the heuristic from (Korhonen and Järvisalo 2021), presented in the previous section. We compute the tree decomposition in a distributed fashion by allocating this task across multiple workers, each running the `FlowCutter` (Hamann and Strasser 2018) anytime algorithm with a different random seed. As a result, when the allocated budget expires, we choose the best decomposition obtained (i.e., the one with minimal treewidth) among all workers.

After selecting the tree decomposition, we identify its centroid node to serve as the root. This is achieved by iteratively removing leaf nodes until one or two central nodes remain, ensuring a balanced starting point for variable selection. A vector of variable scores is then computed based on the decomposition and integrated with classical heuristics to guide branching decisions.

3.2 Model Counting Workers

Once the cubes have been generated and distributed, each worker process is tasked with counting the number of models for its assigned cube. In our framework, we do not impose any restrictions on the specific model counter that can be used; indeed, our system is designed with an abstraction layer for the counter, enabling the straightforward integration of new or third-party model counters. The only requirement is compliance with a common interface for communication with the master process.

However, for practical efficiency, counters are invoked in *assumptions mode*, similar to the approach used in incremental SAT solving (Nadel and Ryvchin 2012). In this setting, *assumptions* are literals that are asserted for the duration of a single solver invocation. This allows the underlying SAT to retain learnt clauses and internal state across different calls, since the base formula remains unchanged and only the set of assumed literals differs. As a result, both clause learning and variable activity data can be preserved, improving performance on related subproblems, as is common practice in incremental SAT solving.

This principle extends naturally to model counting: recent works such as (Lagniez and Marquis 2019) demonstrate that maintaining a persistent cache across multiple counting queries can be highly beneficial. In this way, learnt clauses, caches, and variable weights are retained by the counter, mirroring the advantages of incremental SAT solving and allowing efficient reuse of information across different cubes.

Another important design decision is that model count results are not communicated to the master after every counting invocation. Instead, each worker maintains an accumulated sum of its model counts locally. The current result can be retrieved only when necessary, by invoking the function `getSumCount` with the worker identifier as a parameter (see Algorithm 1, line 14). This approach minimizes communication overhead between the master and the workers and also helps prevent a loss in numerical precision that could result from frequent serialization and deserialization of intermediate results.

4 Related Works

The *cube-and-conquer* paradigm (Heule et al. 2011) is a pivotal approach for tackling challenging SAT instances. In the constraint programming (CP) community, a closely related principle is known as *Embarrassingly Parallel Search* (EPS) (Malapert, Régin, and Rezgui 2016), which extends the concept to CP problems. More broadly, partitioning complex problems into independent subproblems suitable for parallel or distributed solving is a powerful idea that goes well beyond SAT and CP. Since its introduction, cube-and-conquer has inspired numerous variants and adaptations for a variety of domains, with research focusing on improved partitioning heuristics, effective load balancing, and the flexible integration of diverse solver types (Hamadi and Sais 2018).

For model counting, this principle is exemplified by the parallel counter `dCountAntom` (Burchard, Schubert, and Becker 2016), an extension of `CountAntom` that introduces message passing between a single master counter and multiple slave counters. The master delegates subproblems (nodes in the search tree) to the slaves, provided they are at a decision level exceeding a configurable threshold δ. However, only the master is permitted to redistribute work: slave solvers cannot share their workload among themselves, even if they encounter particularly hard subproblems. If a slave w_i is unable to solve a node within a fixed time limit τ, it returns the job to the master; the master may then further partition and redistribute the problem, but only after skipping a predefined number of decision levels, governed by another parameter, σ.

In our view, although `dCountAntom` follows a cube-and-conquer philosophy similar to ours, its distribution strategy introduces several limitations. Most notably, when a slave relinquishes an unfinished job, the computational effort expended on that task is lost, since counters do not exploit solving under assumptions. Additionally, when the master is assigned a particularly challenging subproblem, slave processes often remain idle, resulting in poor resource utilization and workload imbalance; especially if individual slave cores are insufficient to resolve difficult cases within the allowed time, forcing the master to do most of the work.

