

gDMC: A Generic Distributed Model Counting Framework via Work-Stealing

Zhenghang Xu^{1,2}, Minghao Yin^{1,2}, Junping Zhou^{1,2}, Jean-Marie Lagniez³

¹School of Information Science and Technology, Northeast Normal University, Changchun, China

²Key Laboratory of Applied Statistics of MOE, Northeast Normal University, Changchun, China

³Univ. Artois, CNRS, CRIL, F-62300 Lens, France

{xuzh121, ymh, zhoujp877}@nenu.edu.cn, lagniez@cril.fr

Abstract

Propositional Model Counting ($\#SAT$) is essential for probabilistic reasoning but faces scalability limits on single cores. Existing distributed approaches struggle with high initialization overheads (static decomposition) or rigid architecture. We propose a novel, generic framework for distributed *exact* model counting. Leveraging C++ templates, our architecture decouples parallel orchestration from solving logic, enabling state-of-the-art solvers to be parallelized with minimal modification. We implement an adaptive work-stealing strategy that ensures effective load balancing. Experiments on competition benchmarks show that our approach achieves near-linear scalability and significantly outperforms existing distributed solvers.

1 Introduction

Model counting (MC), also known as $\#SAT$, is the problem of computing the number of models (satisfying assignments) of a given propositional formula, typically in CNF. Its direct extension, *weighted model counting* (WMC), is of tremendous importance in a wide range of AI applications, including probabilistic inference [Sang *et al.*, 2005; Chavira and Darwiche, 2008], planning under uncertainty [Domshlak and Hoffmann, 2006], and neural network verification [Baluta *et al.*, 2019]. Beyond AI, model counting is critical in domains such as hardware testing and reliability estimation [Feiten *et al.*, 2012; Teuber and Weigl, 2021; Dueñas-Osorio *et al.*, 2017; Girol *et al.*, 2021; Mei *et al.*, 2024].

The computational hardness of model counting is underscored by Toda’s theorem [Toda, 1991], which establishes that $P^{\#P}$ encompasses the entire polynomial hierarchy (PH). Despite this theoretical hardness, significant progress has been made in the design of efficient sequential model counters. Modern search-based solvers, such as D4 [Lagniez and Marquis, 2017], GANAK [Sharma *et al.*, 2019], and SharpSAT-TD [Korhonen and Järvisalo, 2021], leverage Conflict-Driven Clause Learning (CDCL), component caching, and implicit Boolean Constraint Propagation (BCP) to prune the search space effectively. However, sequential performance is increasingly constrained by the phys-

ical limits of single-core clock speeds and the memory required to maintain massive component caches.

The importance of $\#SAT$ has driven significant research into scalable algorithms, evidenced by recent model counting competitions [Fichte and Hecher, 2025]. To handle the growing complexity of real-world instances, parallel and distributed computing offers a promising path forward. However, distributing $\#SAT$ is fundamentally harder than distributing SAT. In SAT, the search can stop as soon as *one* solution is found, allowing for aggressive, speculative strategies. In $\#SAT$, the entire solution space must be covered, meaning the total runtime is determined by the slowest worker (the ‘straggler’ problem). While GPU-accelerated approaches exist [Fichte *et al.*, 2021], they represent a distinct architectural paradigm; this paper focuses on distributed CPU-based solving.

State-of-the-art parallel and distributed solvers, such as dCountAntom [Burchard *et al.*, 2016] and the more recent DisCount [Xu *et al.*, 2025], typically rely on a *Cube-and-Conquer* strategy. These methods decompose the search space into independent subproblems (cubes) which are then distributed across nodes. However, this approach has several limitations. First, the initialization overhead is significant; as noted in [Xu *et al.*, 2025], the decomposition phase can be costly enough to negate performance gains on medium-hard instances. Second, static decomposition suffers from *redundancy*: separate workers may unknowingly explore identical search spaces. For instance, when different cubes lead to the same connected component, resulting in duplicated effort. Finally, achieving effective load balancing requires generating a massive number of cubes, a parameter that is notoriously difficult to tune and highly instance-dependent.

The distributed solver DMC [Lagniez *et al.*, 2018] offers a compelling alternative to static partitioning by employing a dynamic work-stealing mechanism. Unlike dCountAntom, DMC avoids the exchange of formula fragments and learned clauses, significantly minimizing communication overhead. However, its implementation is tightly coupled to the D4 solver, requiring invasive modifications to the sequential core that hinder modularity and maintenance. Additionally, DMC uses double-precision floating-point arithmetic for aggregation; this induces precision loss on instances with large model counts, rendering it unsuitable for exact counting tasks.

To overcome these limitations, we propose a novel C++

framework that leverages *generic programming* (via Concepts and Templates) to enable efficient, solver-agnostic distributed model counting within the class of DPLL-style counters. Our architecture employs a modular stack that separates distributed orchestration from core solving logic, enabling state-of-the-art solvers to be parallelized with minimal modification. By resolving communication types at compile time, this design ensures zero-cost abstractions with negligible run-time overhead. Furthermore, we formally establish the conditions necessary for safe and complete work sharing.

Specifically, our contributions are threefold. First, we provide a generic, template-based architecture that simplifies the integration of work-stealing into existing DPLL-style model counters. Second, we identify the formal requirements for valid work transfer, establishing the theoretical guarantees necessary for the soundness and completeness of the distributed search. Finally, we demonstrate through extensive experiments that our solution outperforms both static partitioning (e.g., *DisCount*) and existing work-stealing solvers (e.g., *DMC*), achieving near-linear speedups and solving significantly more instances.

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 $\top$ (true) and $\perp$ (false). A *literal* ℓ is either a variable $x \in \mathcal{P}$ or its negation $\bar{x}$. A *cube* (or term) is a conjunction of literals, and a *clause* is a disjunction of literals. When convenient, clauses and cubes are regarded as sets of literals. A *CNF formula* is a conjunction of clauses, also viewed as a set of 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 under ω (denoted $\omega \models \Sigma$). We use $\models$ to denote logical entailment and $\equiv$ for logical equivalence. 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 to compute $\|\Sigma\|$ for a given formula and is the canonical example of a #P-complete problem.

