Parallel Theorem Proving

Maria Paola Bonacina

Abstract This chapter surveys the research in parallel or distributed strategies for mechanical theorem proving in first-order logic, and explores some of its connections with the research in the parallelization of decision procedures for satisfiability in propositional logic (SAT). We clarify the key role played by the *Clause-Diffusion methodology for distributed deduction* in moving parallel reasoning from the *parallelization of inferences* to the *parallelization of search*, which is the dominating paradigm today. Since the quest for parallel first-order proof procedures has not been pursued recently, we endeavor to relate lessons learned from investigations of parallel theorem proving and parallel SAT-solving to novel advances in theorem proving, such as SGGS (*Semantically-Guided Goal-Sensitive reasoning*), a method that lifts the CDCL (Conflict-Driven Clause Learning) procedure for SAT to first-order logic.

1 Introduction

Research on parallel theorem proving, meaning automated theorem proving (ATP) in first-order logic, began in the mid and late 1980s, flourished in the 1990s, and came pretty much to a halt in the early 2000s [190, 49, 206, 84, 36]. Research on parallel satisfiability solving, meaning satisfiability in propositional logic (SAT), began in the early 1990s and is still actively pursued today (see [161, 113, 1, 156], Chapter 1 on Parallel Satisfiability and Chapter 2 on Cube and Conquer). It is probably unknown to most authors active in parallel SAT-solving that Hantao Zhang began work on his parallel SAT solver PSATO [223, 224], which pionereed the divide-and-conquer approach to parallel SAT-solving, after learning about the *Clause-Diffusion methodology for distributed deduction* [26, 47, 48, 50] and its implementation in the AQUARIUS theorem prover [45, 26, 46, 51].

Maria Paola Bonacina
Dipartimento di Informatica, Università degli Studi di Verona, Strada Le Grazie 15, I-37134 Verona, Italy, EU. e-mail: mariapaola.bonacina@univr.it

In previous work, we surveyed parallel theorem proving twice, first at the height of the interest in this topic [26, 49], and then when the involvement of the scientific community with this fascinating subject was already decreasing [35, 36]. In this chapter we revisit parallel theorem proving, in the light of advances in theorem proving itself and in parallel SAT-solving, with the aim of providing the reader with material to reflect about the connections between:

- Past work in parallel theorem proving and selected contemporary approaches to theorem proving;
- Past work in parallel theorem proving and later work in parallel SAT-solving;
- Selected approaches to parallel SAT-solving and potentially new leads for parallel theorem proving.

Since the time of the latest investigations in parallel theorem proving, the field has witnessed a significant growth of the paradigm of *model-based reasoning* [43], where a theorem-proving method is *model-based* if the state of a derivation contains a representation of a candidate partial model that unfolds with the derivation. Traditional model-based first-order methods include *subgoal-reduction strategies* based on *model elimination* and *model-elimination tableaux*, whose parallelization received considerable attention [192, 6, 74, 62, 142, 169, 205, 103, 215]. It is therefore an interesting question to ask what we may learn from past work on parallel theorem proving towards the parallelization of more recent or new approaches to model-based theorem proving.

The growth of model-based reasoning has various motivations. One motivation is the relevance of models for applications. For instance, an assignment to program variables is a model from a logical point of view [39]. Thus, models become "moles" to exercise program paths in testing or examples for example-driven synthesis, and a reasoner that generates models supports automated test generation and program synthesis [81, 139]. Another motivation is inspiration from the practical successes of solvers for propositional satisfiability (see [155] and Chapter 1 on Parallel Satisfiability) and satisfiability modulo theories (SMT) (see [82] and Chapter 5 on Parallel Satisfiability Modulo Theories), which are model-based because they are built on the CDCL (Conflict-Driven Clause Learning) procedure [159, 160, 170, 158], which is inherently model-based. Therefore, another spontaneous question is what we may learn from past work on both parallel theorem proving and parallel SAT-solving towards the parallelization of recent model-based first-order methods. Examples of the latter include DPLL($\Gamma+\mathscr{T}$) [80, 55, 56], which integrates an *ordering-based strategy* in an SMT solver, and SGGS (*Semantically-Guided Goal-Sensitive reasoning*) [59, 58, 60, 61], which generalizes CDCL to first-order logic.

To this end we reconsider and expand our analyses of the parallelization of theorem proving [49, 36], covering *subgoal-reduction strategies*, *ordering-based strategies*, and *instance-based strategies*. Ordering-based strategies are based on *ordered resolution* and *ordered paramodulation/superposition*. They represent the state of the art for first-order logic with equality and are implemented in top-notch theorem provers such as PROVER9 [168], SPASS [214], E [188], and Vampire [135]. Although they are not model-based, their connection with SAT or SMT solvers is

a current research topic [42, 56, 180]. Instance-based strategies integrate instance generation at the first-order level with deciding satisfiability at the propositional or ground level, by a SAT or SMT solver [123, 132]. They are model-based, or at least model-driven, in as much as instance generation is geared to exclude the models proposed by the solver.

We illustrate a selection of methods already covered in our previous surveys [49, 36], to make this chapter self-contained, give the reader a direct impression of those methods, and have material for discussion. For example, ROO [152, 153, 154] illustrates the approach to parallelization by parallel inferences in shared memory that was the state of the art before Clause-Diffusion, and Team-Work [83, 8, 87, 9, 88, 91, 90, 86, 195] is a forerunner of the *portfolio approach*.

Then we summarize twelve years of research on the *Clause-Diffusion methodology* [26, 47, 48, 50] to parallelize ordering-based strategies, including *Modified Clause-Diffusion* [27, 28], and the Clause-Diffusion provers AQUARIUS [45, 26, 46, 51], PEERS [57, 50], and PEERS-MCD [29, 30, 32, 37]. We reflect on the impact of Clause-Diffusion on subsequent research: Clause-Diffusion played a central role, because it was the first approach to move from the parallelization of inferences to the parallelization of search, so it can be considered a forerunner of all parallel or distributed search methods in both theorem proving and satisfiability.

In the final discussion we draw connections between parallel theorem proving and parallel SAT-solving (e.g., [106, 111, 112, 122, 185, 110, 115]), and we discuss ideas for future work in parallel theorem proving, especially parallel model-based theorem proving.

This chapter is organized in three parts: Section 2 provides a parallelization-oriented survey of theorem-proving strategies; Section 3 presents the approaches to parallel theorem proving; and Section 4 contains the final discussion.

It is our contention that much of the research in parallel and distributed theorem proving was simply ahead of its time, with respect to both theorem proving and parallel or distributed computing, and we hope that this chapter will contribute to maintain its intellectual legacy alive and fruitful for future research.

2 Theorem-Proving Strategies

We begin our analysis of the parallelizability of theorem proving with a classification of theorem-proving strategies in three categories: *subgoal-reduction-based*, *ordering-based*, and *instance-based* strategies. Ordering-based strategies include *expansion-oriented* and *contraction-based* strategies. In this section we survey these classes of theorem-proving strategies, covering inference system, search plan, proof generation, redundancy control, and use of models. The presence of *backward contraction* inferences, the *size* of the database of clauses generated and kept by the strategy, and the degree of *dynamicity* of the database of clauses, are among the relevant issues for parallelization.

2.1 Subgoal-Reduction Strategies

We use φ and ψ for clauses, σ for most general unifiers, L and L' for literals, C and D for disjunctions of literals, $\simeq$ for equality, and $\Box$ for the empty clause, which represents a contradiction.

In theorem proving *subgoal reduction* stems from *ordered linear resolution* [136]: at each step the strategy resolves the current goal clause $\varphi_i = L \vee C$ with an input clause $\psi = L' \vee D$, such that $L\sigma = \neg L'\sigma$. The next goal clause φ_{i+1} is the resolvent $(D \vee C)\sigma$, where L, seen as a *subgoal* to be solved, has been replaced by or *reduced to* a new bunch of subgoals D. Such a procedure is called *linear*, because at every resolution step one of the parents is the previous resolvent; it is called *linear input*, if, in addition, the side clause ψ is always an input clause. The *ordered* attribute refers to the requirement that the literals in goal clauses be resolved away in a fixed predefined order, determined by the *literal-selection rule* of the strategy, also known as the AND-*rule*. A typical example is to select literals in left-to-right order. Using only input clauses as side clauses is sufficient for problems made of Horn clauses, that is, clauses with at most one positive literal [118]. For full first-order logic, also ancestor goal clauses have to be considered for side clauses, so that ψ may be a φ_j with $j < i$.

Model elimination (ME) can be described as a variant of ordered linear resolution [151, 70, 53], where L is saved in φ_{i+1} as a *boxed*, or *framed*, literal $[L]$ (*A-literal* in the original ME terminology), so that φ_{i+1} has the form $(D \vee [L] \vee C)\sigma$. The resulting inference rule is called *ME-extension*. In this manner, resolution with an ancestor goal clause can be replaced by *ME-reduction*, which reduces a goal clause $L' \vee D \vee [L] \vee C$ to $(D \vee [L] \vee C)\sigma$ when $L\sigma = \neg L'\sigma$. Thus, ME is a linear input strategy for full first-order logic.

Independent of resolution, the concept of ME is to prove that the input set S of clauses is unsatisfiable by eliminating all possible candidate models [150]. In order to satisfy a set S of first-order clauses, it is necessary to satisfy all its clauses. In order to satisfy a clause, it is necessary to satisfy all its ground instances. In order to satisfy a ground instance, it is necessary to satisfy one of its literals. If the current goal clause $\varphi_i = L \vee C$ and an input clause $\psi = L' \vee D$ are such that $L\sigma = \neg L'\sigma$, no model can satisfy $\varphi_i\sigma$ and $\psi\sigma$ by satisfying $L\sigma$ and $L'\sigma$. The next goal clause $\varphi_{i+1} = (D \vee [L] \vee C)\sigma$ generated by ME-extension says precisely this: the literal $L\sigma$ is framed to denote that it has been added to the current candidate model, so that $\varphi_i\sigma$ is satisfied; since a model that satisfies $L\sigma$ cannot satisfy $L'\sigma$, some other literal in $D\sigma$ must be satisfied to satisfy $\psi\sigma$. In this sense, the literals of $D\sigma$ are *subgoals* of $L\sigma$. An ME-reduction step that reduces a goal clause $L' \vee D \vee [L] \vee C$ to $(D \vee [L] \vee C)\sigma$, when $L\sigma = \neg L'\sigma$, reckons that $L'\sigma$ cannot be satisfied in a model that contains L.

ME-tableaux make this model elimination process perspicuous by building a tableau, that is a tree, whose nodes are labeled by literals, and whose branches represent candidate models [142, 18, 24, 19, 140, 144, 38]. A branch is *closed* if it contains two complementary literals, and it is *open* otherwise. An open branch represents a candidate model, whereas a closed branch represents an eliminated model.

A tableau is *closed* if all its branches are, which means that all candidate models have been eliminated. If $L\sigma = \neg L'\sigma$ for the leaf L of an open branch and an input clause $\psi = L' \vee D$, ME-extension extends the branch with children labeled by the literals of ψ, applies σ to the tableau, and closes the branch with the complementary literals $L\sigma$ and $L'\sigma$. If a branch contains literals L and L' such that $L\sigma = \neg L'\sigma$, ME-reduction applies σ to the tableau and closes the branch.

A *subgoal-reduction derivation* can be described in the form

$$(S;\varphi_0) \vdash (S;\varphi_1) \vdash \ldots (S;\varphi_i) \vdash \ldots$$

where S is the input set of clauses, $\varphi_0 \in S$ is the input clause designated as *initial goal*, and $\varphi_1,\ldots,\varphi_i,\ldots$ is the sequence of succeeding goal clauses. The initial goal clause $\varphi_0 = L_1 \vee \ldots \vee L_k$ yields an *initial tableau* where the root has k children labeled by $L_1,\ldots,L_k$. Note that the literals $L_1,\ldots,L_k$ may share variables, which means that the branches of the tableau may share variables, which is why substitutions apply to the entire tableau. The literals in the current goal clause φ_i label the leaves of the open branches of the tableau, and the framed literals in φ_i label the inner nodes of the tableau.

A derivation is a *refutation* if $\varphi_k = \square$ for some k, $k > 0$, or, equivalently, if the tableau is closed. A subgoal-reduction strategy is *refutationally complete* if, whenever S is unsatisfiable and $S \setminus \{\varphi_0\}$ is satisfiable, there exists a refutation of S by the strategy starting with $(S;\varphi_0)$. If S is unsatisfiable and $S \setminus \{\varphi_0\}$ is satisfiable, it is the addition of φ_0 that makes the set unsatisfiable, and this is why φ_0 is the *initial goal clause*. Since all generated clauses descend from the initial goal clause, subgoal-reduction strategies are *goal-sensitive*. An unsatisfiable S may contain more than one clause φ_0 such that S is unsatisfiable and $S \setminus \{\varphi_0\}$ is satisfiable, so that there may be a choice of initial goal clause.

