DMC: A Distributed Model Counter
Jean-Marie Lagniez, Pierre Marquis, Nicolas Szczepanski

To cite this version:
Jean-Marie Lagniez, Pierre Marquis, Nicolas Szczepanski. DMC: A Distributed Model Counter. 27th International Joint Conference on Artificial Intelligence (IJCAI’18), Jul 2018, Stockholm, Sweden. pp.1331-1338, 10.24963/ijcai.2018/185. hal-03300776

HAL Id: hal-03300776
https://univ-artois.hal.science/hal-03300776
Submitted on 25 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
DMC: A Distributed Model Counter

Jean-Marie Lagniez\textsuperscript{1}, Pierre Marquis\textsuperscript{1,2}, Nicolas Szczepanski\textsuperscript{1}
\textsuperscript{1} CRIL-CNRS UMR 8188, Universit\textsuperscript{e} d’Artois, Lens, France
\textsuperscript{2} Institut Universitaire de France
lagniez@cril.fr, marquis@cril.fr, szczepanski@cril.fr

Abstract

We present and evaluate DMC, a distributed model counter for propositional CNF formulae based on the state-of-the-art sequential model counter D4. DMC can take advantage of a (possibly large) number of sequential model counters running on (possibly heterogeneous) computing units spread over a network of computers. For ensuring an efficient workload distribution, the model counting task is shared between the model counters following a policy close to work stealing. The number and the sizes of the messages which are exchanged by the jobs are kept small. The results obtained show DMC as a much more efficient counter than D4, the distribution of the computation yielding large improvements for some benchmarks. DMC appears also as a serious challenger to the parallel model counter CountAntom and to the distributed model counter dCountAntom.

1 Introduction

Model counting (aka the \#SAT problem) is the task consisting in computing the number of models of a given propositional formula $\Sigma$ (typically in CNF) over the set of its variables. This task is of tremendous importance to many AI problems, including probabilistic inference [Sang et al., 2005; Chavira and Darwiche, 2008; Apel and Brafman, 2012] and forms of planning [Palacios et al., 2005; Domshlak and Hoffmann, 2006]. It has also many applications outside AI, especially in the domains of model checking and hardware testing [Feiten et al., 2012; Klebanov et al., 2013].

The significance of \#SAT explains why much effort has been spent for the last decade in developing new algorithms for model counting (either exact or approximate) which prove practical for larger and larger instances [Samer and Szeider, 2010; Chakraborty et al., 2016]. Especially, several (sequential, exact) model counters have been implemented and evaluated, including search-based model counters, like Cachet [Sang et al., 2004] and sharpSAT [Thurley, 2006], as well as compilation-based model counters, like C2D [Darwiche, 2001; 2004], SDD [Darwiche, 2011; Oztok and Darwiche, 2015], Dsharp [Muise et al., 2012], and D4 [Lagniez and Marquis, 2017a]. However, model counting is computationally hard ($\#P$-complete), and actually much harder in practice than satisfiability (the SAT problem). Hence the solving of many instances corresponding to real problems still remains out of reach.

In order to solve more instances within a reasonable amount of time, a basic approach from a technological side consists in running existing model counters on more efficient processing units. Increasing clock rates goes through improving the level of integration of transistors on chips. Moore’s law, stated in the mid sixties, is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years. But it is known that Moore’s law will “die” at some point since the size of atoms is a fundamental barrier. Thus, most semiconductor industry forecasters, including Gordon Moore himself, expect Moore’s law will end by around 2025. In order to manage CPU power dissipation, processor makers now favor multi-core chip designs.

Such a strategy has been followed with some success for model counting, as exemplified by the performances of the parallel model counter CountAntom [Burchard et al., 2015]. In this model counter multiple threads concurrently compute the number of models of the input formula while sharing the conflict clauses learnt and the cached sub-formulae encountered during the search (see [Burchard et al., 2015] for details). In CountAntom a specific caching scheme (so-called “laissez-faire caching”) is exploited to enable the cores to share a common cache while ensuring that the numbers of models associated with sub-formulae and stored in the cache are correct. However, the limit concerning the level of integration of transistors on chips which prevents from arbitrary large clock speed improvements also applies to multi-core CPUs, so that the number of cores that can be integrated onto a single chip cannot grow as much as one would expect. Practically speaking, one can nowadays buy many-core processors based on hundreds, but not on thousands cores.

Going further requires making an additional step, from multi-threaded parallelism to distributed parallelism, the objective being then to benefit from the computational power of a very large number of possibly heterogeneous computers connected through a network. Of course, the absence of memory shared by the processing units in this general case imposes some constraints (there is no efficient way to exchange
pieces of information between cores located in different computers because information exchange requires message passing through a network). Focused on the model counting issue, a pioneering work in that direction is reported in [Burckard et al., 2016]; this paper describes the distributed, parallel model counter dCountAntom based on CountAntom and extending it by adding a message passing layer enabling a master counter to share work with slave solvers which are instances of CountAntom.