Example 1. As a running example, let us consider the CNF formula Σ defined over $Var(\Sigma) = \{x_1, \dots, x_7\}$:

$$\begin{array}{llll} \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6 & \bar{x}_1 \vee x_4 \vee x_5 & x_1 \vee x_4 \vee x_5 & x_1 \vee x_3 \\ \bar{x}_1 \vee \bar{x}_2 \vee x_6 & \bar{x}_1 \vee x_2 \vee x_3 & \bar{x}_1 \vee x_2 \vee x_3 \vee \bar{x}_7 & \\ \bar{x}_1 \vee \bar{x}_3 \vee x_6 & \bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_6 & \bar{x}_1 \vee x_3 \vee x_4 \vee x_7 & \end{array}$$

In this case, $\|\Sigma\| = 24$.

The Boolean Constraint Propagator, $BCP(\Sigma)$, performs *unit propagation* by iteratively enforcing unit clauses within the formula Σ . $BCP(\Sigma)$ returns $\perp$ (conflict) if unit propagation derives the empty clause (a unit refutation). Otherwise, it returns the set of literals S derived from Σ . The *conditioning* of Σ by a literal ℓ , denoted $\Sigma|_{\ell}$, is obtained by replacing every occurrence of ℓ with $\top$ and every occurrence of its negation $\bar{\ell}$ with $\perp$. This is followed by standard Boolean simplifications to remove satisfied clauses and removing $\perp$ literals from the remaining clauses until the formula is minimal. This notion

extends to a set of literals $S = \{\ell_1, \dots, \ell_m\}$ by iterative application: $\Sigma|_S = (\dots(\Sigma|_{\ell_1})\dots)|_{\ell_m}$.

Example 2 (Example 1 cont'd). Assume we branch on x_7 and x_1 . Then $BCP(\Sigma \wedge x_7 \wedge x_1) = \{x_7, x_1\}$. *Conditioning yields:* $\Sigma|_{\{x_7, x_1\}} = \{(x_2 \vee x_3), (x_4 \vee x_5), (\bar{x}_2 \vee x_6), (\bar{x}_3 \vee \bar{x}_6), (\bar{x}_2 \vee \bar{x}_6), (\bar{x}_3 \vee x_6)\}$. *Branching on the literals $\{x_7, \bar{x}_1\}$ yields* $BCP(\Sigma \wedge x_7 \wedge \bar{x}_1) = \{x_7, \bar{x}_1, x_3\}$ and $\Sigma|_{\{x_7, \bar{x}_1, x_3\}} = \{(x_4 \vee x_5)\}$.

State-of-the-art exact model counters typically extend the CDCL architecture used in SAT solving. Unlike SAT solvers, which terminate upon finding a single model, model counters must traverse the entire solution space. To do this efficiently, they rely on three key techniques: **Clause Learning**, where the solver learns a new clause upon conflict to prune future search; **Component Caching**, which stores results of previously solved subproblems to avoid redundant computations [Sang *et al.*, 2004]; and **Component Decomposition**, which exploits independent sub-formulas $\Sigma_1, \dots, \Sigma_k$ by computing the product of their counts $\prod_{i=1}^k \|\Sigma_i\|$.

The standard model counting procedure generally follows the structure of Algorithm 1 (presented in Section 3, ignoring the modifications highlighted in blue and red). First, $BCP(\Sigma)$ is executed to gather the set of implied literals S , and the formula is simplified to $\Sigma' = \Sigma|_S$. If a conflict arises (i.e., $\Sigma' = \perp$), the function returns 0. If all clauses are satisfied (i.e., $\Sigma' = \emptyset$), the algorithm returns $2^{|Var(\Sigma)| - |S|}$, which accounts for the single model of the implied literals and the 2^k combinations of any remaining variables. Otherwise, the algorithm checks the cache. If the instance is new, it decomposes the formula into connected components. If multiple components exist, they are solved recursively, and their results are multiplied. If the formula forms a single component, the algorithm selects a branching variable x , computes the sum of the counts for the two branches ($\Sigma' \wedge x$ and $\Sigma' \wedge \bar{x}$), and caches the result. Finally, in both the component and branching cases, the computed value val is scaled by $2^{|Var(\Sigma)| - |Var(\Sigma')| - |S|}$. This factor corrects for variables that were present in Σ but eliminated during simplification (e.g., variables that became free), ensuring the count remains correct with respect to the original variable set.

Crucially, the interaction between dynamic clause learning and component caching requires careful management. When a conflict occurs, a learned clause is added to the formula. Although this clause is a logical consequence of the original formula, its presence modifies the implication landscape. As established in [Sang *et al.*, 2004], naively retrieving cached components in the presence of different learned clauses can lead to incorrect counts. We explicitly address this constraint in Section 4, where we define the necessary conditions for safe job sharing in a distributed setting.

The efficiency of the solver heavily relies on the variable selection heuristic (*chooseVariable*). Modern approaches exploit the formula's *primal graph* and its *tree decomposition* $(T, \{B_t\})$, a mapping of variables into a tree structure satisfying standard connectivity properties [Robertson and Seymour, 1986]. Although real-world instances often have high *treewidth* (defined as $\max |B_t| - 1$), heuristics that prioritize variables appearing in the “top” bags of the decomposition

significantly improve performance by fostering component decomposition [Korhonen and Järvisalo, 2021].

3 The Explicit Stack Framework

Exact model counting traverses an *AND-OR Tree*, summing results at **OR-Nodes** (decisions) and multiplying at **AND-Nodes** (decompositions). In standard DFS solvers, pending work remains implicit within the recursion stack. This implicit representation creates a major barrier for distribution, as recursion frames cannot be easily “detached” to offload tasks to other workers.