Given a refutation with $\varphi_k = \square$, the comb-shaped resolution tree formed by the sequence of goal clauses $\varphi_0,\ldots,\varphi_{k-1},\square$ and their companion side clauses is the generated *proof*. The linear shape of the generated proof reveals the linear nature of the strategy. In ME-tableaux, the closed tableau represents the proof.

In subgoal-reduction strategies, *redundancy* appears in the form of repeated subgoals or subgoal instances, and it is countered by techniques called *C-reduction* [193], *caching* [7, 53], *regressive merging* [211], *folding-up* [141, 102] or *success substitutions* [144], and *tabling* or *memoing* [213]. C-reduction, caching, and regressive merging are used in model elimination, folding-up and success substitutions in tableaux, tabling and memoing in declarative programming. In essence, all these techniques descend from the idea of *lemmatization* or *lemmaizing* [150, 7, 53]. Adopting tableaux parlance, if the strategy manages to close a sub-tableau whose root is labeled with literal L, it means that no model of the set S of clauses contains L, that is, $S \models \neg L$. Thus, $\neg L$ can be learned as a lemma, and applied to resolve away, or close, any future subgoal L' such that $L\sigma = L'\sigma$ [141, 102, 38]. [1] Lemmatization

[1] If the sub-tableau is closed using *ME*-reductions with ancestors $L_1,\ldots,L_n$ of L, no model with $L_1,\ldots,L_n$ contains L; the lemma is $\neg L \vee \neg L_1 \vee \ldots \vee \neg L_n$; and $\neg L$ can be applied as a unit lemma only below $L_1,\ldots,L_n$ (cf. Section 2.5 of [38]).

causes the database of clauses to grow, if the lemmas are added to S, but a common characteristic of caching techniques is to store the information on the learned lemma $\neg L$, or dually, on the solved subgoal L, without bothering S. For example, in folding-up, the information is stored in the tableau, at the node labeled with L.

Subgoal-reduction strategies use *depth-first search* (DFS) to search for a proof, with *backtracking* to get out of the deadend represented by a φ_i to which no inference applies (e.g., its leftmost literal can be neither ME-extended nor ME-reduced). A specific DFS plan is characterized by a literal-selection rule and a *clause-selection rule*, also known as the OR-*rule*, that determines the order in which the input clauses are tried. A typical example is to try clauses in top-down order, that is, in the order they are written in the input file. Backtracking undoes the latest inference and substitution application to enable the strategy to try a different inference. For completeness, DFS is enriched with *iterative deepening* [131, 202] on the number of resolution or ME-extension inferences. Thus, subgoal-reduction strategies develop and keep in memory *one proof attempt at a time*, and switch to another one by backtracking.

Prolog Technology Theorem Proving (PTTP) is a major paradigm for subgoal-reduction strategies [198, 199, 201]. PTTP implements ME on top of the *Warren Abstract Machine* (WAM), the virtual machine designed for Prolog [212]. The linear input nature of ME is crucial, because Prolog uses a variant of ordered linear input resolution. The WAM is a stack machine, with goal clauses stored on the stack, and input clauses compiled as a Prolog program. A stack machine implements DFS naturally. For theorem proving, PTTP adds iterative deepening and the *occur check* in unification: when computing a most general unifier σ, the substitution σ cannot include a pair $x \leftarrow t$, where x occurs in t. Unification in Prolog omits this check for the sake of performance and because Prolog, at least in its basic formulation, is a relational language with a limited use of function symbols, so that the likelihood of a pair $x \leftarrow t$, where x occurs in t, is deemed low.

2.2 Ordering-Based Strategies

Ordering-based strategies have two kinds of inference rules. *Expansion inference rules* generate and add clauses:

$$\frac{S}{S'} \quad S \subset S'$$

where $S \subset S'$ says that the existing set S of clauses is being expanded by adding something. An expansion inference rule is *sound* if whatever is added is a logical consequence of what pre-existed, that is, if $S' \subseteq Th(S)$, where $Th(S) = \{\varphi : S \models \varphi\}$. Examples include *binary resolution* and *factoring* [183], which add a *binary resolvent* or a *factor*, *paramodulation* [181, 173], which adds a *paramodulant*, *superposition* [130, 119, 11], which is ordered paramodulation where the literal paramod-

ulated into is an equality, *equational factoring* [12], and *reflection*, which is resolution with $x \simeq x$. Together these inference rules build equality into resolution [120, 184, 12, 52, 172].

Contraction inference rules delete clauses or replace them with smaller ones:

$$\frac{S}{\overline{\overline{S'}}} \quad S \not\subseteq S' \quad S' \prec_{mul} S$$

where $S \not\subseteq S'$ says that something has been deleted; $S' \prec_{mul} S$ says that S' is smaller than S in the *multiset extension* [95] of a *well-founded* ordering $\prec$ on terms, literals, and clauses [92]; and the double inference line [40] emphasizes the diversity of contraction with respect to the traditional notion of inference. Soundness for contraction is called *adequacy* [40]: a contraction inference rule is *adequate*, if whatever is deleted is a logical consequence of what is kept, that is, if $S \subseteq Th(S')$. Since $S \subseteq Th(S')$ implies $Th(S) \subseteq Th(S')$, soundness for contraction is also called *monotonicity* [34], meaning monotonicity of inferences with respect to theoremhood. Examples of contraction inference rules include *tautology deletion*, *subsumption* [183, 187], *clausal simplification*, which is a combination of unit resolution and subsumption, *demodulation* [218] or *simplification* (i.e., simplification by an equation) [130, 119, 11], *functional subsumption* (i.e., subsumption between equations) [119], *purity deletion* [79, 53], and *conditional simplification* [41]. Repeated simplification is called *normalization* or reduction to *normal form*, meaning a form that cannot be rewritten further, and normalization can be viewed as a single contraction step. The normal form of a clause φ is denoted $\varphi\downarrow$.

These strategies are called *ordering-based*, because they use the ordering $\prec$ to define contraction rules and restrict expansion rules: resolution is *ordered resolution*, factoring is *ordered factoring*, paramodulation is *ordered paramodulation*, and superposition is natively ordering-restricted. This means that only maximal literals are resolved upon, factorized, paramodulated into and from, or superposed into and from, and only maximal sides of equations are paramodulated or superposed into and from, where maximality is tested in the clause instantiated by the most general unifier of the inference step [119, 11, 120, 184, 12, 52, 172]. Ordering-based strategies search for a proof by *best-first search* and do not need backtracking. Best-first search is implemented by the *given-clause algorithm* which we shall cover in Section 3.2 because it is relevant to parallelization approaches. Several presentations, surveys, and systematizations of ordering-based theorem proving and its orderings are available [175, 93, 176, 97, 96, 14, 171, 40, 147, 177].

Ordering-based strategies are *not model-based*: in ordering-based inference systems models remain implicit, and come to the forefront only in the proofs of refutational completeness. For example, a proof technique uses *transfinite semantic trees* to survey models and show that the inference system excludes them all [120]. Another proof technique is based on *saturation*. A set of clauses is *saturated* if any inference with premises in the set is redundant (see Section 2.2.2 for redundancy). Refutational completeness is established by showing that a saturated set of clauses that does not contain $\square$ is satisfiable [12].

However, ordering-based strategies may use a *fixed* interpretation for *semantic guidance*, as exemplified in *semantic resolution* [196], *hyperresolution* [182], and *resolution with set of support* [217] (cf. Sections 2.6 in [34] and 2.1 in [43]).

Hyperresolution resolves a clause $L_1 \vee \ldots \vee L_q \vee C$, called the *nucleus*, and clauses $L'_1 \vee D_1, \ldots, L'_q \vee D_q$, with $q \geq 1$, called *electrons*, such that $L_i\sigma = \neg L'_i\sigma$ for all i, $1 \leq i \leq q$, to generate the *hyperresolvent* $(D_1 \vee \ldots \vee D_q \vee C)\sigma$ [182]. *Positive hyperresolution* assumes a fixed Herbrand interpretation I that contains all negative literals, and generates only clauses that are *false* in I, namely clauses whose literals are all positive. Such a clause is called *positive*. The electrons are required to be positive clauses, and $L_1, \ldots, L_q$ are required to be all and the only negative literals in the nucleus. Thus, positive electrons are used to resolve away all negative literals in the nucleus to get a positive hyperresolvent. *Negative hyperresolution* is defined dually with all signs exchanged.

In a *strategy with set of support*, all clauses issued from the negation of the conjecture are considered *goal clauses*. The input set S of clauses is subdivided into the set of support *SOS*, which contains the goal clauses, and its complement $T = S \setminus SOS$. T is assumed to be satisfiable (e.g., it contains the axioms of a theory), so that if S is unsatisfiable, the unsatisfiability is caused by *SOS*. Every expansion inference is required to be *supported*, meaning that at least one parent is in *SOS*. The generated clauses are added to *SOS*, and since they all descend from goal clauses, the strategy is *goal-sensitive*. A strategy with set of support is compatible with contraction (tautology deletion, subsumption, clausal simplification), provided also clauses generated by backward contraction are inserted in *SOS* [53]. Since T does not get expanded, a strategy with set of support is complete for problems with equality only if T is saturated. The interplay of *parallelism and semantic guidance* in ordering-based strategies has not been explored thus far, as we shall discuss in Section 4.

2.2.1 Expansion-Oriented Strategies

The distinction between *expansion-oriented* and *contraction-based* strategies towards analyzing parallelism [49] depends on the distinction between *forward* and *backward* contraction. In *forward contraction*, a newly generated clause φ, called a *raw clause* [26, 49], is deleted or normalized into $\varphi\downarrow$ by previously existing clauses. In *backward contraction*, such a $\varphi\downarrow$ is applied to contract previously existing clauses. Expansion-oriented strategies apply at most forward contraction. Accordingly, an *expansion-oriented derivation* has the form

$$(S_0; N_0) \vdash (S_1; N_1) \vdash \ldots (S_i; N_i) \vdash \ldots$$

where S_i is the set of clauses in the database of clauses, and N_i is the set of *raw clauses*. Every clause in S_i has an *identifier*, typically a positive integer generated progressively by the prover, and is ready to be used as premise. Raw clauses are clauses that were just generated and still need to undergo forward contraction. Ini-

tially, $S_0 = S$ is the input set of clauses and $N_0 = \emptyset$. Expansion takes premises in S_i and adds raw clauses to N_{i+1}. Forward contraction deletes clauses in N_i and adds their non-trivial normal forms to S_{i+1}. It follows that $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$, that is, for an expansion-oriented strategy the database of clauses is *monotonically increasing*.

A derivation is a *refutation* if $\Box \in S_k$ for some k, $k > 0$. A strategy is *refutationally complete* if, whenever S is unsatisfiable, there exists a refutation of S by the strategy. Ordering-based strategies develop *multiple proof attempts* that remain *implicit* in the set of clauses S_i. Only when $\Box \in S_k$, the strategy reconstructs the generated *proof* in the form of the *ancestor-tree* of $\Box$ [28], denoted $\Pi(\Box)$. The reconstruction starts from $\Box$ and proceeds backward until it reaches the input clauses. For instance, if φ is a resolvent of φ_1 and φ_2, $\Pi(\varphi)$ has root labeled φ and subtrees $\Pi(\varphi_1)$ and $\Pi(\varphi_2)$. If φ is generated as the normal form of a pre-existing clause ψ with respect to equations $\varphi_1, \ldots, \varphi_n$, $\Pi(\varphi)$ has root labeled φ and subtrees $\Pi(\varphi_1), \ldots, \Pi(\varphi_n)$, and $\Pi(\psi)$. Every clause has its own variables, and *variants*, that is, clauses that are identical up to variable renaming, are treated as distinct clauses. Therefore, no clause is generated twice, and $\Pi(\varphi)$ is uniquely defined given φ. Since a clause may be used as a premise more than once, $\Pi(\varphi)$ is an ancestor-tree if we allow the same clause to label more than one node, an *ancestor-graph* otherwise.

At the time of our first analysis of parallel theorem proving [26, 49], it was already understood that backward contraction is indispensable for theorem proving by ordering-based strategies, especially in the presence of equality. Thus, the class of expansion-oriented strategies was introduced to cover parallel resolution-based theorem-proving methods without backward contraction, mostly for first-order logic without equality [148, 149, 71, 75, 125] or propositional logic [99], and parallelizations [194, 210, 114, 67, 68] of the *Buchberger algorithm* [63] to compute Gröbner bases of ideals generated by sets of polynomials. The Buchberger algorithm is a *completion procedure* like *Knuth-Bendix completion* [130], with an expansion inference rule similar to superposition and a contraction rule similar to simplification [64]. However, the Buchberger algorithm is guaranteed to converge with or without backward contraction, which can be delayed to a post-processing phase. On the other hand, validity in equational theories, first-order logic, and first-order logic with equality is a semi-decidable problem, so that theorem-proving methods are only *semi-decision procedures*, and backward contraction is crucial in practice to find a proof and terminate. Since expansion-oriented theorem-proving strategies today have mostly pedagogical and historical interest, and the Buchberger algorithm is not a first-order theorem-proving strategy, we refer the reader interested in their parallelization to our previous surveys [49, 36].