`dCountAntom` also aggressively shares learned conflict clauses: when a clause is learned by the master or received from a slave, it is broadcast to all slaves. Furthermore, entire sub-formulae are transmitted from the master to the slaves, leading to significant communication overhead, particularly for large subproblems. In contrast, our approach mitigates such overhead by generating a sufficient number of cubes at the outset and refraining from sharing information between workers. Finally, the overall performance of

dCountAntom appears sensitive to the choice of the key parameters σ and τ, which must be carefully tuned to balance granularity and timeouts for effective parallelism.

Recent research has enhanced these techniques by incorporating structural information into the partitioning phase (Fichte et al. 2023; Bannach and Hecher 2024a; Dreier, Ordyniak, and Szeider 2024). More precisely, Bannach and Hecher (2024b) proposed leveraging tree decompositions to guide cube generation for MaxSAT solving. By exploiting variable dependencies revealed by the tree decomposition, their method produces more balanced and tractable cubes for sequential processing, resulting in notable improvements in MaxSAT solver performance. Although their approach is sequential, it demonstrates the significant benefits of utilizing problem structure during partitioning. To the best of our knowledge, this strategy has not been applied in the context of model counting.

dmc offers another significant alternative to the cube-and-conquer paradigm (Lagniez, Marquis, and Szczepanski 2018). In dmc, each worker executes an instance of d4 (Lagniez and Marquis 2017a), and workload distribution is managed by a dynamic job-sharing mechanism inspired by work-stealing. Unlike dCountAntom, dmc eschews the direct exchange of formula fragments and learnt clauses between workers, thereby reducing communication overhead and enhancing scalability.

A key characteristic of dmc is that it requires specific modifications to the underlying model counter to facilitate communication protocols, arithmetic job representations, and synchronized task delegation. This is in contrast to our approach, which is intentionally non-intrusive: we treat the underlying model counter as a black box, without altering its internal algorithms or data structures. This design greatly simplifies integration and ensures broad compatibility with different model counting tools. Moreover, dmc aggregates model counts using double-precision floating-point arithmetic. While this may suffice for many applications, it can lead to precision loss when formulas have extremely large model counts. By contrast, our approach employs arbitrary-precision arithmetic, guaranteeing exact results even in cases involving very large or combinatorially complex counts.

We did not consider model counters that leverage GPU acceleration, such as the approach proposed in (Fichte, Hecher, and Roland 2021). While integrating GPU-based model counting into our framework could offer interesting performance benefits and is worth exploring in future work, it falls outside the scope of this paper.

5 Experiments

Our approach is implemented in C++. The software used in the experiments, along with the corresponding logs, is available at https://zenodo.org/records/16536062. For the SAT solving component, we employ the CaDiCaL CDCL SAT solver (Biere et al. 2024), using a single incremental solver instance throughout to optimize performance. For tree decomposition, we utilize FlowCutter (Hamann and Strasser 2018), relying on the implementation submitted to the PACE 2017 tree decomposition challenge and available on GitHub[2]. A time budget of 10 seconds is allocated for each tree decomposition computation.

Our preprocessing step uses B+E (Lagniez, Lonca, and Marquis 2016; Lagniez, Lonca, and Marquis 2020), with the source code available via the Compile! Project website[3]. For the experiments, we use the equiv option, which is based on a preprocessing combination of vivification, backbone extraction, and occurrence elimination (Lagniez and Marquis 2014; Lagniez and Marquis 2017b). The rationale for preferring this approach over more aggressive techniques based on definability is practical: computing the set of defined variables and eliminating them can be time-consuming; and, because this preprocessing is not parallelized, it may lead to a significant waste of computational resources by keeping workers idle for extended periods. A time budget of 5 seconds is allocated for the preprocessing step.

Regarding the model counters, we consider both d4 (Lagniez and Marquis 2017a)[4] and SharpSAT-TD (Korhonen and Järvisalo 2021)[5]. We have upgraded both for incremental operation with assumptions.

We evaluated our approach against the state-of-the-art distributed model counters dCountAntom (Burchard, Schubert, and Becker 2016) and dmc (Lagniez, Marquis, and Szczepanski 2018). The benchmarks were taken from the most recent model counting competition[6]. All experiments were conducted on a cluster comprising thirty-two Intel® Xeon® E5-2643 v4 CPUs running at 3.30 GHz, with Rocky Linux 9.5 (Linux kernel 5.14) as the operating system. The cluster nodes are interconnected via an Ethernet controller with a bandwidth of 1 GiB/s. The software environment included GCC 11.5 and Open MPI 5.1.0a1. A wall-clock time limit of 900 seconds and a memory limit of 32 GiB were imposed on each run.