3.1 The Explicit Search State

To enable flexible *work-stealing*, $\mathcal{gDMC}$ reifies the recursion stack into a manageable data structure: the **Explicit Stack $\mathcal{S}$** . Crucially, we do not materialize the entire tree (which grows exponentially); we maintain only the *current active path*. Any node on this path with an unexplored sibling represents a potential **job** that can be offloaded.

It is important to note that $\mathcal{S}$ does not behave strictly as a standard Last-In-First-Out (LIFO) stack during distributed execution. In a purely sequential setting, a node is popped as soon as its local sub-problems are solved. In $\mathcal{gDMC}$, however, a node ν may have offloaded portions of its work to other solvers. Consequently, the node serves a dual purpose: it is both a controller for local iteration and a dependency tracker for remote jobs. Even if the local set of tasks is empty, the node must persist in the stack if it is still waiting for remote results. It acts as an aggregation point, waiting to combine the local accumulator with the results of the outstanding shared tasks. Only when all local and remote obligations are met is the node considered fully resolved and removed from $\mathcal{S}$.

Definition 1 (Stack Nodes). *The stack $\mathcal{S}$ is a sequence of nodes $\nu = \langle \mathcal{F}, v, \mathcal{I}, \mu \rangle$ comprising:*

- $\mathcal{F}$: The set of pending sub-problems to be solved locally.
- v : The local accumulator for the model count.
- $\mathcal{I}$: The set of identifiers for sub-problems offloaded.
- μ : A local scaling factor (representing variables eliminated at this level), always initialized to 1.

We distinguish two node types based on their initialization and aggregation logic. A **Decision Node (Sum)** is initialized with branches $\{\Sigma \wedge x, \Sigma \wedge \bar{x}\}$, where the accumulator v starts at 0 as the additive identity. Conversely, a **Decomposition Node (Product)** is initialized with disjoint components $\{\Sigma_1, \dots, \Sigma_k\}$ and an accumulator v starting at 1, the identity for multiplication.

Example 3 (Ex. 1 cont’d). *Branching on x_7, x_1 partitions the formula into components $\mathcal{C}_1 = \{x_4 \vee x_5\}$ and $\mathcal{C}_2 = \{(x_2 \vee x_3), (\bar{x}_2 \vee x_6), (\bar{x}_3 \vee \bar{x}_6), (\bar{x}_2 \vee \bar{x}_6), (\bar{x}_3 \vee x_6)\}$. The stack is: $\mathcal{S} = (\langle \{\Sigma|_{\bar{x}_7}\}, 0, \emptyset, 1), \langle \{\Sigma|_{x_7, \bar{x}_1}\}, 0, \emptyset, 1), \langle \{\mathcal{C}_1, \mathcal{C}_2\}, 1, \emptyset, 1 \rangle$.*

While the formal definition describes nodes as containing explicit formulas $\mathcal{F}$, materializing these is inefficient in practice. Instead, $\mathcal{gDMC}$ leverages the fact that any search tree sub-problem is uniquely determined by its path from the root.

Specifically, a task Σ_{sub} is represented implicitly by: the original formula Σ_{root} , the current **partial assignment** (unit literals from decisions and propagations), and the set of **relevant variables** restricted to the current connected component.

Consequently, when we describe “pushing a formula $\Sigma \wedge x$ ” onto the stack, the implementation simply records the decision literal x , the set of propagated unit literals, and the variables identifying the current component. The solver reconstructs the effective formula by applying the current assignment to Σ_{root} and filtering by the component’s variables. Similarly, when a job is offloaded to a remote worker, we transmit only the sequence of decisions (the “cube”) that defines the path to that node, along with the set of variables for the component, allowing the remote worker to reconstruct the exact same state locally.

3.2 The Challenge of Distributed Caching

Integrating work-stealing into model counting is complicated by **Component Caching**. The standard caching invariant requires that if a formula Σ is solved, its count $\|\Sigma\|$ is stored for immediate retrieval. However, offloading Σ to a remote worker creates a *broken dependency*: the local solver cannot wait for the result without losing parallelism, nor can it cache a “future” result without prohibitive overhead. This “hole” propagates upward, preventing all parent nodes from being cached.

To mitigate this, $\mathcal{gDMC}$ restricts sharing to the **root of the stack** (the least recently generated sub-problems). This topological constraint ensures the active frontier at the top of the stack remains entirely local and free of remote dependencies, allowing sub-trees to be computed and cached synchronously. Furthermore, anchoring sharing to the stack base preserves a low-overhead linear structure, avoiding the need to manage a complex AND-OR dependency tree in memory.

3.3 Stack Primitives

As $\mathcal{S}$ retains persistent nodes awaiting remote results, the logical frontier diverges from the physical top. We bridge this by maintaining a pointer pos to the active context, denoted $\nu_{curr} = \mathcal{S}[pos] = \langle \mathcal{F}, v, \mathcal{I}, \mu \rangle$.

- $\mathcal{S}.pushOr(\Sigma_1, \Sigma_2)$: Pushes a new **Decision Node** onto $\mathcal{S}$. This operation is valid only if pos corresponds to the physical top of the stack. We increment pos and initialize the new tuple as $\langle \{\Sigma_1, \Sigma_2\}, 0, \emptyset, 1 \rangle$.
- $\mathcal{S}.pushAnd(\{\Sigma_1, \dots, \Sigma_k\})$: Pushes a new **Decomposition Node** onto $\mathcal{S}$. This operation is valid only if pos refers to the physical top of the stack. We increment pos and initialize the tuple as $\langle \{\Sigma_1, \dots, \Sigma_k\}, 1, \emptyset, 1 \rangle$.
- $\mathcal{S}.alreadyShared(\Sigma)$: Checks if the sub-problem Σ is present in the pending formula set $\mathcal{F}$ of ν_{curr} . Returns **true** if $\Sigma \notin \mathcal{F}$ (implying Σ has been stolen or removed by a concurrent worker). Returns **false** otherwise.
- $\mathcal{S}.process(\Sigma)$: Begins local processing of Σ by removing it from $\mathcal{F}$ in ν_{curr} . If $\mathcal{F}$ becomes empty, local work for the node is complete. If the remote identifier set $\mathcal{I}$ is also empty, the node is fully resolved and popped from $\mathcal{S}$ (with pos decremented). Otherwise, the node persists

as a dependency tracker for remote results, and we only decrement pos to return control to the parent context.