2.2.2 Contraction-Based Strategies

Contraction-based strategies apply both forward and backward contraction eagerly and as much as possible. A *contraction-based derivation* has the form

$$(S_0;N_0;R_0) \vdash (S_1;N_1;R_1) \vdash \ldots (S_i;N_i;R_i) \vdash \ldots$$

where S_i is the set of clauses in the database of clauses, those with an identifier and ready to be used as premises; N_i is the set of *raw clauses*, that is, clauses just generated and still to be subject to forward contraction; and R_i is the set of clauses deleted by backward contraction. Initially, $S_0 = S$ is the input set of clauses and $N_0 = R_0 = \emptyset$. Expansion takes premises in S_i and adds raw clauses to N_{i+1}. Forward contraction deletes clauses in N_i and adds their non-trivial normal forms to S_{i+1}. Backward contraction detects which clauses in S_i can be contracted, moves them to N_{i+1}, and also copies them to R_{i+1}. In this way, backward and forward contraction are implemented by the same operations, and clauses generated by backward contraction are treated in the same way as clauses generated by expansion. The copy in the R component is made for the purpose of proof reconstruction. The database S_i of clauses may either expand or shrink, and therefore it is *non-monotonic*.

Forward contraction applies to a clause before it is established in the database; it can be seen as part of the process that leads to the installation of a new clause in the database. With backward contraction, every clause in the database may be subject to contraction. Thus, the notion of *persistent clauses* becomes relevant: a clause is *persistent* if it is never deleted after it has entered S_i at some stage i, $i \geq 0$. The set of persistent clauses, called the *limit* of the derivation, is defined as $S_\infty = \bigcup_{i\geq 0} \bigcap_{j\geq i} S_j$.

The notions of *refutation*, *refutational completeness*, and *proof reconstruction* are the same as for expansion-oriented strategies. Assume $\Box \in S_k$: while an expansion-oriented strategy finds in S_k all ancestors of $\Box$, as clauses deleted by forward contraction are not premises of other steps, a contraction-based strategy reconstructs the proof from $S_k \uplus R_k$. Indeed, clauses deleted by backward contraction may be ancestors of $\Box$, because they may have been used as premises before being deleted, and therefore they may be parents of other clauses. Proof reconstruction is the reason for the R_i component.

If the N_i and R_i components are omitted, the derivation has the form

$$S_0 \vdash S_1 \vdash \ldots S_i \vdash \ldots$$

where $S_0 = S$ is the input set, and at every step S_{i+1} is derived from S_i by an inference that can be either an expansion or a contraction inference.

A key monotonicity property of contraction-based derivations is $\rho(S_0) \subseteq \rho(S_1) \subseteq \ldots \subseteq \rho(S_i) \subseteq \ldots$, where $\rho(S)$ is the set of clauses that are *redundant* in S. This monotonicity property means that if a clause is redundant at a certain stage of the derivation, it will be redundant at all subsequent stages (cf. Lemma 2.6.4 in [26]), a principle later popularized by the slogan *"once redundant, always redundant"* [40]. A clause is redundant in S if adding it to or removing it from S neither improves nor worsens minimal proofs, where improving means making smaller, and worsening means making larger, with respect to a well-founded *proof ordering* (see [52, 40] and Chapter 2 of [26]). Clauses that are not redundant are called *irredundant*. Clauses deleted by contraction rules are redundant, and so are clauses whose generation is prevented by the ordering-based restrictions of expansion rules.

The notion of redundancy is extended from clauses to inferences: an inference is *redundant* if it uses or generates redundant clauses, and *irredundant* otherwise. In turn, redundancy is connected with *fairness*: intuitively, the two concepts are dual, because redundancy aims at capturing which inferences can be ignored, and fairness aims at capturing which inferences must be considered to find a proof. Refutational completeness of the inference system ensures that if the input set S is unsatisfiable, then there exist refutations. Fairness is the complementary property: if refutations exist, a fair derivation is guaranteed to be a refutation. Similarly to redundancy, also fairness is defined based on proof orderings: whenever a minimal proof of the target theorem is reducible by inferences, it is reduced eventually [26, 52, 40]. In practice, a derivation is *fair*, if all irredundant inferences are considered eventually. A search plan is *fair*, if it generates a fair derivation for all inputs. The combination of a *refutationally complete inference system* and a *fair search plan* yields a *complete theorem-proving strategy*.

Contraction-based strategies feature a search plan that prioritizes contraction over expansion, in order to ensure that redundant clauses are deleted prior to being selected as expansion premises. Such a search plan is called a *simplification-first* [44], *contraction-first* [49], or *eager-contraction* [28] search plan.

2.3 Instance-Based Strategies

All first-order clausal theorem-proving strategies can be seen as ways to implement *Herbrand's theorem*, which says that a set S of first-order clauses is unsatisfiable if and only if there exists a finite set of ground instances of clauses of S that is unsatisfiable [70]. The semi-decidability of first-order theorem proving descends from this theorem. *Instance-based strategies* represent the theorem-proving paradigm most directly inspired by Herbrand's theorem. The basic idea is to generate ground instances of input clauses, and test them for propositional unsatisfiability. The first such procedure was the *Gilmore method* [70], followed by SATCHMO [157], and *hyperlinking* [138], the latter at the beginning of the renewed interest in the Davis-Putnam-Logemann-Loveland (DPLL) procedure [79, 78, 70] for propositional satisfiability.

A clause $L_1 \vee \ldots \vee L_q$, called the *nucleus*, and clauses $L'_1 \vee D_1, \ldots, L'_q \vee D_q$, with $q \geq 1$, called *electrons*, such that $L_i\sigma = \neg L'_i\sigma$ for all i, $1 \leq i \leq q$, form a *hyperlink*. Hyperlinking generates the instance of the nucleus $(L_1 \vee \ldots \vee L_q)\sigma$. An instance generated by hyperlinking is termed a *hyperinstance*. Variants of the same clause may be used in a hyperlink, and all literals of the nucleus are linked, since the purpose is not to generate a hyperresolvent (see Section 2.2), but to instantiate the nucleus as much as possible. Since only instances are generated, all contraction is forward contraction, limited to unit subsumption and clausal simplification, because unrestricted subsumption would delete all instances and defeat the purpose of the strategy. An *instance-based derivation* has the form

$$(S_0;F_0) \vdash (S_1;F_1) \vdash \ldots (S_i;F_i) \vdash \ldots$$

where S_i is the set of clauses in the database of clauses, those with an identifier and ready to be used as premises, and F_i is the set of *generated instances*. Initially, $S_0 = S$ is the input set of clauses and $F_0 = \emptyset$. Instance generation takes premises in S_i and adds new clauses to F_{i+1}. Forward contraction deletes clauses in F_i and adds their non-trivial normal forms to F_{i+1}. If all hyperlinks in S_i have been considered, and contraction has been applied, all clauses in $S_i \cup F_i$ are made ground by replacing all variables with a new constant. A SAT solver is applied to the resulting ground set: if it is unsatisfiable, the procedure halts successfully; otherwise, the next phase of hyperlinking starts with state $(S_{i+1};F_{i+1})$ where $S_{i+1} = S_i \cup F_i$ and $F_{i+1} = \emptyset$. It follows that $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$, that is, the database of clauses is *monotonically increasing*.

While early instance-based strategies had a generate-and-test flavor, succeeding ones, such as *CLINS-S* [72, 73], *ordered semantic hyperlinking* (OSHL) [179, 178], *Inst-Gen* [104, 105, 134, 133], as well as strategies that hybridize instance generation and tableaux, such as the *disconnection calculus* [25, 143, 145, 146] and *hypertableaux* [15, 23], progressively emphasized *model-driven* instance generation, putting model building in the driver's seat. The model-building component of the procedure maintains a candidate model. The instance-generation component generates ground instances that are *false* in the model in order to exclude it. The model-building component updates the model to satisfy those ground instances, and the game continues until a contradiction arises.

In summary, for subgoal-reduction strategies the database of clauses is *fixed* and equal to the input set, hence relatively *small*; for expansion-oriented and instance-based strategies it is *large* and *monotonically increasing*; for contraction-based strategies it is *large* and *non-monotonic*. Since expansion-oriented strategies today have mostly pedagogical and historical interest, from now on we use ordering-based strategies to mean contraction-based strategies.

3 Parallelization of Theorem Proving

We distinguish three types of parallelism for deduction: *fine-grain parallelism* or *parallelism at the term/literal level*, *medium-grain parallelism* or *parallelism at the clause level*, and *coarse-grain parallelism* or *parallelism at the search level*.

In parallelism at the term/literal level, the parallelization affects operations *below the inference level*, as in *parallel rewriting*, where parallel rewrite steps together make a normalization inference, or *below the clause level*, as in AND-*parallelism*, where alternative inferences apply in parallel to different literals of a clause. In parallelism at the clause level, the parallelization affects operations *at the inference level*, so that parallelism at the clause level means *parallel inferences*. The possibility of *conflicts* between parallel inferences, and the impact of backward contraction on their incidence, emerge as key issues for fine and medium-grain parallelism. This

discovery [26, 49] led to the move *from parallelism at the clause level to parallelism at the search level* pioneered by Clause-Diffusion.

In parallelism at the search level, the parallelization affects entire derivations, as multiple processes search in parallel for a proof, so that parallelism at the search level means *parallel search* and involves *communication* among the processes. Parallel search yields *multi-search*, where the parallel processes employ different search plans, *distributed search*, where the search space is subdivided among the processes, and their combination.

3.1 Parallelism at the Term or Literal Level

The classification of types of parallelism is based on the granularity of data accessed in parallel, leading to distinctions between *fine-grain*, *medium-grain*, and *coarse-grain* parallelism. Intuitively, the finer the granularity, the higher the possibility that parallel processes result in *conflicts*. For inferences, *fine-grain parallelism* means having parallel processes access in parallel distinct *terms* or *literals* of a clause, so that fine-grain parallelism is *parallelism at the term or literal level*.

3.1.1 Parallelism at the Literal Level for Subgoal-Reduction Strategies

For subgoal-reduction strategies, fine-grain parallelism is AND-*parallelism*, where parallel processes access and reduce in parallel distinct literals of a goal clause. However, literals of the same clause may *share variables*, so that the parallel processes may be *in conflict*, in the sense that they need to instantiate the same variables by different unifiers.

For example, assume that the goal clause φ contains literals $\neg P(x)$ and $\neg Q(x,y)$, where P and Q are predicate symbols, and x and y are variable symbols. The two literals share the variable x. Let S include the clauses $\psi_1 = P(a) \vee C$ and $\psi_2 = Q(f(z),z) \vee D$, where a is a constant symbol, f is a function symbol, and z is a variable symbol. A process that resolves upon $\neg P(x)$ and $P(a)$ and a process that resolves upon $\neg Q(x,y)$ and $Q(f(z),z)$ are in conflict, because the first one needs to apply the substitution $x \leftarrow a$ and the second one needs to apply the substitution $x \leftarrow f(z)$. For this reason, already early provers parallelizing subgoal-reduction strategies, such as PARTHEO [192], METEOR [6], and Parthenon [74, 62], avoided AND-parallelism.

3.1.2 Parallelism at the Term Level for Ordering-Based Strategies

For ordering-based strategies, fine-grain parallelism is *parallel term rewriting*, where a term t is rewritten by applying in parallel multiple rewrite rules, or equa-

tions applied according to the ordering $\succ$. Given two equations that apply to a term t, it is wellknown that there are three cases [130].

The first possibility is that the two equations rewrite t at *disjoint positions*. For example, the equations $i(i(x)) \simeq x$ and $f(x,y) \simeq f(y,x)$ match disjoint positions of the term $h(i(i(a)), f(a,b))$, where f, h, and i are function symbols, and a and b are constant symbols. The two steps can be applied in parallel, yielding $h(a, f(b,a))$, under an ordering $\succ$ where $a \succ b$.

The second possibility is that the two equations have a *variable overlap*. For example, the equations (1) $h(x,x) \simeq x$ and (2) $f(y,b) \simeq y$ overlap at a variable position in $f(h(a,a),b)$, because $h(x,x)$ matches with $h(a,a)$, $f(y,b)$ matches with $f(h(a,a),b)$, and the position of $h(a,a)$ corresponds to that of the variable y in $f(y,b)$. The two equations can be applied in either order, because the two rewriting sequences $f(h(a,a),b) \rightarrow_{(1)} f(a,b) \rightarrow_{(2)} a$ and $f(h(a,a),b) \rightarrow_{(2)} h(a,a) \rightarrow_{(1)} a$ yield the same result.