Figure 2 presents a cactus plot illustrating, for each wall-clock time limit (on the x-axis), the number of instances solved (on the y-axis) by each model counter included in our experiments: dCountAntom, dmc, and DisCount (using d4 as the underlying model counter). To assess the scalability of DisCount, we evaluated its performance using 2, 4, 8, 16, 32, 64, and 128 cores. Regardless of the number of cores allocated, DisCount was configured to generate 30 cubes per worker. For comparison, both dCountAntom and dmc were executed with 128 cores.

The results show that increasing the number of cores significantly reduces the time required by DisCount to solve instances, leading to a higher number of instances solved within the time limit. Specifically, DisCount solves 67 instances using 2 cores, 83 with 4 cores, 93 with 8 cores, 94 with 16 cores, 103 with 32 cores, 109 with 64 cores, and 111 with 128 cores. Notably, the sequential version solves 81 instances, showing that the benefits of the embarrassingly parallel search emerge only when run in parallel.

[2]https://github.com/kit-algo/flow-cutter-pace17.git
[3]http://www.cril.univ-artois.fr/kc/bpe2.html
[4]https://github.com/crillab/d4v2
[5]https://github.com/Laakeri/sharpsat-td
[6]https://zenodo.org/records/14249068

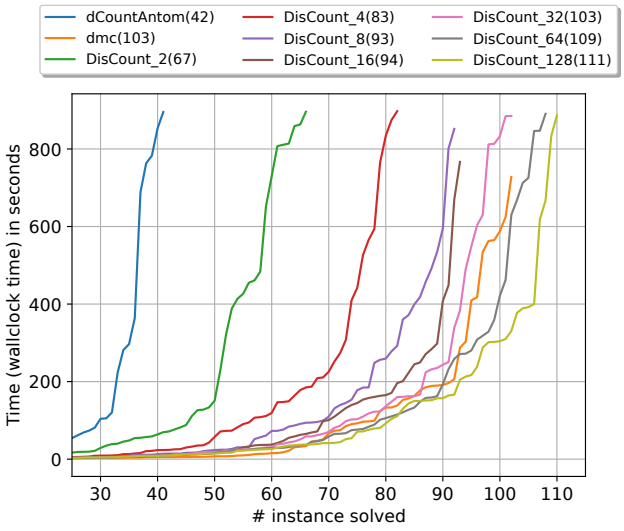


Figure 2: Performance comparison of the model counters `dCountAntom`, `dmc`, and various versions of `DisCount` for different numbers of cores. The plot reports the number of instances solved as a function of time. For each solver, the total number of instances solved is indicated in parentheses in the legend.

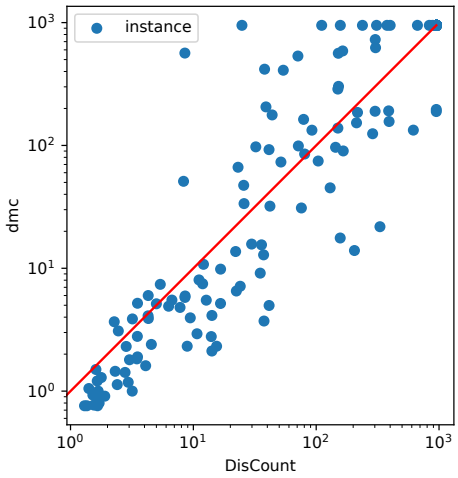


Figure 3: Scatter plot comparing the solving times of `dmc` (y-axis) and `DisCount` (x-axis) for each benchmark instance. Each point corresponds to a single instance; the x-coordinate shows the time (in seconds) required by `DisCount`, while the y-coordinate indicates the time taken by `dmc`.