In this paper, we present a new distributed model counter, called DMC, based on a quite different, yet more flexible architecture than the one used by dCountAntom. Basically, the hierarchical architecture of dCountAntom prevents a slave solver from asking help to other slave solvers, which would nevertheless make sense whenever the sub-formula it works on proves hard to be solved; this limitation is overcome in DMC. In addition, for a sake of efficiency, the numbers and the sizes of the messages which are transmitted in DMC are limited: unlike what happens in dCountAntom neither the conflict clauses which are detected nor the sub-formulae for which some help is expected are explicitly communicated. The sequential #SAT solver on which the implementation of DMC is based, namely the state-of-the-art model counter D4, is also different from the one (called Antom, [Schubert et al., 2010]) used by dCountAntom (and by CountAntom). The empirical results we obtained show DMC as a much more efficient counter than D4, the distribution of the computation leading to significant time savings and yielding large improvements for some benchmarks. We also compared DMC with CountAntom and dCountAntom, and again, empirically, the computational benefits were often very significant. The binary code of DMC is available from the web page of the Compile! project, at www.cril.fr/KC/

2 Formal Preliminaries

Let \( \mathcal{L} \) be a language for propositional logic defined inductively from a countable set \( \mathcal{P} \) of propositional variables, the usual connectives \( (\neg, \lor, \land) \) and including the Boolean constants \( \top \) and \( \bot \). A literal \( \ell \) is a variable \( \ell = x \) from \( \mathcal{P} \) or a negated one \( \ell = \neg x \). A clause is a disjunction of literals or \( \bot \), and it is also viewed as the set of its 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 \( \omega \) is a mapping from \( \mathcal{P} \) to \( \{0, 1\} \). \( \omega \) is a model of a formula \( \Sigma \) when \( \Sigma \) is interpreted to 1 (true) by \( \omega \). An interpretation \( \omega \) is often represented by the set of literals \( \omega \) is a model of them. \( \models \) denotes logical entailment and \( \equiv \) logical equivalence. For any formula \( \Sigma \) from \( \mathcal{L} \), \( \text{Var}(\Sigma) \) is the set of variables from \( \mathcal{P} \) occurring in \( \Sigma \), and \( ||\Sigma|| \) is the number of models of \( \Sigma \) over \( \text{Var}(\Sigma) \).

The primal graph of a CNF formula \( \Sigma \) is the (undirected) graph where vertices correspond to the variables of \( \Sigma \) and an edge connecting two variables exists whenever one can find a clause of \( \Sigma \) where both variables occur. Every connected component of this graph (i.e., a maximal subset of vertices which are pairwise connected by a path) corresponds to a sub-set of clauses of \( \Sigma \), referred to as a connected component of the formula \( \Sigma \).

BCP denotes a Boolean Constraint Propagator, which is a key ingredient of many preprocessors and solvers. \( \text{BCP}(\Sigma) \) returns \( \{\emptyset\} \) if there exists a unit refutation (i.e., a derivation of the empty clause using unit propagation only) from the clauses of the CNF formula \( \Sigma \), and \( \text{BCP}(\Sigma) \) returns the set of literals (unit clauses) which are derived from \( \Sigma \) using unit propagation in the remaining case.

3 DMC: A Distributed Model Counter

Our (exact) model counter DMC is a distributed algorithm associating with an input CNF formula \( \Sigma \) its number of models \( ||\Sigma|| \) computed by taking advantage of a fixed, yet possibly large number of processing units, which can be spread over a computer network. Since our purpose is to get a distributed model counter which does not necessarily boil down to a multithreaded parallel model counter, one does not assume any memory share between the processing units in DMC.

3.1 The Architecture of DMC

Organization. Within DMC, the computation of \( ||\Sigma|| \) from \( \Sigma \) relies on \( n+1 \) processing units with \( n \geq 2 \): one master \( m \) and \( n \) workers \( w_1, \ldots, w_n \). In our experiments, each worker \( w_i \) is an instance of an avatar of the state-of-the-art top-down model counter D4 [Lagniez and Marquis, 2017a]. D4 is a compilation-based counter, which associates with \( \Sigma \) an equivalent Decision-DNNF representation [Darwiche, 2001; 2004], and \( ||\Sigma|| \) can be computed efficiently from such a representation. D4 is based on a SAT solver which exploits assumptions. Assumptions include the assignments of the decision variables which are considered, but also the clauses which are learnt at each call and which are kept for the subsequent calls (this has a significant impact on the efficiency of the whole process as it is the case for SAT solving [Audemard et al., 2013]). D4 takes advantage of the techniques used in other top-down model counters for efficiency reasons (mainly, disjoint component analysis, conflict analysis and non-chronological backtracking, component caching) but exploits specific decomposition heuristics (see [Lagniez and Marquis, 2017a] for details). As the other top-down model counters, D4 computes the number of models of its input formula by developing a search tree containing two types of internal nodes: decision nodes and decomposable and non-decomposable nodes. Decision nodes correspond to variable/truth value assignments, while decomposable nodes correspond to the discovery of disjoint components in the primal graph of the current formula. Promoting the decomposition into disjoint components using suited branching heuristics appears as a key ingredient for efficient model counting.