The third possibility is that the two equations *overlap* at a *non-variable* position. For example, the equations (1) $f(z,e) \simeq z$ and (2) $f(l(x,y),y) \simeq x$, where e is another constant symbol and l another function symbol, overlap at a non-variable position in $f(l(a,e),e)$, as both match the whole term. It is impossible to apply both equations, because the first one rewrites the term to $l(a,e)$ and the second one to a, as shown in the peak $l(a,e) \leftarrow_{(1)} f(l(a,e),e) \rightarrow_{(2)} a$. The two rewriting steps are *in conflict*, as they aim at replacing the same term with different terms.

An *overlap* tout court is a non-variable overlap: two equations *overlap* if the left-hand side of one unifies with a *non-variable* subterm of the other. An overlap is a pre-condition to apply superposition. In the above example, the left-hand sides $f(z,e)$ and $f(l(x,y),y)$ of the equations (1) $f(z,e) \simeq z$ and (2) $f(l(x,y),y) \simeq x$ overlap as they unify with most general unifier $\{y \leftarrow e, z \leftarrow l(x,e)\}$. Indeed, superposition generates from the two equations the new equation $l(x,e) \simeq x$, closing the peak $l(x,e) \leftarrow_{(1)} f(l(x,e),e) \rightarrow_{(2)} x$, of which the above peak is an instance. A sufficient and necessary condition to avoid conflicts is to exclude the non-variable overlap case by requiring the equations to be *non-overlapping*, which means that neither left-hand side unifies with a *non-variable subterm* of the other.

Historically, *parallel rewriting* [107, 108, 94, 127] allows parallel processes to apply in parallel equations that match the term at disjoint positions, while *concurrent rewriting* [129, 2, 3] allows them to apply equations that match at disjoint positions or have a variable overlap.

In equational declarative languages for specification or programming, equations are required to be *regular*, that is, non-overlapping and *left-linear*. The latter property says that no variable occurs repeated in the left-hand side. Regularity suffices to ensure uniqueness of normal forms, which means that the set of equations defines a functional program, in the sense that the output $t\downarrow$ is unique for a given input term t to be reduced [117]. Thus, the study of parallel or concurrent term rewriting was motivated primarily by the quest for fast implementations of interpreters of equational declarative languages [107, 108, 94, 127].

In theorem proving it is impossible to restrict our attention to non-overlapping equations, since this would mean barring superposition, which is the main expansion

inference rule to deduce equations from equations. The same consideration applies to *Knuth-Bendix completion*, where superposition first appeared [130]. Nevertheless, the possibility of implementing Knuth-Bendix completion on top of parallel [65] or concurrent [128] rewriting, the latter only in the ground case, was explored. If all equations are ground, superposition collapses to simplification, and all operations of completion are done by rewriting. In the non-ground case, superposition is done sequentially, and only normalization can take advantage of parallel rewriting.

3.2 Parallelism at the Clause Level

Medium-grain parallelism for inferences means having parallel processes access in parallel distinct *clauses*, and perform one or more inferences with those clauses as premises. Thus, medium-grain parallelism is *parallelism at the clause level*.

3.2.1 Parallelism at the Clause Level for Subgoal-Reduction and Instance-Based Strategies

For subgoal-reduction strategies parallelism at the clause level is OR-*parallelism*, where parallel processes access in parallel distinct goal clauses, and resolve them with as many side clauses to generate new goal clauses. This means trying in parallel the proof attempts that a sequential strategy tries in sequence by going from one to the next via backtracking. Each goal clause is seen as a *task* (φ, j, k), where φ is a goal clause, j is the number of ME-extension steps used to generate φ, and k is the limit of iterative deepening. The task consists of reducing φ to $\Box$ by applying at most $k - j$ ME-extension steps. When a new goal clause φ_{i+1} is generated from a goal clause φ_i, a new task $(\varphi_{i+1}, j+1, k)$ is generated from task (φ_i, j, k). A task (φ, j, k) is active only if $j < k$.

Assume that there are n processes, all with initial limit k for iterative deepening. As soon as there are n active tasks, all processes may be active. A way of initializing the derivation is to have a sequential preprocessing phase where one process proceeds sequentially, generating at least n tasks. Then, a *parallel subgoal-reduction derivation*, with parallelism at the clause level, has the form

$$(S; G_0) \vdash (S; G_1) \vdash \ldots (S; G_i) \vdash \ldots$$

where the G_i component represents the set of active tasks.

Each process maintains a *queue* of its active tasks and the distribution of tasks among the processes is realized by *task stealing*. When the queue of a process is empty, that process steals active tasks from the queues of others. Task stealing is implemented by representing a task (φ, j, k) by an encoding of the WAM operations (see Section 2.1) that generate (φ, j, k) from the input set of clauses. When there are no more active tasks, the search restarts with the initial goal and a higher

limit of iterative deepening. Communication of tasks is achieved by message passing in PARTHEO [192], in shared memory in Parthenon [74, 62], and either way in METEOR [6].

For instance-based strategies, parallelism at the clause level means that multiple parallel processes pick different clauses as nuclei and generate in parallel all their hyperinstances [219]. However, a most natural way to parallelize hyperlinking [138] is to execute in parallel the instance-generation and satisfiability-testing phases [219]. This may be an example where parallelization contributed to improving the underlying theorem-proving method, as the notion of doing in parallel instance generation and satisfiability testing may have given ammunition to the design of instance-based strategies with a tighter integration between model building and instance generation (see Section 2.3).

3.2.2 Parallelism at the Clause Level for Ordering-Based Strategies

ROO [152, 153, 154] is the paradigmatic example of parallelism at the clause level for ordering-based strategies. ROO is a parallelization of up to version 2.2 of the OTTER *theorem prover* [162, 163, 164, 167]. The idea of ROO is to parallelize the *given-clause algorithm* at the heart of OTTER. This algorithm was later adopted by most theorem provers implementing ordering-based strategies, such as SPASS [214], E [188] and its predecessor DISCOUNT [9, 89], Vampire [135], Gandalf [207], WALDMEISTER [116], and Zipperposition [76]. In the given-clause algorithm the database of clauses is organized as two lists of clauses that we call `already-selected` and `to-be-selected` [34, 40]. In OTTER, these lists were named originally `axioms` and `set-of-support`, abbreviated `sos`; in later versions `axioms` was renamed `usable`. In E these lists are called `active` and `passive`.

The standard initialization is to start with all input clauses in `to-be-selected` and empty `already-selected`. For a strategy with set of support (see Section 2.2), one starts with `already-selected` containing the clauses of T and `to-be-selected` those of *SOS*, which explains the original names of the two lists in OTTER.

The given-clause algorithm performs a loop, until either a refutation is found or the list `to-be-selected` becomes empty. In the latter case the input set of clauses is recognized to be satisfiable. For a first-order theorem prover, termination typically occurs either with a refutation or when the prover hits a predefined time or space threshold. At every iteration of the loop, the prover selects from `to-be-selected` the *best* clause according to a heuristic evaluation function [4, 164, 87, 32, 167, 189]. This clause is the *given clause*. Thus, the given-clause algorithm realizes a *best-first search*. The prover performs all applicable expansion inferences having as premises the given clause and clauses in `already-selected` and moves the given clause from `to-be-selected` to `already-selected`. Every raw clause φ thus generated is subject to forward contraction, so that it is

either deleted or reduced to a normal form $\varphi\downarrow$ (where $\varphi\downarrow$ and φ may be identical), which gets an identifier and is appended to `to-be-selected`.

For backward contraction, the prover detects which previously existing clause ψ can be contracted by such a $\varphi\downarrow$ and subjects every such ψ to forward contraction. Any resulting $\psi\downarrow$ gets a *new identifier*, is added to `to-be-selected`, and will try in turn to backward-contract other clauses. In the case of backward contraction it cannot be that ψ and $\psi\downarrow$ are identical, because ψ was found reducible to begin with. The backward contraction phase terminates when the set of clauses S is such that $\rho(S) = \emptyset$, which is guaranteed to occur eventually thanks to the well-foundedness of the ordering $\succ$.

The OTTER version [167] of the given-clause algorithm applies backward contraction to both lists, so that the set S such that $\rho(S) = \emptyset$ at the end of every iteration of the loop is the union of `already-selected` and `to-be-selected`. The E version [186, 188], tried first in DISCOUNT [9, 89], applies backward contraction only to `already-selected`, so that the set S such that $\rho(S) = \emptyset$ at the end of every iteration of the loop contains the clauses in `already-selected`. If a clause in `already-selected` is backward contracted, its descendants in `to-be-selected` are deleted.

The idea of the E version is that it is not necessary to keep `to-be-selected` fully reduced, since clauses in `to-be-selected` are not used as premises of expansion inferences. Since `to-be-selected` is allowed to contain redundant clauses, the given clause is subject to forward contraction as soon as it is extracted from `to-be-selected` and before it is used as an expansion premise. Most contemporary provers implementing ordering-based strategies feature both the OTTER and E versions of the given-clause algorithm.

At the time of ROO, only the OTTER version of the given-clause algorithm existed. The concept of ROO is to store the lists `already-selected` and `to-be-selected` in shared memory, and let n parallel processes pick n given clauses and perform in parallel the ensuing expansion and forward contraction inferences. The expansion and forward contraction phases for a given clause are together called *task A*. The parallel processes are not allowed to append the clauses thus generated to `to-be-selected`, because that could cause conflicts in accessing the shared list, and more importantly because the clauses generated in parallel are not guaranteed to be irredundant. Indeed, if N_1 and N_2 are the sets of clauses generated by parallel processes p_1 and p_2, the clauses in N_1 are not forward contracted with respect to those in N_2 and vice versa. ROO features an additional list, termed the `K-list`, and lets the parallel processes append their new clauses to the `K-list`. A single process performs contraction within the `K-list` and then transfers all clauses from the `K-list` to `to-be-selected`. This activity is called *task B*.

All the more, the parallel processes are not allowed to do backward contraction, in order to avoid conflicts in deleting or rewriting clauses in `already-selected` and `to-be-selected` in shared memory. They are only allowed to test for backward contraction: if a parallel process discovers that one of its newly generated clauses can backward contract a clause ψ in `already-selected` or `to-be-selected`, it adds the identifier of ψ to a list named `to-be-deleted`.

The single process in charge of task B then proceeds to backward contract every such ψ. All processes follow the same schedule: execute task B, if either `K-list` or `to-be-deleted` is not empty and no other process is doing task B; execute task A otherwise.

Thus, ROO has to do backward contraction sequentially, as only one process is allowed to execute task B at any given time. Since an ordering-based prover typically spends most of its time doing contraction, and especially backward contraction, ROO incurs a problem identified as the *backward-contraction bottleneck* [26, 49], which manifests itself as follows: the single process executing task B lags behind, `K-list` and `to-be-deleted` grow too long, and the other processes remain idle waiting for clauses to reach `to-be-selected` and become available as given clauses.

The backward-contraction bottleneck affects also the application of the *transition-based approach to parallel programming* [220] to *Knuth-Bendix completion* [221]. The considered version of Knuth-Bendix completion is the original one [130], which only handles rewrite rules and fails if it generates an equation that cannot be oriented by the ordering into a rewrite rule. For theorem proving, *unfailing* or *ordered* completion [119, 11], which handles also equations, and therefore does not fail, supersedes the original Knuth-Bendix completion [130]. Nevertheless, the transition-based parallelization of Knuth-Bendix completion [221] is relevant to our analysis as another instance of parallelism at the clause level. It performs parallel inferences in shared memory, with locks and critical regions to prevent conflicts between inference steps that involve the same rewrite rules. Backward contraction causes a *write-bottleneck* as all the backward-contraction inferences request write-access to shared memory. It is plausible that the difficulty with backward contraction suggested applying the transition-based approach to the Buchberger algorithm [67, 68] instead, since in the Buchberger algorithm backward contraction is not as crucial (cf. Section 2.2.1).

Because of the backward-contraction bottleneck, parallelism at the clause level was largely abandoned, and an approach à la ROO was never tried in combination with the E version of the given-clause algorithm.

3.3 The Rise of Parallel Search

The above analysis of parallelism at the term/literal and clause levels reveals that a key element to understanding whether and how theorem proving can be parallelized is an abstract analysis of the *conflicts* between parallel inferences [26, 49]. The analysis is abstract in the sense of not being tied to a memory model or an implementation.

Two expansion inferences read their premises and generate and add their consequences. If they add their consequences to a shared data structure, some access control is required, but two expansion inferences are not in conflict in an essential way, because they only read their premises. On the other hand, contraction infer-

ences delete or rewrite one of their premises, and therefore determine three types of conflicts:

1. *Write-write conflict between contraction inferences*: two contraction steps aim at rewriting the same clause φ;
2. *Write-read conflict between contraction inferences*: a contraction step aims at rewriting a clause φ that another contraction step aims at using as a premise to contract some other clause ψ;
3. *Write-read conflict between contraction and expansion inferences*: a contraction step aims at rewriting a clause φ that an expansion step aims at using as a premise to generate other clauses.