In contrast, `dCountAntom` running on 128 cores solves only 42 instances, substantially fewer than `DisCount`. Moreover, `DisCount` is able to solve all the instances solved by `dCountAntom`. As noted in (Lagniez, Marquis, and Szczepanski 2018), the disappointing performance of `dCountAntom` correlates with the choice of parameters δ and τ, whose default values may not be well suited to our benchmark set. We also tried alternative parameter values but saw no performance gains.

`dmc` solved 103 instances, which is consistent with previous observations (Lagniez, Marquis, and Szczepanski 2018) that `dmc` outperforms `dCountAntom`. Compared to `dmc`, `DisCount` was able to solve eight additional instances. This result is particularly notable because integrating `d4` with `DisCount` required minimal implementation effort, in contrast to the significant reengineering needed to adapt `dmc` for use with other state-of-the-art model counters.

`DisCount` solves all but two of the instances solved by `dmc`. Upon further inspection, we found that the tree decompositions used by `DisCount` for these two benchmarks were highly unbalanced. In both cases, the first bag of the decomposition was very small, meaning that after only a few variable assignments, the remaining formula decomposed into connected components, an opportunity our current implementation does not exploit. As a result, all workers ended up solving essentially the same hard subproblem, leading to significant inefficiency for these particular benchmarks.

Figure 3 presents a scatter plot comparing the solving times of `dmc` and `DisCount` for each benchmark instance. Each data point corresponds to a single instance, with the x-axis indicating the time (in seconds) required by `DisCount` and the y-axis representing the time required by `dmc`. The experimental results show that for instances solved in under 50 seconds, `dmc` is noticeably more efficient. However, as the running time increases, `DisCount` becomes increasingly competitive, eventually outperforming `dmc` on instances that require more than 700 seconds to solve.

The initial inefficiency of `DisCount` can be primarily attributed to the overhead associated with cube generation, which requires the construction of $128 \times 30 = 3840$ cubes. Specifically, for 25 instances, the time spent in cube generation exceeded 20 seconds. Even though this phase is parallelized, it may fail to complete efficiently on particularly challenging benchmarks, especially when a large proportion of generated cubes are unsatisfiable, leading to a significant number of SAT solver invocations during this phase. These observations suggest that there is room for improvement by reducing the number of cubes allocated to each worker which could decrease cube generation time and improve overall performance on some instances.

Regarding the performance of `DisCount`, the version utilizing `SharpSAT-TD` and running on 128 cores was able to solve 104 instances. On the other hand, the version employing `d4` as the underlying counter managed to solve 7 instances that the `SharpSAT-TD`-based version could not. Conversely, the `SharpSAT-TD`-based version solved 3 instances that were not solved by the `d4`-based version. These experiments highlight the significance of the chosen counter with respect to the specific instance under consideration.

Figure 4 presents the distribution of speedup achieved by `DisCount` as the number of cores increases, with each boxplot corresponding to a different core count. The y-axis represents speedup relative to the sequential baseline (`DisCount` running on two cores), and the x-axis indicates the number of cores used. For any instance Σ solved in t_n seconds by `DisCount` running on n cores, the reported speedup is defined as $\frac{\min(t_1,3600)}{t_n}$, where t_1 is the solution

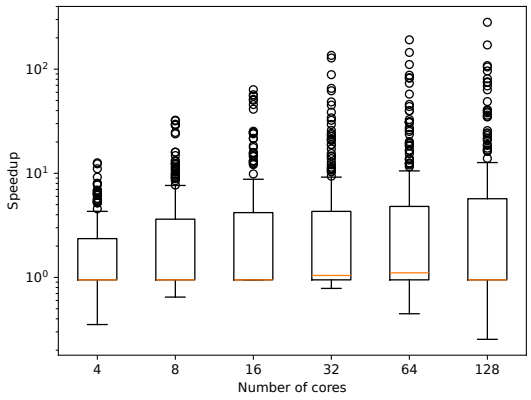


Figure 4: Scatter plot comparing the solving times of `dmc` (y-axis) and `DisCount` (x-axis) for each benchmark instance. Each point corresponds to a single instance; the x-coordinate shows the time (in seconds) required by `DisCount`, while the y-coordinate indicates the time taken by `dmc`.