- $\mathcal{S}.\text{identity}()$: Returns the neutral element for the operation associated with ν_{curr} (0 if ν_{curr} is a Decision Node, 1 if it is a Decomposition Node).
- $\mathcal{S}.\text{isShared}()$: Checks if the current node ν_{curr} has partially offloaded its work. Returns `true` if the set of remote identifiers $\mathcal{I}$ is not empty.
- $\mathcal{S}.\text{complete}(c \times \mu)$: Updates the accumulator v of ν_{curr} with the result of a sub-computation. The value to aggregate is derived from c (the raw count computed by the solver) and μ (the scaling factor, typically 2^k , for variables free in the sub-problem).

3.4 Integration in the Counting Algorithm

Algorithm 1 illustrates the integration of the $\mathcal{g}\text{DMC}$ framework into a standard DPLL-style model counter. The implementation utilizes two global controllers: the explicit stack manager $\mathcal{S}$, which maintains the search state and aggregates results, and the communication manager (referenced as co), which handles network protocols and work redistribution. The core counting logic (comprising unit propagation, component analysis, and variable branching) is shown in black and adheres to the classical formulation. The **blue blocks** mark the synchronization points with $\mathcal{S}$, while the **red block** orchestrates the interaction with remote workers to enable dynamic load balancing.

The first interaction occurs immediately upon entering the function. Before performing any propagation, the solver queries $\mathcal{S}.\text{alreadyShared}(\Sigma)$. This acts as a guard clause: if the current sub-problem Σ has been removed from the stack (stolen by another worker), the function aborts immediately by returning the neutral element $\mathcal{S}.\text{identity}()$. This prevents the local worker from solving a task that is no longer its responsibility. If the task remains, $\mathcal{S}.\text{process}(\Sigma)$ is called to mark Σ as “busy,” ensuring it cannot be stolen while the local worker is actively solving it.

Whenever the solver decomposes the problem, it registers the new sub-problems in the stack to expose the frontier of work. Specifically, before iterating over a set of connected components, the algorithm calls $\mathcal{S}.\text{pushAnd}$ to create a new decomposition node containing these components. Similarly, prior to making a decision on a variable x , $\mathcal{S}.\text{pushOr}$ is invoked to record the two resulting branches ($\Sigma \wedge x$ and $\Sigma \wedge \bar{x}$). These operations ensure that the explicit stack always reflects the current available tasks for potential thieves.

After the recursive calls return, the local solver has computed a partial result val . However, before returning this value, it must query $\mathcal{S}.\text{isShared}()$. If this check returns `true`, it indicates that sibling tasks were offloaded and the local result is incomplete. In this case, the algorithm calls $\mathcal{S}.\text{complete}$ to aggregate val into the persistent stack node and returns $\mathcal{S}.\text{identity}()$, ensuring that the parent recursive call aggregates only the neutral element. Conversely, if the node was not shared, it implies the sub-problem was processed entirely locally. The stack node is effectively popped, and the computed val is returned normally to the parent to continue the standard aggregation.

Algorithm 1: $\text{count}(\Sigma)$

Input : A CNF formula Σ

Output: The number of models $\|\Sigma\|$

```

1 if  $\mathcal{S}.\text{shouldPoll}()$  and  $\text{co}.\text{shareRequest}()$  then
2    $\text{co}.\text{transferWorkToThieves}(\mathcal{S})$ ;
3 if  $\mathcal{S}.\text{alreadyShared}(\Sigma)$  then return  $\mathcal{S}.\text{identity}()$ ;
4  $\mathcal{S}.\text{process}(\Sigma)$ ;
5  $S \leftarrow \text{BCP}(\Sigma)$ ;  $\Sigma' \leftarrow \Sigma|_S$ ;
6 if  $\Sigma' = \perp$  then return 0;
7 if  $\Sigma' = \emptyset$  then return  $2^{|\text{Var}(\Sigma)| - |\mathcal{I}|}$ ;
8 if  $\text{inCache}(\Sigma')$  then return  $\text{getCache}(\Sigma')$ ;
9  $\text{Components} \leftarrow \text{getConnectedComponents}(\Sigma')$ ;
10 if  $|\text{Components}| > 1$  then
11    $\mathcal{S}.\text{pushAnd}(\text{Components})$ ;
12    $val \leftarrow 1$ ;
13   for  $C \in \text{Components}$  do  $val \leftarrow val \times \text{count}(C)$ ;
14   if  $\mathcal{S}.\text{isShared}()$  then
15      $\mathcal{S}.\text{complete}(val \times 2^{|\text{Var}(\Sigma)| - |\text{Var}(\Sigma')| - |\mathcal{I}|})$ ;
16     return  $\mathcal{S}.\text{identity}()$ ;
17   return  $val \times 2^{|\text{Var}(\Sigma)| - |\text{Var}(\Sigma')| - |\mathcal{I}|}$ ;
18  $x \leftarrow \text{chooseVariable}(\Sigma')$ ;
19  $\mathcal{S}.\text{pushOr}(\Sigma' \wedge x, \Sigma' \wedge \bar{x})$ ;
20  $val \leftarrow \text{count}(\Sigma' \wedge x) + \text{count}(\Sigma' \wedge \bar{x})$ ;
21 if  $\mathcal{S}.\text{isShared}()$  then
22    $\mathcal{S}.\text{complete}(val \times 2^{|\text{Var}(\Sigma)| - |\text{Var}(\Sigma')| - |\mathcal{I}|})$ ;
23   return  $\mathcal{S}.\text{identity}()$ ;
24  $\text{addToCache}(\Sigma', val)$ ;
25 return  $val \times 2^{|\text{Var}(\Sigma)| - |\text{Var}(\Sigma')| - |\mathcal{I}|}$ ;

```

We defer the discussion of the **red block** in Algorithm 1 to Section 4, where we detail the Master-Worker coordination and the network protocols required to interconnect individual solver instances.