Workers. Each worker \( w_i \) of DMC has at start its own copy \( \Sigma_i \) of the input CNF formula \( \Sigma \) once vivified [Piette et al., 2008], so that all \( \Sigma_i \) are identical and equivalent to \( \Sigma \) at the beginning. The CNF formula \( \Sigma_i \) is completed during the computation with the clauses learnt by the underlying SAT solver (if any) until the worker terminates. Learnt clauses are marked as such. Since those clauses are logical consequences of \( \Sigma \), the CNF formulae \( \Sigma_i \) stored by the workers remain logically equivalent to \( \Sigma \) during the computation (but they are not pairwise identical in general because of the learnt clauses.
that usually differ). The addition of clauses to \( \Sigma_i \) as soon as they are learnt typically increases the set of clauses which can be derived from \( \Sigma_i \) using unit propagation only. Whenever a worker \( w_i \) is busy, its job consists in counting the number of models of the connected component \( C \) of \( \Sigma_i \) under a given partial assignment \( \gamma \), where \( V \) is the set of variables of \( C \) (for the sake of simplicity, one says that \( w_i \) computes \( ||\Sigma|| \) w.r.t. \((\gamma, V)\)). The partial assignment \( \gamma \) consists of literals based on decision variables or obtained using unit propagation from those literals. As with previous counters, when the current set of clauses (i.e., \( C \) conditioned by \( \gamma \)) can be partitioned into two subsets that do not share any variable, a decomposition node is added to the search tree developed by \( w_i \) (one can count separately the numbers of models of the two disjoint components of \( C \) and then multiply these numbers to get the number of models of \( C \)). Each job done by \( w_i \) is thus characterized by a pair \((\gamma, V)\), associated with a unique identifier of the form \( $k = wid(\gamma, V) \). In general, a number of jobs \((\gamma, V)\) will be considered by each worker \( w_i \) during the overall computation process.

What make the workers used in \( \text{DMC} \) different from standard sequential model counters are threefold: (1) the CNF instances on which each avatar \( w_i \) of \( \mathcal{D}4 \) is run are not given explicitly, but must be computed during a preliminary step from the CNF formula \( \Sigma_i \), a partial assignment \( \gamma \) (gathering the truth values assigned to some decision variables) and a set \( V \) of variables (characterizing the clauses on which the focus must be laid); when run on a given instance, instead of systematically returning a number of models, every worker returns an arithmetic expression based on \( + \) and \( \times \), non-negative integers and identifiers of jobs of the form \( $k \) (the identifiers corresponding to expressions or numbers of models returned by other workers to which the worker under consideration delegated some parts of the work to be done for solving the current instance), (2) since for the model counting purpose there is no need to generate a compiled form, no Decision-DNNF representation is actually computed, (3) some preprocessing techniques reported in [Lagniez and Marquis, 2017b] are first applied to the instance on which the model counter is run afterwards; for the sake of flexibility, it is not mandatory that all the avatars of the \#SAT solver used in \( \text{DMC} \) exploit the same preprocessing techniques; more generally, avatars of other (compilation-based or search-based) top-down model counters, like \text{Cachet} or \text{sharpSAT}, could be used instead of avatars of \( \mathcal{D}4 \), provided that \( (1) \) is ensured.

**Master.** The role of the master is to ensure some connections between the workers, in such a way that when \( w_i \) is busy (i.e., \( w_i \) is currently involved in the computation of \( ||\Sigma|| \) w.r.t. some \((\gamma, V)\)) and \( w_j \) is available (idle and ready to work), \( w_i \) may ask \( w_j \) to compute for it \( ||\Sigma|| \) w.r.t. \((\gamma', V')\) where \( \gamma' \) is an extension of \( \gamma \) (i.e., \( \gamma' \) contains all the elementary assignments of variables to truth values that occur in \( \gamma \), and possibly additional ones), and \( V' \) is a subset of \( V \). To do so, \( m \) maintains a list of all the workers, where each worker \( w_i \) is associated with a flag indicating whether \( w_i \) is busy or available.