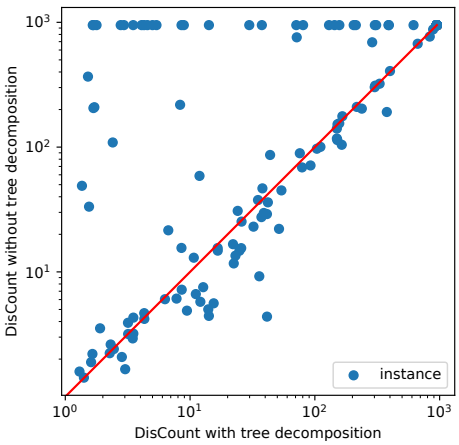


Figure 5: Scatter plot comparing the solving times of `DisCount` (with tree decomposition-based cube generation, x-axis) and `DisCount-noDecomp` (without tree decomposition, y-axis) for each benchmark instance. Each point represents a single instance; the x- and y-coordinates correspond to the solving times (in seconds) for `DisCount` and `DisCount-noDecomp`, respectively.

time using two cores. If Σ is not solved within 900 seconds on a single worker, t_1 is undefined. Therefore, the observed speedups are conservative lower bounds on the actual speedups that could be achieved if running `DisCount` with one worker to completion were feasible.

As the number of cores increases from 4 to 128, the median speedup generally rises, demonstrating good scalability of our approach. Additionally, the spread of speedup values becomes wider, with an increasing number of outliers corresponding to exceptional speedups on particular instances; especially pronounced beyond 32 cores. The interquartile range remains broad across all configurations, indicating substantial variability in parallel speedup depending on the instance. Overall, the results show that increasing core count is beneficial both for median and peak speedup, although the gains vary significantly across instances. This highlights both the effectiveness of our parallel cube-and-conquer strategy as well as the importance of instance characteristics, since factors such as load imbalance or inherent sequential bottlenecks in cube generation or solving can limit parallel efficiency in certain cases.

To assess the impact of using tree decomposition during cube generation, we evaluated a variant of `DisCount` that omits this heuristic, referred to as `DisCount-noDecomp`. Figure 5 presents a scatter plot comparing the running times of `DisCount-noDecomp` and the baseline `DisCount` employing tree decomposition. As illustrated in the figure, incorporating the tree decomposition heuristic for cube generation significantly improves the performance of `DisCount`. While `DisCount-noDecomp` was able to solve only 83 instances, the full version of `DisCount` with tree decomposition solved 111 instances, highlighting the effectiveness of the heuristic.

To conclude our experimental evaluation, we analyzed the effect of varying the parameter $nbCubes$, which controls the number of cubes generated by the workers. Due to resource constraints, these experiments were conducted using `DisCount` on 80 cores. We considered the following values for $nbCubes$: 5, 10, 30, 100, and 500. The corresponding numbers of instances solved were 106, 106, 107, 104, and 96, respectively. As can be observed, increasing the number of generated cubes leads to a decrease in overall effectiveness. This result is consistent with our previous observations regarding the computational cost of the cube generation phase.

6 Conclusion and Perspectives

In this paper, we have introduced a general and modular framework for distributed and parallel model counting, grounded in the cube-and-conquer paradigm. Our method achieves efficient search-space partitioning through parallel cube generation, and facilitates the seamless integration of arbitrary model counters via an abstraction layer, eliminating the need to modify underlying counting engines. By employing advanced structural heuristics, such as tree decomposition-based branching strategies, our approach attains both balanced partitioning and enhanced model counting performance. In contrast to previous distributed methods like `dCountAntom` and `dmc`, our framework minimizes communication overhead, supports arbitrary-precision arithmetic to avoid numerical precision loss, and remains non-intrusive with respect to existing model counters. Experimental results on challenging benchmark instances from the 2024 model counting competition demonstrate the scalability and flexibility of our framework.

Directions for future work include dynamic load balancing, detection of connected components during the cube generation procedure, integration with approximate counting techniques, and the development of advanced preprocessing strategies adapted for distributed environments. We believe that the flexibility and efficiency of our approach make it a promising foundation for continued research and practical applications in distributed model counting.

Acknowledgements

This work has benefited from the support of the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) of the French National Research Agency (ANR), NSFC under Grant No.61976050, and Jilin province science and technology department project under Grant 20240602005RC.