4 Distributed Orchestration

Algorithm 2 employs a centralized master-worker architecture. Initialization (lines 1–3) follows the `discount` protocol [Xu *et al.*, 2025]: the Master broadcasts Σ and enforces the cluster’s best tree decomposition as a static branching order. The root task ($idTask = 0$) is then assigned to a single worker (line 5). Once initialized, the full problem is assigned to a single root worker via a blocking `sendJob` call with an initial task identifier $idTask = 0$ (line 5), while remaining workers enter an *idle* state. To simplify synchronization, $\mathcal{g}\text{DMC}$ implements a *point-to-group sharing strategy*: the Master targets only one active worker for work-stealing at a time, tracked by the variable *contacted*.

While at least one worker is active, the Master monitors the cluster. If *contacted* = `nil`, the Master selects a new target and sends a work request. Upon a positive non-blocking `isResponding` check, the Master facilitates direct transfers from the target to multiple idle workers (lines 11–13), assigning each sub-task a unique *idTask* for aggregation. This one-to-many transfer mitigates latency and rapidly re-integrates workers. Finally, completed workers return to the *idle* set; if the *contacted* worker finished, the pointer is reset.

Algorithm 2: gDMC Global Orchestration

Input : A CNF formula Σ and a set of workers $\mathcal{W}$
Output: The number of models $\|\Sigma\|$

```

// Initialization Phase
1  $\Sigma \leftarrow \text{preprocess}(\Sigma)$ ;
2  $\text{broadcast}(\Sigma, \mathcal{W})$ ;
3  $\text{coordinateHeuristics}(\mathcal{W})$ ;

// Distributed Counting
4 Select root worker  $w_r \in \mathcal{W}$ ;
5  $\text{sendJob}(\Sigma, w_r, 0)$ ;  $\text{idTask} \leftarrow 1$ ;
6  $\text{idle} \leftarrow \mathcal{W} \setminus \{w_r\}$ ;  $\text{contacted} \leftarrow \text{nil}$ ;
7 while  $\text{idle} \neq \mathcal{W}$  do
    // Select work source
    8 if  $\text{contacted} = \text{nil}$  and  $\text{idle} \neq \mathcal{W}$  then
        9  $\text{contacted} \leftarrow \text{select one from } (\mathcal{W} \setminus \text{idle})$ ;
        10  $\text{requestWork}(\text{contacted})$ ;

    // Offload tasks to idle pool
    11 while  $\text{isResponding}(\text{contacted})$  and
         $\text{hasWork}(\text{contacted})$  and  $\text{idle} \neq \emptyset$  do
        12  $w_{\text{idle}} \leftarrow \text{pop}(\text{idle})$ ;
        13  $\text{pairWorkers}(\text{contacted}, w_{\text{idle}}, \text{idTask}++)$ ;

    // Update worker states
    14 foreach  $w \in \mathcal{W} \setminus \text{idle}$  do
        15 if  $\text{isFinished}(w)$  then
            16  $\text{idle} \leftarrow \text{idle} \cup \{w\}$ ;
            17 if  $w = \text{contacted}$  then  $\text{contacted} \leftarrow \text{nil}$ ;

18 return  $\text{reduceResults}(\mathcal{W})$ ;
    
```

When all workers return to the idle state, the Master initiates result aggregation (line 18). Unlike DMC, which streams partial expressions to the Master, gDMC maintains stacks locally. To resolve a task, a worker must complete any previously shared nodes. When two workers communicate, the “thief” tracks the “victim” to return the computed value once the sub-stack for that idTask is resolved. This propagates values back through the dependency chain until the root task ($\text{idTask} = 0$) is completed.

4.1 Worker-Master Integration

We now detail the integration logic (the red block in Algorithm 1) that links the individual solver to the global orchestrator. To minimize communication overhead, the function `shouldPoll` acts as a throttle, ensuring the worker polls for Master requests only periodically (e.g., every $k = 1,000$ decisions) to avoid system-call degradation.

The functions `shareRequest` and `transferWorkToThiefs` manage the worker’s network interaction. `shareRequest` checks for an incoming `requestWork` signal. If detected, the worker begins a blocking communication to indicate if it has viable work. A worker may refuse to share if it is processing its last available formula or if the remaining sub-problems are too small to justify the communication cost. To prevent a “ping-pong” effect between active and idle states, we enforce a granularity threshold; for instance, formulas with fewer than 30 variables are solved locally rather than shared.

Finally, `transferWorkToThiefs` handles the offloading of a pending node to a thief. The victim performs a blocking

send of the task data (variables and partial assignments) to the thief. Crucially, the victim does not wait for the result. It simply adds the assigned idTask to the set of remote identifiers $\mathcal{I}$ of the current stack node (marking it as persistent) and immediately resumes its local search.

4.2 Distributed Caching and Soundness

Sharing arbitrary nodes from the stack is not permissible. As previously mentioned, sharing is restricted to nodes whose parents have already been shared. However, this topological constraint is insufficient to prevent *cache inconsistency*, particularly when dealing with unsatisfiable sub-formulas.

Cache inconsistency occurs when the value c stored in the cache for a formula Σ satisfies $c \neq \|\Sigma\|$. This phenomenon is a byproduct of conflict-driven clause learning. While learned clauses are treated as part of the formula for BCP, they are not necessarily part of the original local sub-problem Σ . If a learned clause α is derived from variables in $\text{Var}(\Sigma)$ but is not a logical consequence of Σ (i.e., $\Sigma \not\models \alpha$), using α to simplify Σ can result in an incorrect model count.