Conflicts of Type (1) are exemplified by the conflicts in parallel rewriting (see Section 3.1). Conflicts of Types (2) and (3) are due to backward contraction, because a raw clause is not used as a premise of another inference while it is subject to forward contraction. A conflict of Type (2) is harmless: the *once redundant always redundant* principle ensures that no matter which step commits, the other clause, whether φ or ψ, will still be reducible [26]. Conflicts of Type (3) are the most problematic, because φ is redundant, a clause generated by φ will also be redundant, and therefore the contraction step should have priority.

Subgoal-reduction strategies have a static database of clauses given by the input set and no contraction. The absence of contraction means *no conflicts* between inferences. A static, relatively small, database of clauses represents *read-only* data that can be kept in shared memory, and even compiled as done for declarative programs. Instance-based strategies have a monotonically increasing database of clauses given by the input set plus instances, and only forward contraction. The absence of backward contraction means *no conflicts* between inferences. This explains why approaches to parallelize subgoal-reduction and instance-based strategies adopted parallelism at the clause level (cf. Section 3.2.1).

The situation is different for ordering-based strategies: the database of generated clauses is large, and non-monotonic, in fact *highly dynamic*, due to backward contraction, which causes conflicts among inferences. There is *no read-only data*, as any clause can be rewritten by succeeding ones. Contraction is essential for equational reasoning: indeed subgoal-reduction and instance-based strategies as described thus far are for first-order logic, and ordering-based strategies for first-order logic with equality. This analysis motivates resorting to *coarse-grain parallelism* for ordering-based strategies for first-order logic with equality.

Coarse-grain parallelism in deduction is *parallelism at the search level* or *parallel search*: multiple processes $p_0, \ldots, p_{n-1}$ search in parallel for a proof, each developing *its own derivation* and maintaining *its own database of clauses*. It is sufficient that one of the processes succeeds, and as soon as that happens, all may halt. Since each process has its own database of clauses, the issue of conflicts disappears, and especially the *backward-contraction bottleneck* cannot arise. While parallelism at the term/literal or clause levels aims at speeding up a given search, parallelism at the search level aims at finding a proof sooner by generating new searches, or by searching in different ways.

The counterpart of allowing every process to build its own database of clauses is the redundancy of having the same clauses in all or some of these databases. However, this duplication is not considered redundancy in parallel search, as long as the clauses are not redundant in the logical sense (cf. Section 2.2.2). Moreover, as long as contraction inferences are adequate, having φ in the database of a process and $\varphi \downarrow$ in the database of another process is not a correctness issue, unlike in other distributed applications, where the lack of *agreement* or *coherence* may affect correctness.

A general issue with parallelism at the search level is to *differentiate* the searches conducted in parallel by the processes. Intuitively, it is wasteful to have different processes visit the same search space, performing the same inferences in the same order. On the other hand, it is unavoidable that their searches have something in common, given that they are all solving the same problem. The idea is *to minimize the overlap* of the searches performed by the parallel processes [26, 49, 50, 29, 30, 31, 33, 37].

One approach to this issue is to differentiate the processes by letting them execute *different search plans on the same data*. The dual approach is to differentiate them by letting them execute *the same search plan on different data*. A way to differentiate data is to *subdivide clauses and inferences* among the processes, in order to subdivide the work to be executed. In parallel theorem proving this distinction was presented first as *competition* versus *cooperation* [190, 206, 84]. The analogue in parallel SAT-solving is the dichotomy between the *portfolio* approach and the *divide-and-conquer* approach. However, these terminologies suggest that the two parallelization principles may not coexist, while several methods explore their combination. We distinguish between *multi-search* and *distributed search* [36, 37].

Both multi-search and distributed search approaches feature *communication* among the parallel deductive processes. In the case of distributed search the rationale for communication is obvious: since the space to be searched is subdivided, communication is needed for completeness. However, also in the case of multi-search communication is necessary, otherwise multi-search reduces to running independent experiments in parallel.

For *subgoal-reduction strategies*, parallel search is typically *multi-search*, because the database of clauses is small and static. On the other hand, the large database of generated and kept clauses of ordering-based strategies suggests distributed search, and the notion of subdividing the search space by subdividing clauses and inferences. Since ordering-based strategies also offer a variety of search plans, *both multi-search and distributed search* have been applied to *ordering-based strategies*. In the rest of this section, we cover multi-search for subgoal-reduction strategies, multi-search for ordering-based strategies, and distributed search for ordering-based strategies.

3.4 Multi-search

A *multi-search method* is a parallel search method where the parallel deductive processes apply different search plans to search for a solution. As a way to differentiate the searches further, multi-search may also allow the processes to employ different inference systems. In multi-search with *homogeneous systems*, the deductive processes have different search plans and the same inference system. In multi-search approaches with *heterogeneous systems*, the deductive processes differ in the inference system or in both inference system and search plan.

3.4.1 Multi-search for Subgoal-Reduction Strategies

For subgoal-reduction strategies, ways of differentiating the search plans include assigning to the parallel processes different literal-selection rules [203], different clause-selection rules, different limits for iterative deepening, different choices of initial goal clause, or any combination thereof. These possibilities have been explored in the successors of PARTHEO [192], namely SETHEO, E-SETHEO, SPTHEO, CPTHEO and P-SETHEO [142, 169, 205, 103, 215].

A *multi-search subgoal-reduction derivation* with n processes $p_0, \ldots, p_{n-1}$ takes the form

$$(S; G_0^j) \vdash (S; G_1^j) \vdash \ldots (S; G_i^j) \vdash \ldots$$

where S is the input set of clauses, and G_i^j is the set of active tasks at process p_j, $0 \leq j \leq n-1$, and stage i, $i \geq 0$ (see Section 3.2.1 for the notion of task). The processes may communicate tasks, so that each process may have a set of active tasks as an effect of communication. If the processes start with different limits $k_0, \ldots, k_{n-1}$ for iterative deepening, a process may have in its set active tasks with different limits, such as (φ, n, k) and (φ', n', k'): if $k < k'$, task (φ, n, k) must be given higher priority by the process, in order to preserve completeness.

An example of a heterogeneous system is HPDS [204], with three deductive processes and a *deduction controller*. The three deductive processes execute *guided linear deduction* (GLD), which is similar to model elimination (cf. Section 2.1), hyper-resolution (HR) (cf. Section 2.2), and *unit-resulting resolution* (UR) [200], respectively. The UR inference rule resolves a clause $L_1 \vee \ldots \vee L_q \vee L_{q+1}$, called the *nucleus*, and unit clauses $L'_1, \ldots, L'_q$, with $q \geq 1$, called *electrons*, such that $L_i\sigma = \neg L'_i\sigma$ for all i, $1 \leq i \leq q$, to generate the unit clause $L_{q+1}\sigma$. If the L_{q+1} literal is allowed to be absent, UR resolution is able to generate $\square$. As UR resolution alone does not form a refutationally complete inference system, its purpose is to accelerate the generation of unit clauses for other inference rules. For example, UR resolution is used to generate unit lemmas for a PTTP prover [200].

HPDS implements this concept in a parallel setting. Every process is endowed with forward and backward subsumption, employs a DFS plan with iterative deepening, and sends the clauses it generates, including subsumed clauses tagged as such, to the deduction controller. The deduction controller forwards to the GLD and

HR processes the unit clauses generated by the UR process, and feeds the latter with the clauses generated by the other two. It may also forward to the HR process clauses generated by GLD, but not vice versa, so that the GLD process only receives unit lemmas. Furthermore, the deduction controller gives every process information on clauses subsumed by the other processes.

Another instance of multi-search with heterogeneous systems is CPTHEO [103], built on top of the model elimination prover SETHEO [142]. SETHEO is equipped with a resolution-based prover preprocessor, named Delta [191], with the idea of generating in advance, by resolution, clauses that could be useful as lemmas for the subgoal-reduction derivation. CPTHEO replaces preprocessing with cooperation in a parallel setting: it launches SETHEO and Delta in parallel, and lets SETHEO use the clauses generated by Delta as lemmas, according to different communication schemes.

For instance, SETHEO sends to Delta goal clauses from tasks that SETHEO cannot solve within the current limit of iterative deepening. Delta responds with lemmas that resolve with those goal clauses. SETHEO restarts with its next round of iterative deepening and a database of clauses enriched with the received lemmas.

Alternatively, SETHEO sends to Delta the literals labeling the open leaves in its current tableau. Delta replies by sending lemmas including *similar* literals of opposite sign. Similarity is measured according to various heuristic criteria [102]. In either scheme Delta ranks its generated clauses and selects the best to be sent as lemmas. The ranking is based on clause size, with small size deemed preferable, size of $\Pi(\varphi)$ for clause φ, with large size deemed preferable, assuming that a clause that required more inferences to be generated is more precious, and similarity-based criteria [102].

3.4.2 Multi-search for Ordering-Based Strategies

Multi-search for ordering-based strategies was introduced with the *Team-Work method* [83, 8, 87, 9, 88, 90, 91, 86, 195]. Team-Work was devised for purely equational problems, but its concept applies just as well to first-order logic with equality. The Team-Work method provides for n deductive processes $p_0, \ldots, p_{n-1}$, one of which also plays the role of *supervisor*. All processes start with the same input problem, the same inference system, and a time period, again the same for all processes, but different search plans. For instance, in the context of the given-clause algorithm, this may mean different evaluation functions to select the given clause [4, 164, 87, 32, 167, 189]. Every process develops its own derivation and builds its own database of clauses independently. When the allotted time period expires, every process evaluates its current database of clauses, based on a set of heuristic measures, the same for all processes. For example, the number of generated clauses may indicate how productive a process has been, while the number of deleted clauses may suggest whether the process has generated some very effective simplifiers or subsumers.

Then, every process sends to the supervisor its scores according to the heuristic measures. The supervisor picks a winner, the one with the best scores, and broadcasts its identity, say p_j. The winner p_j becomes the supervisor for the next round, and all the other processes send to p_j their best clauses according to other heuristic criteria, relative to individual clauses [4, 87], rather than the whole database. For example, an equation that has simplified many other clauses may be deemed precious. The new supervisor p_j broadcasts its database, enriched with these best clauses received from the others. In this manner, all deductive processes restart with the best database generated thus far and augmented with selected good clauses from the other derivations.

A *multi-search ordering-based derivation* with n processes $p_0, \ldots, p_{n-1}$ has the form

$$S_0^j \vdash S_1^j \vdash \ldots S_i^j \vdash \ldots$$

where S_i^j is the database of clauses at process p_j, $0 \leq j \leq n-1$, and stage i, $i \geq 0$. Initially, $S_0^0 = S_0^1 = \ldots = S_0^{n-1} = S$ is the input set of clauses. Such a derivation is a *refutation* if $\Box \in S_i^j$ for some i and j. A *Team-Work derivation* is a multi-search ordering-based derivation characterized by a series $\mathscr{A} = i_0 < i_1 < \ldots < i_k < i_{k+1} < \ldots$ of special stages, where $i_0 = 0$, and for all $i \in \mathscr{A}$, $S_i^0 = S_i^1 = \ldots = S_i^{n-1}$: the stages in $\mathscr{A}$ are those where all the processes restart with the same database.

Fairness of a multi-search derivation does not require that all search plans be fair. In the context of Team-Work, it is sufficient that at least one of the search plans is fair, and that a database produced by a fair search plan is selected as the winner infinitely often [8].

Starting at least with OTTER [162, 163, 164, 167], automated theorem provers have many options and parameters that can be set for each problem. A multi-search à la Team-Work adds even more, including the set of heuristics to evaluate databases, the set of heuristics to evaluate clauses, and the time period. One may also program the prover to vary selected parameters during a derivation. For example, the time period may increase over time, so that the processes cooperate a lot at the beginning and behave more independently later, or vice versa. The sequential basis for the implementation of Team-Work is the DISCOUNT theorem prover [9, 89], meaning that every p_j executes an instance of DISCOUNT.

The purpose of Team-Work is to *interleave* and *combine* different search plans. The periodic restart from a common database lets a process apply a search plan to a database generated by another search plan, realizing the interleaving. The mechanism whereby the database of the winner is enriched with clauses deemed good by other processes provides the combination. Since different search plans may generate clauses in different orders, their interleaving and combination may enable one of the processes to discover a proof sooner than any of the search plans would allow if applied sequentially. The downsides include the delays imposed by the periodic synchronizations, and the risk that the heuristics are misleading, so that discarding the databases with lower scores will make the search longer rather than shorter.