References

Baluta, T.; Shen, S.; Shinde, S.; Meel, K. S.; and Saxena, P. 2019. Quantitative verification of neural networks and its security applications. In Cavallaro, L.; Kinder, J.; Wang, X.; and Katz, J., eds., *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 1249–1264. ACM.

Bannach, M., and Hecher, M. 2024a. On weighted maximum model counting: Complexity and fragments. In *36th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2024, Herndon, VA, USA, October 28-30, 2024*, 1–9. IEEE.

Bannach, M., and Hecher, M. 2024b. Structure-guided cube-and-conquer for maxsat. In Benz, N.; Gopinath, D.; and Shi, N., eds., *NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings*, volume 14627 of *Lecture Notes in Computer Science*, 3–20. Springer.

Biere, A.; Faller, T.; Fazekas, K.; Fleury, M.; Froleyks, N.; and Pollitt, F. 2024. Cadical 2.0. In Gurfinkel, A., and Ganesh, V., eds., *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, 133–152. Springer.

Bodlaender, H. L. 2005. Discovering treewidth. In Vojtás, P.; Bieliková, M.; Charron-Bost, B.; and Sýkora, O., eds., *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, volume 3381 of *Lecture Notes in Computer Science*, 1–16. Springer.

Burchard, J.; Schubert, T.; and Becker, B. 2015. Laissez-faire caching for parallel #sat solving. In Heule, M., and Weaver, S. A., eds., *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, 46–61. Springer.

Burchard, J.; Schubert, T.; and Becker, B. 2016. Distributed parallel #sat solving. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, 326–335. IEEE Computer Society.

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.* 172(6-7):772–799.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In de Mántaras, R. L., and Saitta, L., eds., *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 328–332. IOS Press.

Domshlak, C., and Hoffmann, J. 2006. Fast probabilistic planning through weighted model counting. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, 243–252. AAAI.

Dreier, J.; Ordyniak, S.; and Szeider, S. 2024. SAT backdoors: Depth beats size. *J. Comput. Syst. Sci.* 142:103520.

Dubois, O.; André, P.; Boufkhad, Y.; and Carlier, J. 1993. SAT versus UNSAT. In Johnson, D. S., and Trick, M. A., eds., *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 415–436. DIMACS/AMS.

Dubray, A.; Schaus, P.; and Nijssen, S. 2023. Probabilistic inference by projected weighted model counting on horn clauses. In Yap, R. H. C., ed., *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, 15:1–15:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Dueñas-Osorio, L.; Meel, K. S.; Paredes, R.; and Vardi, M. Y. 2017. Counting-based reliability estimation for power-transmission grids. In Singh, S., and Markovitch, S., eds., *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, 4488–4494. AAAI Press.

Feiten, L.; Sauer, M.; Schubert, T.; Czutro, A.; Böhl, E.; Polian, I.; and Becker, B. 2012. #sat-based vulnerability analysis of security components - A case study. In *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT 2012, Austin, TX, USA, October 3-5, 2012*, 49–54. IEEE Computer Society.

Fichte, J. K.; Hecher, M.; Morak, M.; Thier, P.; and Woltran, S. 2023. Solving projected model counting by utilizing treewidth and its limits. *Artif. Intell.* 314:103810.

Fichte, J. K.; Hecher, M.; and Roland, V. 2021. Parallel model counting with CUDA: algorithm engineering for efficient hardware utilization. In Michel, L. D., ed., *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Girol, G.; Farinier, B.; and Bardin, S. 2021. Not all bugs are created equal, but robust reachability can tell the difference. In Silva, A., and Leino, K. R. M., eds., *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, 669–693. Springer.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2021. Model counting. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 993–1014.

Hamadi, Y., and Sais, L., eds. 2018. *Handbook of Parallel Constraint Reasoning*. Springer.

Hamann, M., and Strasser, B. 2018. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics* 23.

Heinz, S., and Sachenbacher, M. 2009. Using model counting to find optimal distinguishing tests. In van Hoeve, W. J., and Hooker, J. N., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*, volume 5547 of *Lecture Notes in Computer Science*, 117–131. Springer.