This issue is well-documented in the context of component caching [Sang *et al.*, 2004]: when a sub-formula is found to be unsatisfiable, any cache entries added after the initial conflict must be invalidated. In a local solver, this cleaning is straightforward. In a distributed setting, however, sharing a sub-problem may prevent a worker from ever detecting the specific conflict that would have triggered a cache invalidation, leading to the propagation of “poisoned” results.

Example 4 (Example 1 cont’d). Suppose we first solve branch $\bar{x}_7$ completely (count 12), then explore x_7 . Inside x_7 , we branch on x_1 . We share $\bar{x}_1$ as Task 1, and after decomposing the x_1 branch, we share component C_2 as Task 2. Then $\mathcal{S} = (\langle \emptyset, 12, \emptyset, 0 \rangle, \langle \emptyset, 0, \{1\}, 0 \rangle, \langle \{C_1\}, 1, \{2\}, 1 \rangle)$.

Assume that solving $\Sigma|_{\bar{x}_7}$ generated the learnt clause $\alpha = (\bar{x}_1 \vee x_4)$. When solving the local component C_1 , this clause forces x_4 to be true, reducing C_1 ’s count to 2 instead of the correct 3. If the remote component $C_2 \models \perp$, the solver detects the conflict too late: the corrupt count for C_1 (2) may already be computed and cached, causing incorrect results in future lookups, or worse, the inconsistency might never be detected.

We demonstrate that such inconsistencies only arise in branches containing an unsatisfiable connected component, thus ensuring soundness.

Proposition 1. Let ρ be the set of unit literals (the current path) such that Σ is a connected component of $\Sigma_{\text{root}}|_{\rho}$. Let c be the model count computed for Σ . If every connected component of $\Sigma_{\text{root}}|_{\rho}$ is satisfiable, then $c = \|\Sigma\|$.

Proof. Let Δ be the learned clauses. Assume for contradiction a cache inconsistency ($c \neq \|\Sigma\|$) occurs even if all components of $\Sigma_{\text{root}}|_{\rho}$ are satisfiable. This implies the existence of $\alpha \in \Delta|_{\rho}$ where $\text{Var}(\alpha) \subseteq \text{Var}(\Sigma)$ but $\Sigma \not\models \alpha$.

By construction, every clause learned is a logical consequence of the original formula, implying $\Sigma_{\text{root}}|_{\rho} \models \alpha$. Since $\Sigma_{\text{root}}|_{\rho}$ is partitioned into independent connected components $\{\Sigma, C_1, \dots, C_n\}$, the formula is logically equivalent to the conjunction of these components: $\Sigma_{\text{root}}|_{\rho} \equiv \Sigma \wedge C_1 \wedge \dots \wedge C_n$.

If all components are satisfiable, α must be a logical consequence of Σ alone. For α to be a consequence of the global formula without being a consequence of Σ , it would necessarily require the unsatisfiability of at least one other component C_i to “force” the implication through a contradiction. Since all components are satisfiable by hypothesis, $\Sigma \models \alpha$ must hold. This contradicts the assumption that $\Sigma \not\models \alpha$. $\square$

Cache inconsistency has been addressed in distributed model counting. In `CountAntom` [Burchard *et al.*, 2015], the authors ignore the problem; however, their architecture differs. `CountAntom` manages a single global search tree at the Master, whereas our approach involves multiple workers independently constructing portions of the search space. In DMC, this issue does not arise because `D4` invokes a SAT solver at every decision node to ensure branch satisfiability.

However, most modern model counters forgo satisfiability checks at every decision node to avoid computational overhead. This design choice complicates the direct integration of our distributed framework with such solvers, as unverified branches can lead to cache inconsistencies. Fortunately, this can be mitigated by ensuring that all connected components of an AND node are verified as satisfiable before the node is marked as eligible for sharing. While this introduces a cost exceeding standard BCP, the overhead remains negligible in practice because the solver is not invoked at every decision. Moreover, since work sharing is restricted to the upper levels of the search tree, the number of AND nodes requiring verification is relatively small. Consequently, the total number of calls to the SAT oracle is minimized, ensuring that these safety checks do not degrade global search performance.

4.3 Worker Invocation

Following the protocol established by `DisCount` [Xu *et al.*, 2025], our framework invokes underlying model counters in *assumptions mode*, a technique derived from incremental SAT solving [Nadel and Ryzhichin, 2012]. In this configuration, the sub-problem assigned to a worker is defined by a set of *assumptions* (literals asserted only for the duration of a specific solver invocation). This approach is particularly advantageous for distributed model counting as it allows the underlying solver to retain its internal state across successive tasks. Since the base formula Σ remains static, workers can preserve variable activity data and *learned clauses*, which remain logically valid for any sub-problem derived from Σ . Furthermore, maintaining a persistent component cache across multiple counting queries has been shown to be highly beneficial [Lagniez and Marquis, 2019]. By utilizing incremental invocations, `gDMC` enables workers to reuse cache entries and variable weights, significantly accelerating the processing of related sub-problems.