Ingredients of Team-Work appeared also in multi-search approaches with heterogeneous systems. For example, *requirement-based cooperative theorem proving*

[101] runs SPASS and DISCOUNT in parallel. The two provers communicate by *expansion requests* and *contraction requests*. In an *expansion request*, a prover sends to the other a clause φ, and the receiver replies by sending all resolvents between φ and the clauses in its `already-selected` list. In a *contraction request*, a prover sends to the other a clause φ, and the receiver replies by sending all its clauses that contract φ.

The TECHS system [85] is even more heterogeneous, as it runs in parallel SPASS, DISCOUNT, and SETHEO, thereby mixing contraction-based and subgoal-reduction strategies, a feature that recalls HPDS and CPTHEO (cf. Section 3.4.1). In TECHS, SPASS and DISCOUNT exchange equations, while SPASS and SETHEO exchange lemmas, from SPASS to SETHEO, and subgoals, from SETHEO to SPASS. These heterogeneous systems share with Team-Work the notion of heuristic selection of good clauses to be shared. For example, short clauses are deemed good, so that unit clauses and especially unit equations are the best.

The legacy of the Team-Work method is threefold. First, the notion of interleaving search plans migrated into the design of search plans for sequential theorem provers: the prover is programmed to execute a search plan for a fixed interval of time, then another one for the next interval, and so on. This feature is available, for instance, in Vampire [135]. This development is rather natural as interleaving is a standard way to simulate a parallel computation by a sequential computation. In the theory of parallel computing, a parallel computation that can be sequentially simulated by interleaving is not regarded as truly concurrent, although we are not aware of results on sequential derivations simulating multi-search derivations. Second, the notion of letting a process send to another one its best clauses is connected with *learning*, in the sense of learning the results of other derivations starting from the same problem [89]. This concept is generalized to learning from proofs of similar problems, as in the approaches that apply machine learning and big-data technologies to theorem proving [98, 209]. Third, Team-Work can be considered a forerunner of the *portfolio approach* to parallel SAT-solving (cf. Section 4.1 in this chapter and Chapter 1 on Parallel Satisfiability).

3.5 Distributed Search

A *distributed-search method* is a parallel-search method where the search space is *subdivided* among the parallel deductive processes, in order to subdivide the work to be performed, and possibly reach a solution sooner. As a way to differentiate the searches further, distributed search may also allow the processes to apply different search plans, leading to methods with *both distributed search and multi-search*.

In general, subdividing the work may mean subdividing data, as in *data-driven* parallelism, or subdividing operations, as in *operation-driven* parallelism. In theorem proving, there are typically few inference rules and a huge number of generated clauses, and therefore the subdivision and the parallelism are naturally data-driven. However, the subdivision is designed knowing which inferences need to be applied

to the clauses, so that the two aspects are intertwined. This also means that distributed search is usually coupled with *homogeneous systems*, where all deductive processes feature the same inference system, although in principle it could be combined also with *heterogeneous systems*, where the deductive processes employ different inference systems.

3.5.1 Distributed Search for Ordering-Based Strategies

Distributed search for ordering-based strategies was introduced with the *Clause-Diffusion methodology* [26, 47, 48, 50], implemented in the AQUARIUS [45, 26, 46, 51] and PEERS [57, 50] provers, and then investigated through *Modified Clause-Diffusion* [27, 28], the PEERS-MCD [29, 30, 32, 37] prover, and a formal analysis of distributed search for contraction-based proof search [31, 33]. To the best of our knowledge, Clause-Diffusion was the first parallel-search method for automated first-order theorem proving, and many of the elements of the analysis of parallelism for deduction (see Section 3.3) were discovered with and around Clause-Diffusion and its developments. The reason for calling it a *methodology* is that Clause-Diffusion came since the start with a choice of solutions for several issues. In this presentation we cover all issues and the most mature and most successful solutions, hence Modified Clause-Diffusion, referring the interested readers to the original articles for other possibilities.

3.5.2 The Basic Clause-Diffusion Mechanisms

Clause-Diffusion provides for n deductive processes $p_0, \ldots, p_{n-1}$, that are all *peers*. In a Clause-Diffusion prover, n is a parameter set by the user. All processes start with the same input problem, inference system, and search plan, although different search plans may be assigned. Every process develops its own derivation and builds its own database of clauses independently. The processes are *asynchronous*, as the only synchronization occurs when one sends all others a halting message. This happens, for example, when one of the processes finds a proof.

Clause-Diffusion subdivides the search space by subdividing clauses, so that *every clause is owned by a process*. A *distributed-search ordering-based derivation*, or *distributed derivation* for short, has the form

$$(O_0;NO_0)^j \vdash (O_1;NO_1)^j \vdash \ldots (O_i;NO_i)^j \vdash \ldots$$

where for every process p_j, $0 \leq j \leq n-1$, and stage i, $i \geq 0$, $S_i^j = O_i^j \uplus NO_i^j$ is the *local database* of clauses at p_j; O_i^j is the set of clauses *owned* by p_j; NO_i^j is the set of clauses *not owned* by p_j; and $\bigcup_{j=0}^{n-1} S_i^j$ represents the *global database* at stage i. Initially, $S_0^0 = S_0^1 = \ldots = S_0^{n-1} = S$ is the input set of clauses. In the early Clause-Diffusion terminology owned clauses are termed *residents* and the others *visitors* or

visiting clauses [26, 50]. A distributed derivation is a *refutation* if $\Box \in S_i^j$ for some i and j.

Since every clause is owned by a process, for every stage i, $i \geq 0$, we have $\bigcup_{j=0}^{n-1} O_i^j = \bigcup_{j=0}^{n-1} S_i^j$. This also means that every clause $\varphi \in NO_i^j$ is owned by some p_k, with $k \neq j$, so that $\varphi \in O_l^k$ for some $l \geq 0$. Furthermore, under the customary assumptions that every clause has its own variables, and variants are distinct clauses, every clause is owned by *only one* process, so that $O_i^j \cap O_i^k = \emptyset$ for all $i \geq 0$ and $0 \leq j \neq k \leq n-1$.

Assume that a clause ψ is generated by process p_j, and that its normal form after forward contraction $\varphi = \psi\downarrow$ is not trivial, so that φ is kept. *Regardless of whether ψ was generated by expansion or backward contraction*, process p_j assigns φ to some p_k according to an *allocation criterion*. The number k becomes part of the *identifier* of φ: for example, if φ is the m-th clause generated and kept by p_j, its identifier includes the fields $\langle k, m, j\rangle$. These three components suffice to identify a clause *uniquely* across all processes, so that the identifier of a clause is a *global* attribute.

If $k = j$, p_j adds φ to O^j; if $k \neq j$, p_j adds φ to NO^j. Either way, p_j applies φ to backward-contract clauses in S^j, and broadcasts it as an *inference message* $\langle \varphi, k, m, j\rangle$ to all other processes. This broadcasting mechanism is the reason for the name Clause-Diffusion, as clauses are *diffused*. These messages are called *inference messages*, because received clauses will be used for inferences.

Any other process p_q, $q \neq j$, upon receiving the inference message $\langle \varphi, k, m, j\rangle$ applies forward contraction to the received clause φ. If φ is deleted by forward contraction no other operation is needed. Otherwise, let $\varphi\downarrow$ be the normal form of φ with respect to S^q, where $\varphi\downarrow$ and φ may be identical. If $k = q$, p_q adds $\varphi\downarrow$ to O^q; if $k \neq q$, p_q adds $\varphi\downarrow$ to NO^q. Either way, p_q applies $\varphi\downarrow$ to backward-contract clauses in S^q.

3.5.3 The Subdivision of Clauses in Clause-Diffusion

Allocation criteria to subdivide clauses play an important role in differentiating the searches and limiting their *overlap* [31, 33]. The intuition is that different searches, and searches that differ from a sequential one, may enable one of the processes to find a proof sooner. A simple option is that each process assigns clauses according to a *round-robin schedule*, called *alternate-fit* [26, 50] or *rotate* [30]: p_j assigns φ to p_k for $k = (q+1)$ *mod* n, if p_j assigned the previous clause to p_q.

In the *half-alternate-fit* criterion [26, 50], p_j assigns every other clause to itself and in a round-robin manner otherwise. Let p_{q_1} and p_{q_2} be the two most recently used destinations; if $q_1 = j$, p_j assigns φ to p_k for $k = (q_2+1)$ *mod* n; if $q_1 \neq j$, p_j assigns φ to itself.

Alternatively, every process p_j may estimate the workload of each process as measured by the number of generated clauses, a criterion named *best-fit* [26, 50] or *select-min* [57]. Clearly, p_j knows exactly how many clauses it generated thus far. For all other processes p_q, $q \neq j$, p_j may consider the latest inference message

$\langle \psi, k, m, q \rangle$ received from p_q and take m as an estimate of the number of clauses generated by p_q. Then p_j assigns the next φ to the process with the smallest estimated workload. However such a criterion may lead the processes to assign too many clauses to others, since a process may underestimate the workload of others but not its own. Therefore, this criterion may be corrected by letting each process assign a fixed percentage of clauses to itself as in the *half-alternate-fit* criterion.

A different approach is to determine the owner of a clause *based on properties of the clause itself*. For example, assume that every symbol in the signature has an associated *weight*. The sum of the weights of the symbols occurring in a clause is the weight of the clause. This is a feature that the Clause-Diffusion provers inherit from OTTER, where it is used for *deletion by weight*, a contraction rule that allows the prover to delete all clauses whose weight is above a certain threshold [162, 163, 164, 167]. Such a rule is not adequate (see Section 2.2), but it may be useful in practice. A simple weight-based allocation criterion is to assign clause φ to p_k, where $k = w \bmod n$ and w is φ's weight. This criterion was called *syntax* in the PEERS prover [57, 50].

The next step is to use *information from the ancestor-graph* $\Pi(\varphi)$ (cf. Section 2.2.1) in order to allocate φ. Since theorem provers anyway save the data to generate $\Pi(\varphi)$ for every kept clause φ in order to be able to build $\Pi(\square)$, storing this information is no additional burden. This concept is achieved by the *ancestor-graph-oriented* (AGO) allocation criteria [30]. The general idea is to use information from the finite portion of the search space that has been generated to assign clauses to processes and therefore induce a subdivision of the search space that lies ahead.

The AGO criterion *parents* determines φ's owner by applying a function f to the identifiers of φ's parents. As the function f may vary, this is actually a family of criteria. If φ was generated from premises ψ_1 and ψ_2 by a binary expansion inference rule, such as resolution, paramodulation, or superposition, its parents are ψ_1 and ψ_2. If φ is a factor of ψ, its parent is ψ. If φ was obtained by normalizing ψ during backward contraction, ψ is considered as the sole parent. Since f is a function, clauses that have the same parent(s) are assigned to the same process. The intuition is that clauses that have the same parents are spatially close in the search space, and therefore should belong to the same process. If they were assigned to different processes, the effect could be to bring those different processes to be active in the same region of the search space, increasing their overlap.

The AGO criterion *majority* considers all ancestors of clause φ, that is, all clauses that occur in its ancestor-graph $\Pi(\varphi)$. It assigns to every process p_j a number of votes equal to the number of clauses in $\Pi(\varphi)$ owned by p_j. The process, say p_k, that gets the most votes owns φ. Ties are broken arbitrarily. The idea is that the process that owns the most ancestors of φ is already most active in the region of the search space where φ is, and therefore should get φ as well. Assigning φ to another process, say p_q, with $q \neq k$, could increase the overlap between p_k and p_q.

There remains what to do with input clauses. One process, say p_0, reads the input file and handles input clauses like raw clauses. Most allocation criteria listed above apply regardless of whether the clause was read or generated. The *select-min* criterion does not apply to input clauses, because at the beginning the processes have

no workload: therefore, *select-min* assigns input clauses in round-robin fashion. The AGO criteria do not apply to input clauses, because input clauses do not have ancestors. The *parents* criterion assigns all input clauses to p_0. The *majority* criterion cannot proceed in this manner, because otherwise all clauses would belong to p_0, as p_0 would always have the majority of ancestors. This does not happen with the *parents* criterion, because the function f applies to the entire identifiers of the parents, not only to the owners. Thus, also the *majority* criterion assigns input clauses in round-robin style.

3.5.4 The Subdivision of Inferences in Clause-Diffusion

In Clause-Diffusion the ownership of clauses induces a *subdivision of expansion inferences* as follows. Assume that p_j is about resolving clauses $\varphi = L \vee C$ and $\psi = \neg L' \vee D$, such that $L\sigma = L'\sigma$. Clause-Diffusion allows p_j to proceed with the inference if and only if p_j *owns* ψ, that is, *the parent with the negative literal resolved upon*. Similarly, assume that p_j is about paramodulating or superposing clause $\varphi = l \simeq r \vee C$ into clause $\psi = L[s] \vee D$, such that $s\sigma = l\sigma$. Clause-Diffusion allows p_j to proceed with the inference if and only if p_j *owns* ψ, that is, *the clause paramodulated or superposed into*. When paramodulating φ into ψ, the prover needs to consider all non-variable subterms of ψ and only l and r in φ. In other words, there is more work connected with the clause paramodulated into. For superposition, that is, paramodulation into equalities, a prover needs to test for both superposition of $\varphi = l \simeq r \vee C$ into $\psi = s \simeq t \vee D$ and superposition of ψ into φ. The owner of ψ will superpose φ into ψ and the owner of φ will superpose ψ into φ. For factoring, p_j is allowed to generate the factors of ψ if and only if it *owns* ψ. For hyperresolution and unit-resulting resolution, p_j is allowed to proceed if and only if it *owns the nucleus* of the inference step.