At the beginning, the master process \( m \) first activates worker \( w_1 \); this worker \( w_1 \) starts counting the models of \( \Sigma \) (i.e., the first job of \( w_1 \) is to compute \( ||\Sigma|| \) w.r.t. \( (\{\}, \text{Var}(\Sigma)) \)) by developing a search tree. Once it has activated \( w_1 \), the master \( m \) wakes up in a sequential way the other workers \( w_2, \ldots, w_n \). Each time a worker \( w_j \) has just been waked up or has finished its current job, \( w_j \) becomes available again and is ready to work. \( w_j \) contacts the master \( m \) to let \( m \) know that \( w_j \) is available, and \( m \) turns the corresponding flag to "available". Whenever \( m \) knows that a worker \( w_j \) is available, \( m \) asks successively each busy worker \( w_j \) whether \( w_j \) needs some help or not. \( w_j \) may respond positively or not (e.g., when its current job is such that \( \gamma \) is large enough or \( V \) is small enough, \( w_i \) may decide to finish the job alone). The choice of accepting some help for finishing the job is made by comparing the number of variables that remain in the instance to the value of the parameter \( \text{nbVarSeq} \) of \( \text{DMC} \). If this number of variables is small enough (i.e., lower than \( \text{nbVarSeq} \)), then \( w_i \) finishes the job itself. This test is important to avoid ping-pong effects, i.e., the fact that the workers finally spend more time communicating one another (asking some help) instead of actually counting models. As soon as a worker \( w_i \) responds positively ("I need some help") to the master, \( m \) communicates to \( w_i \) the name \( j \) of \( w_j \) so that \( w_i \) can send a message to \( w_j \), and \( m \) switches the flag associated with \( w_j \). The message sent to \( w_j \) consists of a pair \((\gamma', V')\) such that \( \gamma' \) extends \( \gamma \) and \( V' \) is a subset of \( V \). Thus \( w_j \) starts the computation of \( ||\Sigma|| \) w.r.t. \((\gamma', V')\) so that \( w_j \) is busy again. Note that knowing \( \gamma' \) and a single variable occurring in \( V' \) is not enough to characterize \( V' \) entirely because of the learnt clauses which may differ among the workers \( w_i \) and \( w_j \).

At the end, when all the workers are available, the computation can stop: the master \( m \) broadcasts a message to every worker \( w_i \) to let \( w_i \) know that it can terminate and \( m \) finally computes and returns the number of models of \( \Sigma \), by evaluating in a bottom-up way the arithmetic tree corresponding to the set of pairs \( $k = expression \) collected so far.

**Job sharing.** Whenever a worker \( w_i \) is ready to get some help from another worker \( w_j \), \( w_i \) chooses an open node \( N \) in its own backtracking queue. By construction, \( N \) corresponds either to (1) a pending literal \( l \) over variable \( \nu \) (the remaining child of a decision node over \( \nu \) considered by \( w_i \) when solving the job \((\gamma, V)) \) or (2) a child of a decomposable \& node. \( w_i \) then associates with \( N \) the identifier \( wid(\gamma', V') \) where \( \gamma' \) is an extension of \( \gamma \) of truth values of the nodes encountered from the root of the search tree explored by \( w_i \) for solving \((\gamma, V)) \) to \( N \), and \( V' \) is the subset of variables occurring in the formula rooted at \( N \). In our implementation, the node \( N \) chosen by \( w_i \) is the first node which is not already associated with an identifier when the queue is parsed from its end (the first node pushed into the queue) to its head (which corresponds to the next node that will be explored by \( w_i \) if a backtracking occurs). The rationale for this choice is threefold: it makes the part of the search space to be explored by \( w_i \) as large as possible, thus avoiding \( w_i \) to turn back to the idle state too early; it prevents from generating a very large arithmetic tree; and it promotes component caching which, in practice, is typically much more effective at the bottom of the search tree (to be explored by \( w_i \)) than at its top.

Once it has sent the job \((\gamma', V')) \) to \( w_j \), worker \( w_i \) resumes its own job. When a backtracking actually takes place and an
open node associated with an identifier is considered, \( w_i \) can skip this node since \( w_i \) knows that the work corresponding to this part of the search space has been or is currently achieved by another worker. When its backtrack queue is finally empty, \( w_i \) has finished its job: \( w_i \) sends to the master a message consisting of the identifier \( \text{uid} (\gamma, V) \) of the job, and an arithmetic expression characterizing the number of models of the instance given by \( (\gamma, V) \). This expression is based on \( + \) (at each decision node one must add the numbers of models of the two children, once normalized), \( \times \) (at each decomposable node one must multiply the numbers of models of the children), and at each decision node \( N \) normalisation requires to multiply the number of models of each child by \( 2^c \) where \( c \) is the number of variables of \( N \) occurring only in the other child), non-negative numbers (numbers of models that have been computed by \( w_i \) itself) and identifiers (of the form \( \text{uid} \), naming expressions characterizing numbers of models which are computed by other workers to which \( w_i \) asked some help).