Unlike distributed counters that treat each task as an isolated query, each worker in `gDMC` manages a local collection of stacks. Upon receiving a sub-problem with a specific *idTask*, a worker initializes a corresponding stack to track the search and any internal decompositions. Rather than returning partial counts immediately to the Master, workers accumulate these solved sub-stacks locally to avoid frequent transfers of large arithmetic values. Once the Master deter-

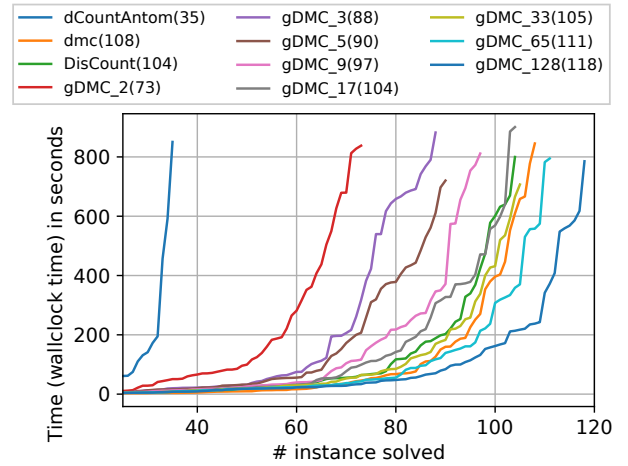


Figure 1: Solved instances over time for `dCountAntom`, `dmc`, `DisCount`, and `gDMC` configurations. Parentheses in the legend indicate the total number of solved instances.

mines that the entire search space has been covered (signaled by all workers entering the *idle* state) it triggers the final reduction phase. During this phase, workers perform a decentralized reduction, propagating computed values back to the original victims via the stored *idTask* dependencies. This strategy minimizes network congestion and ensures that the final model count for the entire formula (*idTask* = 0) is only resolved once all dependencies are satisfied.

5 Experiments

Our framework is implemented in C++ as an extension of `DisCount`, leveraging concepts [Reis and Stroustrup, 2006] and templates. We utilize concepts to define strict interfaces for the model counter, enabling seamless interchangeability of the stack manager and communication protocols. `gDMC` supports arbitrary-precision aggregation. Furthermore, this template-based architecture simplifies MPI integration by eliminating the need for invasive modifications to the solver’s build configuration. The source code and experimental logs are available on Zenodo (<https://zenodo.org/records/20157902>).

We use `Glucose 3.0` [Audemard *et al.*, 2013] in incremental mode and `FlowCutter` [Hamann and Strasser, 2018] (PACE 2017) for tree decompositions (10s budget). Preprocessing utilizes `B+E` [Lagniez *et al.*, 2016] with the `equiv` option (vivification, backbone, occurrence elimination). To prevent worker idle time caused by slow, serial definability checks, we limit this phase to 5 s. Finally, we upgraded `SharpSAT-TD` [Korhonen and Järvisalo, 2021] to support incremental assumptions.

We evaluated our approach against state-of-the-art distributed model counters `dCountAntom` [Burchard *et al.*, 2016], `dmc` [Lagniez *et al.*, 2018], and `DisCount` [Xu *et al.*, 2025]. For a fair and accurate comparison, `DisCount` use `SharpSAT-TD` as the base solver. For our experiments, work-stealing requests are issued periodically every $k = 1,000$ decisions, and we do not share formulas with fewer

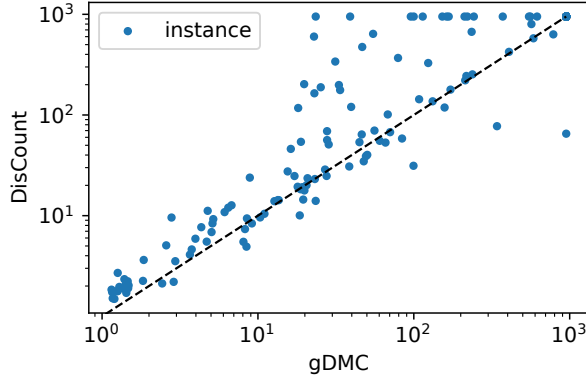


Figure 2: Scatter plot comparing solving times (in seconds) of DisCount (y -axis) and gDMC (x -axis) on a logarithmic scale. Each point corresponds to a single benchmark instance.

than 30 variables. The benchmarks were taken from the most recent competition [Fichte and Hecher, 2025].

All experiments ran on a cluster of thirty-two Intel® Xeon® E5-2643 v4 CPUs at 3.30 GHz, with Rocky Linux 9.5 (Linux kernel 5.14). Nodes are connected by 1 GiB/s Ethernet. The software environment used GCC 11.5 and Open MPI 5.1.0a1. Finally, a wall-clock time limit of 900 seconds and a memory limit of 32 GiB were imposed on each run.

Figure 1 presents a cactus plot of solved instances over time for dCountAntom, DMC, DisCount, and gDMC. To assess the scalability of gDMC, we evaluated configurations of 2, 3, 5, 9, 17, 33, 65 and 128 cores. Since gDMC requires a dedicated master node, these configurations correspond to 1, 2, 4, ..., 64, and 127 active workers, respectively. The competing solvers dCountAntom, DMC, and DisCount were executed using the full 128-core allocation.

The results demonstrate that increasing the core count significantly reduces the solving time for gDMC, maximizing the number of instances solved within the limit. In the 128-core comparison, dCountAntom is the least efficient (35 solved), followed by DisCount (104) and DMC (108), with gDMC achieving the best performance in both count and speed. Notably, contrary to findings in [Xu *et al.*, 2025], DisCount trails slightly behind DMC in this evaluation. We attribute this discrepancy to two distinct factors. First, the increased difficulty of this year’s benchmarks hinders DisCount’s ability to effectively partition the search space into cubes. Second, DMC uses double-precision arithmetic, thereby avoiding the computational overhead of the arbitrary-precision arithmetic employed by DisCount.

Figure 2 compares solving times of DisCount and gDMC; points above the diagonal favor gDMC. gDMC solves more instances, evidenced by top-border points (DisCount timeouts), and is consistently faster. This gap widens with instance difficulty. We attribute this efficiency to two factors: gDMC avoids cube-generation overhead on simple instances, while dynamic work-stealing balances load on complex ones, mitigating the long-tail latency of static partitioning.

Figure 3 depicts the distribution of speedups achieved by gDMC as computational resources scale from 2 to 128 cores.