As far as contraction inferences are concerned, there is *no subdivision of forward-contraction inferences*, as every process p_j applies all the clauses in its current local database S^j to try to delete or reduce a raw clause it has generated. There is also *no subdivision of backward-contraction inferences that delete clauses*, such as subsumption, functional subsumption, or tautology elimination (cf. Section 2.2). Every process p_j is allowed to use any clause in S^j to delete any other clause in S^j by such an inference rule, regardless of ownership.

On the other hand, ownership is used to *subdivide backward-contraction inferences that generate new clauses*, such as clausal simplification and equational simplification or normalization. Assume that process p_j detects that clause $\varphi \in S^j$ can be backward-simplified by some other clause $\psi \in S^j$. If p_j owns φ, p_j is allowed to generate $\varphi\downarrow$. If p_j does not own φ, p_j is allowed to delete φ, but it is not allowed to generate $\varphi\downarrow$. Whoever owns φ will generate $\varphi\downarrow$, give it a *new identifier*, and broadcast it as an inference message.

3.5.5 Distributed Global Contraction, Distributed Fairness, and Distributed Proof Reconstruction

Clause-Diffusion led to the formulation and solution of three general issues in distributed search for ordering-based strategies: *distributed fairness* [26, 47, 50, 28], *distributed proof reconstruction* [28], and *distributed global contraction* [26, 50, 28].

Distributed fairness, that is, fairness of a distributed derivation, is guaranteed by two conditions. First, each process must be *locally fair*, which means it considers eventually all irredundant inferences. Second, all persistent irredundant clauses must be broadcast eventually. Clause-Diffusion fulfills the second condition eagerly, by broadcasting kept clauses right after forward contraction. The reason for this eager choice is the second property, namely *distributed proof reconstruction*.

Proof reconstruction requires the clauses deleted by backward contraction to be saved (cf. Section 2.2.2). Thus, the *distributed derivation* takes the form

$$(O_0;NO_0;R_0)^j \vdash (O_1;NO_1;R_1)^j \vdash \ldots (O_i;NO_i;R_i)^j \vdash \ldots$$

where for every process p_j, $0 \leq j \leq n-1$, and stage i, $i \geq 0$, $S_i^j = O_i^j \uplus NO_i^j$ is the database of clauses at process p_j and stage i, partitioned into owned (O_i^j) and not owned (NO_i^j) clauses, while R_i^j is the set of clauses that p_j deleted by backward contraction. *Distributed proof reconstruction* means that if $\Box \in S_i^k$, process p_k can reconstruct $\Pi(\Box)$ by consulting only $S_i^k \uplus R_i^k$. In order to guarantee distributed proof reconstruction, it is not sufficient that all persistent irredundant clauses be broadcast eventually, since clauses deleted by backward contraction, which are redundant and not persistent, may be needed to reconstruct the proof. A stronger, and sufficient, condition is that all clauses ever used as premises are broadcast. This is why Clause-Diffusion lets every process broadcast a clause φ after φ emerges from forward contraction, that is, as soon as φ is ready to be used as a premise [28].

The problem of *distributed global contraction* is to ensure that notwithstanding the subdivision of inferences among the parallel processes, if φ is globally redundant at some stage i, φ is recognized as redundant eventually by every process. Formally, if $\varphi \in \rho(\bigcup_{j=0}^{n-1} S_i^j)$ at some stage i, then for all processes p_j, $0 \leq j \leq n-1$, there exists a stage l, $l \geq i$, such that $\varphi \in \rho(S_l^j)$. Assume that $\varphi \in \bigcup_{j=0}^{n-1} S_i^j$, and $\varphi \in \rho(\bigcup_{j=0}^{n-1} S_i^j)$, because there is a $\psi \in \bigcup_{j=0}^{n-1} S_i^j$ such that ψ can delete φ by contraction. By the broadcasting mechanism of Clause-Diffusion, the two clauses are guaranteed to meet at every process, so that global redundancy becomes local redundancy. By fairness, every process eventually applies ψ to delete φ, so that global contraction becomes local contraction, unless the derivation succeeds sooner. Furthermore, by the subdivision of backward simplification, distributed global contraction is achieved while avoiding both the redundancy of letting all processes generate $\varphi\downarrow$ and the redundancy of preventing all processes but the owner from deleting φ.

In summary, Clause-Diffusion is a methodology to transform a sequential ordering-based theorem-proving strategy into a distributed one, in the sense that each

parallel process executes the sequential strategy, modified with subdivision of labor and communication according to Clause-Diffusion. If the requirements for distributed fairness are fulfilled, a complete sequential strategy yields a complete distributed strategy.

3.5.6 The Clause-Diffusion Provers

At the implementation level, Clause-Diffusion is a methodology to transform a sequential ordering-based theorem prover into a distributed one, and indeed all Clause-Diffusion provers have a pre-existing sequential code base.

The first Clause-Diffusion prototype is AQUARIUS [45, 26, 46, 51]. AQUARIUS is the parallelization of OTTER 2.2 [163], using PCN for communication by message passing [100, 69]. AQUARIUS implements the *rotate* allocation criterion, with variants such as letting every process p_j own the factors of φ if p_j owns φ, or even allowing every process to own all input clauses. The latter trick violates the principle that every clause is owned by *only one* process, and it was tried only to watch its effect in experiments, especially when the input clauses include the axioms of some theory. Since Otter implements *unfailing* or *ordered* completion [119, 11], AQUARIUS offers also a Clause-Diffusion parallelization of ordered completion. AQUARIUS features also multi-search, since its options enable the user to shut off the subdivision of clauses, so that every process assigns all its generated clauses to itself, and attach different search plans to the processes. For a Clause-Diffusion prover that uses the given-clause algorithm, different search plans may mean different evaluation functions to select the given clause [162, 4, 163, 164, 87, 32, 167, 189].

While AQUARIUS, like OTTER, handles first-order logic with equality, the subsequent Clause-Diffusion provers focus on equational logic. A reason for this choice is that a motivation for exploring distributed search is to avoid the backward-contraction bottleneck, and backward contraction is crucial to solve equational problems.

The second Clause-Diffusion prototype is PEERS [57, 50], whose name, chosen by Bill McCune, emphasizes that the deductive processes in Clause-Diffusion are peers. PEERS is the parallelization of code from the OTTER PARTS STORE (OPS), for theorem proving in equational theories possibly with *associative-commutative* (AC) function symbols, using p4 for communication by message passing [66]. If paramodulation is done modulo AC [174], there are generally so many AC-paramodulants that generating all AC-paramodulants between the given equation and all those in `already-selected` is too much for an iteration of the given-clause loop (cf. Section 3.2.2). Therefore, PEERS employs a variant of the given-clause algorithm, called the *pairs algorithm*: in every iteration of the loop the prover selects *a pair of equations* and performs all expansion inferences from the equations in the pair, provided at least one of them comes from `to-be-selected`. The evaluation function to select the best clause as given clause is replaced by an evaluation function that selects the best pair of equations.

PEERS implements the *rotate*, *syntax*, and *select-min* allocation criteria, with variants such as allowing every process p_j to own $\varphi\downarrow$ if p_j owns φ and $\varphi\downarrow$ is generated by backward contraction. Assume that the input set S is satisfiable. In principle, a theorem-proving strategy may not terminate, because it is a semi-decision procedure. In practice, a theorem prover terminates on a satisfiable input, because either it generates a finite saturated set (see Section 2.2), or, more likely, because it reaches a predefined threshold on running time or memory space. For the first kind of situation, PEERS implements the *Dijkstra-Pnueli global termination detection algorithm* [208] to recognize that all processes are idle. For the second kind of situation, a process p_k that has reached a threshold broadcasts a message to inform all others that it quits the search. Clause-Diffusion allows $p_0, \ldots, p_{k-1}, p_{k+1}, \ldots, p_{n-1}$ to continue, but in PEERS and its successors, for simplicity, such a message from p_k is a halting message.

The third Clause-Diffusion prototype is PEERS-MCD, thus named because it implements Modified Clause-Diffusion. The first version, called PEERS-MCD.A [28], is obtained by modifying PEERS to execute Modified Clause-Diffusion, still using code from OPS as the sequential base and p4 for message passing.

The second version, dubbed PEERS-MCD.B [29], is the parallelization, according to Modified Clause-Diffusion, of version 0.9 of the EQP prover [165] for equational theories possibly with *associative-commutative* (AC) function symbols. In addition to ordered paramodulation or superposition (cf. Section 2.2), EQP features *blocking* [197, 137, 10, 126] and *basic paramodulation* [13]. Blocking prevents a paramodulation step whose most general unifier contains at least a pair $x \leftarrow t$ where t is reducible. Basic paramodulation stipulates that a term is *basic*, if it is not introduced by a substitution, and restricts paramodulation and simplification to apply only to basic terms. The restriction to simplification is not implemented in EQP, renouncing refutational completeness. In terms of search plan, EQP features both the given-clause algorithm and the pairs algorithm. PEERS-MCD.B and its successors inherit all these features, and adopt the Message Passing Interface (MPI) and its implementation mpich for message passing [109].

PEERS-MCD.B is the first Clause-Diffusion prover to implement the AGO allocation criteria (see Section 3.5.3). The EQP prover made history by proving mechanically that *Robbins algebras are Boolean* [166, 77], a conjecture that remained open in mathematics since 1933 and was considered a challenge in automated theorem proving since 1990 [216]. Thanks to the AGO allocation criteria, PEERS-MCD.B exhibited *super-linear speedup* on several problems, including two lemmas representing two-thirds of the proof of the Robbins theorem [29, 30], and the *Levi commutator problem in group theory* [32].

The next version of PEERS-MCD is PEERS-MCD.C, which features version 0.9d of EQP as its sequential base. PEERS-MCD.C maintains the super-linear speedup in the first two lemmas that form the proof of the Robbins theorem, and adds an almost linear speedup in the third lemma [36].

PEERS-MCD.D [37] still has EQP0.9d as its sequential base. It differs from all previous versions of PEERS-MCD, because it offers distributed search, multi-search, and their combination. It can run in one of three modes: (1) *pure distributed-search*

mode: the search space is subdivided among the processes; all processes execute the same search plan; (2) *pure multi-search mode:* the search space is not subdivided; every process executes a different search plan; and (3) *hybrid mode:* the search space is subdivided, and the processes execute different search plans.

A first way to differentiate the search plans in PEERS-MCD.D is to have half the processes execute the given-clause algorithm and the other half execute the pairs algorithm. Another way is to let the processes employ different evaluation functions to select the given clause or pair of equations. The two ways may also be combined, if the number of processes is sufficiently high.

PEERS-MCD.D implements three heuristic evaluation functions to select given clauses based on their similarity with the target theorem [4, 87, 37]. If multi-search with similarity-based heuristics is selected, process p_k executes the given-clause algorithm with the first heuristic function if $k \ mod \ 3 = 0$, with the second heuristic function if $k \ mod \ 3 = 1$, and with the third heuristic function if $k \ mod \ 3 = 2$. These heuristics do not apply to the pairs algorithm.

PEERS-MCD.D also turns the `pick-given-ratio` parameter into a way of differentiating searches in multi-search. This parameter appeared first in OTTER [162, 163, 164, 167] and has been adopted by most ordering-based theorem provers [189]. It allows the prover to mix best-first search and breadth-first search: if the parameter `pick-given-ratio` has value x, the given-clause/pair algorithm picks the oldest, rather than the best, equation/pair once every $x+1$ choices. In other words, it picks the best according to the heuristic evaluation function x times, then the oldest, and then it repeats. PEERS-MCD.D lets each process use a different value of `pick-given-ratio`: if multi-search with different ratios is selected, process p_k resets its `pick-given-ratio` to $x+k$.

Prior to the Robbins theorem, another challenge problem for automated theorem provers was the *Moufang identities in alternative rings* [5]. Alternative rings are rings where the product is not associative. The first automated proofs of these identities by a sequential prover involve several ingredients [5], including inference rules that *build the cancellation laws* in the inference system [121]. PEERS-MCD.D proves the Moufang identities in alternative rings *without cancellation laws* and exhibiting several instances of *super-linear speedup* with respect to EQP0.9d [37]. This finding suggests that parallel search can even compensate for a weaker inference system. These results are obtained in pure distributed-search mode or hybrid mode, whereas multi-search alone shows no speedup at all. The best performances arise in hybrid mode. Thus, distributed search is necessary to conquer these problems, and the addition of multi-search improves the outcome further.