As a matter of illustration, Figure 1 presents DMC at work on a toy example, with three workers.

Figure 1: DMC at work on a toy example, with three workers.

\[
\begin{align*}
\text{Initial CNF formula } \Sigma &= (\neg a \vee b \vee \neg f) \land (\neg a \vee \neg b \vee \neg f) \land (\neg a \vee b \vee c \vee f) \land (\neg a \vee d \vee e \vee f) \\
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{Steps} & \text{Worker } w_1 & \text{Worker } w_2 & \text{Worker } w_3 & \text{Master} & \text{Worker } w_1 & \text{Worker } w_2 & \text{Worker } w_3 \\
\hline
1 & (\gamma, V) = (\neg a, (a, b, c, d, e, f)) & \text{Waiting} & \text{Waiting} & \text{Master} & \text{Need help ?} & \text{Yes} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
2 & (\gamma, V) = (\neg a, \neg b, c, d, e, f)) & \text{Learn the clause } \neg a \vee f & (\gamma, V) = (a, (b, c, d, e, f)) & \text{Master} & \text{Need help ?} & \text{No} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
3 & (\gamma, V) = (\neg a, b, c, d, e, f)) & \text{Job done } s_2 = 32 & (\gamma, V) = (a, (b, c)) & \text{Master} & \text{Need help ?} & \text{Yes} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
4 & \text{Job done } s_1 = s_2 + (s_3 \times 3) & (\gamma, V) = (a, (b, c)) & \text{Job done } s_3 = 3 & \text{Master} & \text{Need help ?} & \text{No} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
5 & \text{Job done } s_1 = s_2 + (s_3 \times 3) & (\gamma, V) = (a, (b, c)) & \text{Job done } s_3 = 3 & \text{Master} & \text{Need help ?} & \text{Yes} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
6 & \text{Evaluation } s_1 = s_2 + (s_3 \times 3) & (\gamma, V) = (a, (b, c)) & \text{Job done } s_3 = 3 & \text{Master} & \text{Need help ?} & \text{Yes} & \text{Listen } w_1 \\
& & & & & \text{Contact } w_2 & s_2 & \text{Listen } w_3 \\
\hline
\end{array}
\end{align*}
\]
nected component as the node associated with \( v \). But this is not the case: the clauses learnt by \( w_j \) are not necessarily the same ones as those learnt by \( w_i \) when the message \( (\gamma', V') \) is sent from \( w_i \) to \( w_j \) and this changes the picture a lot. Even if the preprocessing techniques used by \( w_i \) and \( w_j \) coincided, there would be no guarantee that \( V' \) corresponds to a connected component of \( \text{BCP}(\Sigma_j | \gamma') \) when \( V' \) is a connected component of \( \text{BCP}(\Sigma_j | \gamma') \). This is due to the fact that the notion of decomposability is syntactic: the connected components of two equivalent \( \text{CNF} \) formulae do not coincide in general. Therefore, the number of models of the \( \text{CNF} \) formula characterized by \( \Sigma_j, \gamma' \) and \( V' \) can differ from the number of models of the \( \text{CNF} \) formula characterized by \( \Sigma_i, \gamma' \) and \( V' \).