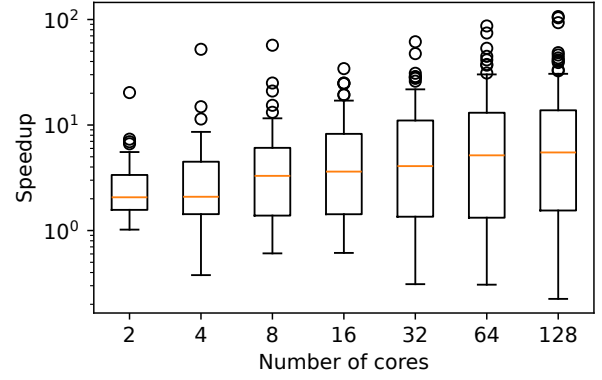


Figure 3: Distribution of speedups achieved by gDMC across varying core counts, relative to the single-worker baseline (2 cores).

The y -axis represents the speedup factor relative to a baseline configuration of 2 cores (1 master and 1 worker). For any instance solved in t_n seconds using n cores, the speedup is computed as $\frac{\min(t_2, 900)}{t_n}$, where t_2 is the baseline solving time. Since t_2 is capped at the 900-second timeout, these reported values constitute a conservative lower bound on the actual speedup for instances that time out in the baseline configuration. The results demonstrate robust scalability. We observe a consistent upward trend in the median speedup (indicated by the orange line) as the number of cores increases, confirming that gDMC effectively utilizes additional worker nodes. Furthermore, the presence of high-performing outliers indicates that for many complex instances, the framework achieves near-linear speedups. This suggests that the dynamic work-stealing strategy successfully mitigates the idle time and communication overhead typically associated with distributed exact counting.

6 Conclusion and Perspectives

In this paper, we presented gDMC, a generic, solver-agnostic framework for DPLL-style distributed exact model counting. By leveraging C++ concepts and templates, we decoupled distributed orchestration from the core solving logic, enabling zero-overhead integration with state-of-the-art counters like SharpSAT-TD. Our approach employs a novel *point-to-group* work-stealing strategy and a local aggregation mechanism that ensures soundness and cache consistency without the bottleneck of global result streaming.

Future work focuses on three key directions. First, we plan to integrate probabilistic counters such as GANAK. Second, we aim to exploit our solver-agnostic design for DPLL-style counters to implement a heterogeneous portfolio, running distinct solvers concurrently to dynamically leverage their respective strengths on sub-problems. Finally, we intend to evaluate the scalability of our architecture on high-performance computing clusters with thousands of cores, testing the limits of our synchronization protocols in massively parallel environments.

Acknowledgments

This work has been partly supported by the CERADOC project of the French National Agency for Research (ANR-25-CE23-3078), National Natural Science Foundation of China (No. 62532014), Jilin Province Science and Technology Department Project (20240602005RC), Scientific Research Project of the Education Department of Jilin Province (JJKH20250334KJ).

References

- [Audemard *et al.*, 2013] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.
- [Baluta *et al.*, 2019] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1249–1264. ACM, 2019.
- [Burchard *et al.*, 2015] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #sat solving. In Marijn Heule and Sean A. Weaver, editors, *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*, pages 46–61. Springer, 2015.
- [Burchard *et al.*, 2016] Jan Burchard, Tobias Schubert, and Bernd Becker. Distributed parallel #sat solving. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 326–335. IEEE Computer Society, 2016.
- [Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.
- [Domshlak and Hoffmann, 2006] Carmel Domshlak and Jörg Hoffmann. Fast probabilistic planning through weighted model counting. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 243–252. AAAI, 2006.
- [Dueñas-Orsorio *et al.*, 2017] Leonardo Dueñas-Orsorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 4488–4494. AAAI Press, 2017.
- [Feiten *et al.*, 2012] Linus Feiten, Matthias Sauer, Tobias Schubert, Alexander Czutro, Eberhard Böhl, Ilia Polian, and Bernd Becker. #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*, pages 49–54. IEEE Computer Society, 2012.
- [Fichte and Hecher, 2025] Johannes K. Fichte and Markus Hecher. The model counting competitions 2021–2023, 2025.
- [Fichte *et al.*, 2021] Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Parallel model counting with CUDA: algorithm engineering for efficient hardware utilization. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [Girol *et al.*, 2021] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not all bugs are created equal, but robust reachability can tell the difference. In Alexandra Silva and K. Rustan M. Leino, editors, *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*, pages 669–693. Springer, 2021.
- [Hamann and Strasser, 2018] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics*, 23, 2018.
- [Korhonen and Järvisalo, 2021] Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 8:1–8:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017.
- [Lagniez and Marquis, 2019] Jean-Marie Lagniez and Pierre Marquis. 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*, pages 1536–1543. AAAI Press, 2019.

- [Lagniez *et al.*, 2016] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 751–757. IJCAI/AAAI Press, 2016.
- [Lagniez *et al.*, 2018] Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. DMC: A distributed model counter. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1331–1338. ijcai.org, 2018.
- [Mei *et al.*, 2024] Jingyi Mei, Marcello M. Bonsangue, and Alfons Laarman. Simulating quantum circuits by model counting. In Arie Gurfinkel and Vijay Ganesh, editors, *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*, pages 555–578. Springer, 2024.
- [Nadel and Ryvchin, 2012] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *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*, pages 242–255. Springer, 2012.
- [Reis and Stroustrup, 2006] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 295–308. ACM, 2006.
- [Robertson and Seymour, 1986] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
- [Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. 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*, 2004.
- [Sang *et al.*, 2005] Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 475–482. AAAI Press / The MIT Press, 2005.
- [Sharma *et al.*, 2019] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019.
- [Teuber and Weigl, 2021] Samuel Teuber and Alexander Weigl. Quantifying software reliability via model-counting. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2021.
- [Toda, 1991] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991.
- [Xu *et al.*, 2025] Zhenghang Xu, Minghao Yin, and Jean-Marie Lagniez. An embarrassingly parallel model counter. In *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning, KR 2025, Melbourne, Australia. November 11-17, 2025*, 2025.