Heule, M.; Kullmann, O.; Wieringa, S.; and Biere, A. 2011. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Eder, K.; Lourenço, J.; and Shehory, O., eds., *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, volume 7261 of *Lecture Notes in Computer Science*, 50–65. Springer.

Izza, Y.; Huang, X.; Ignatiev, A.; Narodytska, N.; Cooper, M. C.; and Marques-Silva, J. 2023. On computing probabilistic abductive explanations. *Int. J. Approx. Reason.* 159:108939.

Jeroslow, R. G., and Wang, J. 1990. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.* 1:167–187.

Korhonen, T., and Järvisalo, M. 2021. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In Michel, L. D., ed., *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, 8:1–8:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Koriche, F.; Lagniez, J.; Marquis, P.; and Thomas, S. 2013. Knowledge compilation for model counting: Affine decision trees. In Rossi, F., ed., *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 947–953. IJCAI/AAAI.

Lagniez, J., and Marquis, P. 2014. Preprocessing for propositional model counting. In Brodley, C. E., and Stone, P., eds., *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, 2688–2694. AAAI Press.

Lagniez, J., and Marquis, P. 2017a. An improved decision-dnnf compiler. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 667–673. ijcai.org.

Lagniez, J., and Marquis, P. 2017b. On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.* 58(4):413–481.

Lagniez, J., and Marquis, P. 2019. A recursive algorithm for projected model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, 1536–1543. AAAI Press.

Lagniez, J.; Lonca, E.; and Marquis, P. 2016. Improving model counting by leveraging definability. In Kambhampati, S., ed., *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 751–757. IJCAI/AAAI Press.

Lagniez, J.; Lonca, E.; and Marquis, P. 2020. Definability for model counting. *Artif. Intell.* 281:103229.

Lagniez, J.; Marquis, P.; and Szczepanski, N. 2018. DMC: A distributed model counter. In Lang, J., ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, 1331–1338. ijcai.org.

Malapert, A.; Régin, J.; and Rezgui, M. 2016. Embarrassingly parallel search in constraint programming. *J. Artif. Intell. Res.* 57:421–464.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-driven clause learning SAT solvers. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 133–182.

Mei, J.; Bonsangue, M. M.; and Laarman, A. 2024. Simulating quantum circuits by model counting. In Gurfinkel, A., and Ganesh, V., eds., *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, volume 14683 of *Lecture Notes in Computer Science*, 555–578. Springer.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, 530–535. ACM.

Muise, C. J.; McIlraith, S. A.; Beck, J. C.; and Hsu, E. I. 2012. Dsharp: Fast d-dnnf compilation with sharpsat. In Kosseim, L., and Inkpen, D., eds., *Advances in Artificial Intelligence - 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012. Proceedings*, volume 7310 of *Lecture Notes in Computer Science*, 356–361. Springer.

Nadel, A., and Ryvchin, V. 2012. Efficient SAT solving under assumptions. In Cimatti, A., and Sebastiani, R., eds., *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, 242–255. Springer.

Oztok, U., and Darwiche, A. 2015. A top-down compiler for sentential decision diagrams. In Yang, Q., and

Wooldridge, M. J., eds., *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 3141–3148. AAAI Press.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-dnnf representations. In Biundo, S.; Myers, K. L.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, 141–150. AAAI.

Robertson, N., and Seymour, P. D. 1986. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms* 7(3):309–322.

Sang, T.; Beame, P.; and Kautz, H. A. 2005a. Heuristics for fast exact model counting. In Bacchus, F., and Walsh, T., eds., *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, 226–240. Springer.

Sang, T.; Beame, P.; and Kautz, H. A. 2005b. Performing bayesian inference by weighted model counting. In Veloso, M. M., and Kambhampati, S., eds., *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 475–482. AAAI Press / The MIT Press.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. GANAK: A scalable probabilistic exact model counter. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1169–1176. ijcai.org.

Teuber, S., and Weigl, A. 2021. Quantifying software reliability via model-counting. In Abate, A., and Marin, A., eds., *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, 59–79. Springer.

Thurley, M. 2006. sharpsat - counting models with advanced component caching and implicit BCP. In Biere, A., and Gomes, C. P., eds., *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, 424–429. Springer.

Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.* 20(5):865–877.

Wilkinson, B., and Allen, M. 2005. *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education.