As a matter of illustration, let us consider a very simple example (see Figure 1, with \( w_i = w_1 \) and \( w_j = w_3 \), steps 3 to 5) Suppose that \( w_i \) has previously learnt a clause \( \neg a \lor \neg b \), which has not been learnt by \( w_j \); if the set of clauses of \( \Sigma_j | \gamma \) reduces to \( \neg a \lor b \lor \neg f, \neg a \lor \neg b \lor \neg f, \neg a \lor b \lor c \lor f, \neg a \lor d \lor e \lor f, \neg a \lor d \lor e \lor f \), and \( \neg a \lor \neg f \), and \( \gamma' \) is extended by setting \( a \) to \( b \), then \( \text{BCP}(\Sigma_j | \gamma') \) has two connected components: one corresponding to \( b \lor c \) and the other one corresponding to \( d \lor e \). Suppose that \( w_i \) helps the job of \( w_j \) for computing the number of models of \( b \lor c \). Then \( w_i \) sends to \( w_j \) the message \( (\gamma', V') \) with \( V' = \{ b, c \} \). But \( V' \) is not the set of variables of a connected component of the \( \text{CNF} \) formula \( \text{BCP}(\Sigma_j | \gamma') \) when \( \Sigma_j | \gamma \) reduces to \( \neg a \lor b \lor \neg f, \neg a \lor \neg b \lor \neg f, \neg a \lor b \lor c \lor f, \neg a \lor d \lor e \lor f, \neg a \lor d \lor e \lor f \). This explains why a preliminary step is necessary. This step consists in computing first a model \( \omega \) of \( \text{BCP}(\Sigma_j | \gamma') \); if there is no such a model, then the number of models corresponding to \( (\gamma', V') \) is 0 and \( w_j \) finished the job \( (\gamma', V') \); in the remaining case, \( \omega \) is projected onto the set of variables not occurring in \( V' \), giving rise to a partial assignment \( \gamma'' \) and then the model counter associated with \( w_j \) computes and returns the number of models of \( \text{BCP}(\Sigma_j | \gamma'') \). By construction, this number of models coincides to the number of models of the connected component of \( \text{BCP}(\Sigma_j | \gamma') \). Stepping back to the previous example, a model \( \omega \) of \( \Sigma_j | \gamma' = (b \lor \neg f) \land (\neg b \lor \neg f) \land (b \lor c \lor f) \land (d \lor e \lor f) \) is computed by \( w_j \); for instance \( \omega = \{ b, \neg e, d, \neg f \} \); then, the model counter associated with \( w_j \) computes and returns the number of models of \( \text{BCP}(\langle b \lor \neg f \rangle \land \langle \neg b \lor \neg f \rangle \land \langle b \lor c \lor f \rangle \land \langle d \lor e \lor f \rangle) \) / \( d \land e \land \neg f \), which is equal to the number of models of \( b \lor c \) as expected.

3.2 Comparison with \((d)\text{CountAntom}

\text{CountAntom}. Unlike \text{CountAntom} [Burchard et al., 2015], \text{DMC} is a "true" distributed model counter, in the sense that the processing units which are used for the computation of \(|\Sigma|\) are not necessarily restricted to the cores of the (unique) processor on which the computation takes place. This makes a very significant difference between the two counters. While the limitation of the number of processing units is in favor of \text{DMC}, the presence of a memory shared by the threads of the processor renders feasible for \text{CountAntom} to let the processing units share many information, especially the conflict clauses learnt and the cached sub-formulae encountered during the search. Obviously enough, the possibility to delegate some jobs to workers implemented on other computers restricts the applicability of component caching in \text{DMC}.

Each job \( (\gamma, V) \) done by a worker \( w_i \) (possibly with the help of other workers), every \( \text{CNF} \) sub-formula considered by \( w_i \) can be cached once \( w_i \) has succeeded in computing the corresponding number of models: if a formula put in the cache is encountered again by \( w_i \) during the achievement of the job \( (\gamma, V) \), then its number of models can be retrieved from the cache instead of re-computing it. However, since the cache used by \( w_i \) is stored in the memory of \( w_i \) and there is no memory shared between the workers, the workers helping \( w_i \) to do the job \( (\gamma, V) \) cannot take advantage of it.

\text{dCountAntom}. \text{dCountAntom} [Burchard et al., 2016] is a distributed model counter (the only one we are aware of). It extends \text{CountAntom} with an additional message passing layer allowing a master counter (which is unique) to delegate work to slave counters (clones of \text{CountAntom}) and permits those slave solvers to exchange information in order to guide the computation. The master counter can share a node (corresponding to a part of the search tree to be explored) with a slave only if the node is at a decision level of sufficient depth (its value is made precise by a parameter \( \delta \) of \text{dCountAntom}) but a slave solver \( w_i \) cannot give jobs to other slave solvers. Whenever a slave \( w_i \) cannot solve a node within a preset amount of time \( \tau \), \( w_i \) gives up the corresponding job and gives it back to the master which continues the computation. In order to break down such a hard node, its children can only be shared with the slave processes again after a predefined number of decision levels.

In our opinion, the distribution strategy followed by \text{dCountAntom}, which follows a cube-and-conquer approach, suffers from several weaknesses. First of all, each time a slave gives up a job, the corresponding computational effort is wasted. Furthermore, when the instance considered by the master is hard enough, the slave processes can be idle most of the time. Since slaves cannot delegate jobs to other slaves (which would make sense when trying to solve difficult sub-formulae), if the number of cores of the processor on which a \text{CountAntom} slave is run is not enough to get a result before the fixed time limit, almost all the work will finally be done by the master itself, leading to a significant workload imbalance. This is avoided by \text{DMC} since the distribution policy used in it promotes the concentration of the computational efforts on the hardest parts of the search space. Indeed, the distribution policy followed by \text{DMC} is close to work stealing, a well-known scheduling strategy for parallel computations, especially suited for multithreaded computer programs (see, for instance, [Blumofe and Leiserson, 1999]). The nodes \( N \) corresponding to the unexplored parts of the current search tree of a worker correspond to its work items. As in work stealing, the exploration of the search tree rooted at \( N \) may spawn new work items that can feasibly be executed in parallel with its other work (these new items correspond to decision nodes which are descendants of \( N \)). However, when a processor \( w_j \) runs out of work, it does not look at the queues of other processors \( w_i \) (again, there is no memory shared by the processing units) for "stealing" work items, but instead \( w_j \) sends a request to the master. Then another worker \( w_i \), which is currently busy will eventually accept this help and give a work item to \( w_j \).
In **dCountAntom**, conflict clauses are shared as soon as they are learnt (whenever the master learns a clause or receives such a clause from a slave, it transmits it to all slaves). Furthermore, every sub-formula corresponding to a sub-problem to be solved is transmitted by the master to a slave. This leads to many communications passing through the master, including communications involving messages of large size (those corresponding to some sub-formulae to be solved). Contrastingly, the distribution policy used by **DMC** is more parsimonious as to the messages transmitted. The clauses learnt are not exchanged by the workers and the sub-formulae corresponding to unexplored parts of the search space are not explicitly communicated. In more detail, the sizes of the messages transmitted from the workers to the master for asking some help when available or accepting/refusing to be helped, are bounded by the size of the identifier of the worker so they are very short. The sizes of the messages \((\gamma', V')\) transmitted from a worker to another worker which is ready to help are upper bounded by the number of variables in \(\Sigma\). The sizes of the arithmetic expressions transmitted from any worker \(w_i\) to the master when solving \((\gamma, V)\) is at most linear in the size of the longest branch of the search tree developed by \(w_i\) to achieve this job.

Finally, the performance of **dCountAntom** looks heavily dependent on the choices of the parameter \(\delta\) and of the time-out \(\tau\) set up for the slaves. In **DMC**, the only parameter used is nbVarSeq. No parameters like \(\delta\) and \(\tau\) are considered, thus the issue of tuning the values of these parameters is also avoided.

## 4 Experiments

### Experimental Setting.

In order to evaluate and compare **DMC** with the model counters **D4**, **CountAntom**, and **dCountAntom**, we have considered 50 CNF instances gathered into 9 data sets, as follows: BN (Bayesian networks) (11), BMC (Bounded Model Checking) (3), Configuration (5), Planning (12), Random (2), Qif (2) (Quantitative Information Flow analysis - security), Circuit (3), Fault Injection (6), Output Probability (6). All the instances from the first 7 families come from the SAT LIBrary www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html. They have been selected as hard enough for not being solved with a time limit of 3600s by at least one of the counters considered in the experiments reported in http://www.cril.univ-artois.fr/KC/documents/d4Results.pdf, namely **D4**, **C2D**, **Dsharp**, **Cachet**, **sharpSAT**. Especially, some of those instances have not been solved by **D4**, **C2D**, or **Dsharp**, **Cachet**, **sharpSAT**. Basically, the rationale of the selection process was to avoid trivial instances, while keeping sufficiently many benchmarks for which the computations of **D4** terminated (this was useful to evaluate the improvement of **DMC** over **D4**). The instances of the last two families include those considered in the experiments reported in [Burchard et al., 2015; 2016] and are available from https://projects.informatik.uni-freiburg.de/projects/countantom/files.

The experiments have been conducted on a cluster of sixteen 32 GiB RAM computers containing each two quad-core bi-processors Intel XEON X5500 at 2.67 GHz. The cluster is equipped with an Ethernet controller at 1 GiB/s. A time-out (wall clock time) of 3600s has been considered for each instance. In the experiments, every worker used within **DMC** is an avatar of **D4**, which takes advantage of the same preprocessing combination, namely the computation of the backbone of the input [Monasson et al., 1999], followed by an occurrence reduction step [Lagniez and Marquis, 2017]. A worker does not ask any help whenever the number of variables \(V\) of its current job \((\gamma, V)\) does not exceed the value of the parameter nbVarSeq of **DMC**, fixed to 30 in our experiments. In our implementation of **DMC**, the communications between processes are managed using the library Open-MPI for message passing interface (MPI). MPI is hardware independent and it supports many communication techniques (point-to-point, collective, shared-memory, Ethernet, etc.).

A first round of experiments has consisted in letting the number of cores to vary when running **DMC** on the set of benchmarks, in order to evaluate the benefits obtained in terms of computational time, as well as the corresponding speedups. Here, for any instance \(\Sigma\) solved in \(t_n\) seconds by **DMC** running on \(n\) cores, the speedup is given by the ratio \(\frac{\min(t_1,3600)}{t_n}\), where \(t_1\) is the number of seconds used by **DMC** running on one core only (so \(t_1\) is undefined if \(\Sigma\) has not been solved within 3600s by **DMC** running on one core). Thus, the measured speedups are in fact lower bounds of the actual speedups that would be computed by letting **DMC** on one core to run up to exhaustion if this was feasible. The number \(n\) of cores used for the workers varied from 1 up to 128 following a geometric progression with common ratio 2. The case when a single worker is considered mainly amounts to comparing **DMC** with the avatar of **D4** considered in the experiments. It turns out that the performance of **DMC** running on one core is really close to the one of the corresponding avatar of **D4**. This can be easily explained by the architecture used (when one core is considered, only, the communication time is negligible, the unique worker does all the job). A second round of experiments has consisted in running **CountAntom** on one computer of the cluster (thus exploiting 8 cores) and **dCountAntom** on the whole cluster (using 128 cores).

### Experimental Results.

The results we have obtained are synthesized on Figure 2.

Figure 2 (a) presents a cactus plot which indicates for a given amount of (wall clock) time (reported on the y-axis) the number of instances solved reported (on the x-axis). This is done for all the counters considered in the experiments, i.e., **DMC** running on 1, 2, 4, 8, 16, 32, 64, or 128 cores, **CountAntom** running on 8 cores and **dCountAntom** running on 128 cores. As to **DMC**, it can be observed that adding some cores leads as expected to reduce the time needed to solve the instances, thus to solve more instances within a given limit. **DMC** running on 8 cores has been able to solve 44 instances, while **CountAntom** (running on 8 cores as well) solved only 26 instances. Interestingly, the instances solved by the two counters differ: 2 instances solved by **CountAntom** have not been solved by **DMC**, and 14 instances solved by **DMC** have not been solved by
The importance of the caching ingredient and of the learnt clauses in the solving process may explain the discrepancy. dCountAntom running on 128 cores has been able to solve only 16 instances in due time. For 10 instances, the program did not terminate properly but with a segmentation fault. So even if dCountAntom had succeeded in solving them, it would have solved less instances than CountAntom running on 8 cores. Such disappointing results can be explained by the significance of the values of the parameters $\delta$ and $\tau$ chosen for the computations (their default values may be not suited to the instances we have considered). As explained in [Burchard et al., 2016], the performance of dCountAntom turns out to be very sensitive to the values of $\delta$ and $\tau$.

Figure 2 (b) focuses on DMC and reports on the $y$-axis some box-and-whisker plots synthesizing the amount of (wall clock) time spent to solve the instances depending on the number of cores used, which is reported on the $x$-axis. The bottom and top of each box are the first and third quartiles, and the band inside the box is the second quartile (the median). The ends of the whiskers represent the minimum and maximum of all of the data. In many cases, this figure shows significant improvements in the time spent by the workers in communication tasks or being idle, over the total time used for solving the instance. Empirically, it turns out that these ratios are typically small, especially for the hardest instances, even when the number of cores is high. For instance, for the hardest instance among those considered in our experiments, namely comm-p10-p-t10 from the planning family, with 17539 variables and 75516 clauses, which has been solved by DMC on 128 cores in 2872.58s (and not solved by any of the other model counters we considered), the ratio is equal to 0.81% only. This shows that our objective of keeping small enough the communication time and the time during which the workers are idle has been reached to a significant extent.

## 5 Conclusion

We have presented DMC, a distributed model counter for CNF formulae, based on the sequential model counter D4. Unlike the parallel counter CountAntom, DMC can take advantage of a (possibly large) number of sequential model counters running on (possibly heterogeneous) computing units spread over a network of computers. In DMC, the workload distribution follows a policy close to work stealing and the number and the sizes of the messages which are exchanged by the jobs are kept small. As such, DMC differs significantly from the distributed counter dCountAntom which is based on a cube-and-conquer approach and uses a much more demanding communication strategy. The empirical results we have obtained show DMC as a much more efficient counter than D4, the distribution of the computation leading to time savings of several orders of magnitude and yielding large improvements for some benchmarks. Experimentally, DMC appears also as a serious challenger to CountAntom and to dCountAntom.
Indeed, while the number of processing units used in our experiments was quite restricted, our empirical evaluation has shown that the ability to distribute the model counting task thanks to DMC permits to solve in a reasonable amount of time instances that were out of reach before.

As a next step, it would be interesting to take advantage of the best of CountAntom within DMC, i.e., to exploit the shared memory between workers running on the same computer. The problem of determining the amount of information to be shared by “co-located” workers for optimizing the performance of the computation does not look so easy. This is an interesting perspective for further research.

Acknowledgments

This work has been partly supported by the CPER DATA project funded by the Région Hauts-de-France. We are also grateful to the reviewers for their useful comments.

References