In summary, super-linear speedup by Clause-Diffusion is possible, precisely because parallel search, and all the more distributed search, does not mean executing in parallel the same steps of the sequential search, but generating a *different search* that may visit the search space in a different way. The analysis of the experiments shows that whenever there is a super-linear speedup, the Clause-Diffusion prover *generates fewer clauses* than the sequential prover, *retains a higher percentage* of them, and generates a *different proof* [30, 37]. Generating fewer clauses and retaining more of them suggest better focus and less redundancy. Thus, the interpretation

of the experiments is that an effective subdivision of the search space prevents the processes from overlapping too much, reduces the amount of redundancy, and allows the winning process to focus on a proof sooner. Since the proof is often not unique, these differences are also reflected in a different proof being found. Note that different proof does not necessarily mean shorter proof: in theorem proving a shorter proof may require a longer run. The observation of super-linear speedup also indicates that the sequential search plan is not optimal for the problem, which is not surprising, given the generic and still largely syntactic nature of most heuristics in theorem proving.

While generating a different search may yield a faster proof, up to the point of a super-linear speedup, it also means that *scalability* may be irregular. Precisely because the point is not to use more computers to do the same steps, there is no guarantee that the performance improves regularly with the number of processes. For example, it may happen that the performance scales well with up to six processes, and becomes worse with seven or eight. A pattern of this type suggests that the problem may not be hard enough to justify more computing power beyond a certain point, so that subdividing the search space further is counterproductive.

In other cases, the performance oscillates: two processes do better than one, but four do worse than two, and six speed up again; or, neither four nor six improve, but seven or eight do. In these instances, an explanation is that the subdivision of the search space in Clause-Diffusion depends on the number of processes, as it is done by dynamic allocation of generated clauses during the derivation. Assume that we have two processes p_0 and p_1. When we add a third process p_2, the portions of the search space assigned to p_0 and p_1 change with respect to what they were with two processes. The three searches developed by p_0, p_1, and p_2 differ from those developed by p_0 and p_1 when running as two processes. Since the result depends on the subdivision of the search, it may happen that two processes do better than four on a certain combination of problem and strategy. However, combining distributed search and multi-search may smooth these oscillations and improve scalability [37].

4 Discussion

In this section first we draw connections between parallel theorem proving and parallel satisfiability solving. The readers will find more by reading this chapter together with Chapter 1 on Parallel Satisfiability and Chapter 2 on Cube and Conquer. Then, we discuss future directions for research in parallelization of theorem proving in the light of advances in first-order model-based reasoning [43].

4.1 Parallel Theorem Proving and Parallel Satisfiability

The idea of subdivision of the search space in Clause-Diffusion influenced the design of the parallel SAT solver PSATO [223, 224], which is considered a forerunner of the *divide-and-conquer* approach to parallel SAT-solving. More generally, research in parallel SAT-solving inherited from research in parallel theorem proving the focus on *parallel search*. In addition, inferences and data in propositional logic are simpler than in first-order logic, so that there is no room for parallelism below or at the inference level. The concepts of *distributed search* and *multi-search* apply with the same meaning also in parallel SAT-solving, corresponding to the *divide-and-conquer* and *portfolio* approaches, respectively.

PSATO is a *distributed-search* parallelization of SATO [222, 225], which implements the DPLL procedure [79, 78, 70] for propositional satisfiability. The original Davis-Putnam (DP) procedure [79] is for first-order logic, and features propositional, or ground, resolution. The Davis-Putnam-Logemann-Loveland (DPLL) procedure [78] replaces propositional resolution with *splitting*, seen as breaking disjunctions apart by *case analysis*, to avoid the growth of clauses and the nondeterminism of resolution. Splitting is understood also as *guessing*, or *deciding*, the truth value of a propositional variable, in order *to search for a model* of the given set of clauses. Thus, DPLL is a *model-based* procedure, where all operations are centered around a candidate partial model, called the *context*, represented by a sequence, or *trail*, of literals.

A PSATO derivation features $n+1$ processes, with one master process that subdivides the work, and n client processes each searching for a model by executing SATO. The key idea is to subdivide the search space by using *guiding paths*. The notion of guiding path is inspired by the view of the search space of a SAT problem as the tree of recursive calls of the DPLL procedure. In this tree a node has typically two outgoing arcs, one labeled L and the other labeled $\neg L$, where L is a literal occurring in the input problem. The two arcs correspond to the two cases of the case-splitting on L (either L is *true* or L is *false*), and lead to the two ensuing recursive calls, one where L is asserted and one where $\neg L$ is asserted.

A guiding path is a path in this tree; it is represented as a sequence of pairs $\langle (L_1,\delta_1),(L_2,\delta_2),\ldots,(L_k,\delta_k)\rangle$, where, for $1 \leq i \leq k$, the L_i's are the literals labeling the path; $\delta_i = 1$, if L_i is a first child; and $\delta_i = 0$, if L_i is a second child. A node labeled $(L,1)$ is *open*, because L is still to be flipped; a node labeled $(L,0)$ is *closed*, because L has been already flipped. A *job* is given by a pair (S,P), where S is the input set of clauses and P is a guiding path. Given a path $P = \langle (L_1,0),(L_2,0),\ldots,(L_i,1),\ldots,(L_k,\delta_k)\rangle$, where i is the smallest index for which $\delta_i = 1$, two new *disjoint* paths are generated by splitting on L_i, yielding $P_1 = \langle (L_1,0),(L_2,0),\ldots,(\neg L_i,0)\rangle$ and $P_2 = \langle (L_1,0),(L_2,0),\ldots,(L_i,0),\ldots,(L_k,\delta_k)\rangle$.

In PSATO, the master process is responsible for preparing the jobs and assigning a job and a time limit to each client process. Every client will return either `sat` with a model of S; or `unsat`, meaning that its assigned subtree contains no model; or a guiding path, representing the search remaining when the time is up. The subtrees assigned to the clients are *disjoint* portions of a finite search space, so that the sub-

division has *no overlap* by definition. In contrast, in first-order theorem proving the search space is infinite, its representation is far more complex [54, 33, 38], and a strategy may at most try to *limit the overlap* of the searches by heuristic subdivision criteria as done in Clause-Diffusion (cf. Section 3.5.3).

The transition from the DPLL to the CDCL (Conflict-Driven Clause Learning) procedure [159, 160, 170, 158] is a game changer in parallel SAT-solving as it is in sequential SAT-solving. CDCL means *conflict-driven SAT*: when the current candidate model falsifies a clause, called a *conflict clause*, this conflict is *explained* by a heuristically controlled series of resolution steps, where every resolvent is also a conflict clause. A resolvent is *learned*, and the candidate partial model is repaired in such a way as to remove the conflict, by satisfying the learned clause and backjumping as far away as possible from the conflict.

Learning a conflict clause is a form of *lemmatization*, as every resolvent is a lemma, a logical consequence of the input set of clauses. All learned clauses are former conflict clauses. Similarly to other situations (cf. Section 2.1), a purpose of learning lemmas is to avoid repetitions: in CDCL it prevents the procedure from falling repeatedly into the same conflicts. In this sense, learning clauses is a way of pruning the search space.

The CDCL procedure involves several ingredients, in addition to conflict-driven clause learning and backjumping. *Activity-based decision heuristics* (e.g., VSIDS) select the literal for the next decision by counting how many times a literal appear in learned clauses and favoring the *most active* literals [226].

Clausal propagation consists of detecting *conflict clauses* and *implied literals*. A *conflict clause* is a clause whose literals are all *false* in the current candidate model. A literal is *implied* if it is the only unassigned literal of a clause: such a literal must be added to the trail in order to satisfy the clause, which is the *justification* of the implied literal. In the *two watched literals* scheme for clausal propagation [226, 124], it is sufficient to watch two non-*false* (i.e., either *true* or unassigned) literals per clause in order to detect conflict clauses and implied literals. Indeed, a conflict clause has zero non-*false* literals, and a justification has one non-*false* literal, so that a clause with two is neither a conflict clause nor a justification.

The possibility of periodically *restarting* the search with an empty trail and a set of clauses augmented with learned clauses may serve the purpose of compacting the trail or changing dynamically the order in which literals are picked for decision.

From the point of view of our analysis of parallelization of reasoning, *clause learning* is a key difference between DPLL and CDCL. Parallelizing DPLL can be seen as analogous to parallelizing tableau-based *subgoal-reduction strategies*: the database of clauses is *fixed*, equal to the input set, and the strategy searches for a model by exploring a tree that represents a survey of all possible interpretations. On the other hand, parallelizing CDCL can be seen as analogous to parallelizing *expansion-oriented strategies*, as the database of clauses *grows* due to *learning*. In CDCL learned clauses can be deleted based on heuristics (e.g., delete the oldest, or the least involved in resolution). These deletions can be considered a kind of forward contraction, while there is no analogy with backward contraction, since, for example, input clauses are not subject to deletion.

For CDCL, the definition of guiding path is updated to abandon the reference to the search space of a recursive DPLL procedure: a guiding path is simply a sequence of literals, and a node labeled L is *open* if L is a decided literal, *closed*, if L is an implied literal [185]. Also, the notion of guiding path is replaced by that of *cube* [115]. Logically speaking, a cube is a conjunction, or a set, of literals. In practice, cubes are typically much longer than guiding paths [115].

In keeping with the model-based character of the CDCL procedure, a cube can be understood as an *assignment* that assigns *true* to the literals in the cube. Then, the SAT problem is generalized to the *satisfiability modulo assignment* (SMA) problem, defined as the problem of deciding the satisfiability of S with respect to an assignment J to some of the literals in S. If J is empty, SMA reduces to SAT, while an intermediate state of a SAT search is an SMA instance, since during the search a SAT solver maintains a partial candidate model represented by an assignment of truth values to propositional variables. Approaches to parallel SAT-solving by distributed search such as PAMIRAXT [185] and *cube and conquer* [115] (see Chapter 2), attack a SAT problem with input set S, by having n processes $p_0, \ldots, p_{n-1}$ working in parallel on n SMA instances with input set S and initial assignments $J_0, \ldots J_{n-1}$, each containing a distinct cube.

Approaches to parallel SAT-solving by *multi-search* assign to the processes $p_0, \ldots, p_{n-1}$ different search plans, as in MANYSAT [112]. Similarly to ATP systems, also SAT solvers have many options and parameters that define the search plan and whose variation may serve the purpose of differentiating the searches. For example, the p_j's may employ different heuristics to pick the next literal for decision, or different heuristics to determine when to restart. Another way to differentiate the searches is to use *randomization* as in CL-SDSAT [122]: a randomized SAT solver makes a certain percentage of its decisions at random, starting from a given randomized seed, rather than based on a heuristic. Then, the p_j's may use different percentages or different seeds.

Activity-based decision heuristics and restart heuristics tend to *intensify* the search of a process, meaning that the process focuses on a certain region of the search space. In parallel search, this phenomenon may be useful to *reduce the overlap* between the processes, if each p_j focuses on a different region [111, 110].

In both distributed-search and multi-search parallel SAT-solving methods, the processes may *communicate learned clauses* [111, 112, 185, 122]. A learned clause φ is not sent to a process whose initial cube satisfies φ: indeed, in a model-based strategy a satisfied clause is *redundant* [61]. Upon receiving a learned clause, a process needs to determine its two watched literals for clausal propagation.

Since learned clauses are generated resolvents, communication of learned clauses in parallel SAT-solving reminds one of Clause-Diffusion (cf. Section 3.5.2). The possibility of applying heuristics to select for broadcasting only useful learned clauses is in the spirit of the Team-Work method (cf. Section 3.4.2). A typical heuristic is to broadcast learned clauses whose *size* is below a certain threshold. This is similar to what happens in Clause-Diffusion with *deletion by weight*: a clause whose weight is above the threshold gets deleted by forward contraction and therefore it is not broadcast. This kind of heuristic can be made dynamic by varying the threshold

during the search [111]. In propositional logic the size of a clause is the number of its literals. In a SAT solver the size of a clause is the number of its non-*false* literals with respect to the current candidate model. Thus, a clause may have different sizes under different cubes. Therefore, whether a learned clause is communicated depends on the given cube, as suggested in PMSAT [106].

Since a purpose of learning conflict clauses is to prune the search space, receiving from process p_k a learned conflict clause may help process p_j prune its search space. This is analogous to what happens in parallel search for ordering-based strategies, where receiving from process p_k a good simplifier may help process p_j prune its search space. On the other hand, communication is a cost in both contexts.

In parallel SAT-solving, the communication of learned clauses may be at odd with having low overlap or no overlap: if the processes delve into remote regions of the search space, sharing learned clauses may become useless [111]. In parallel search for theorem proving it is much harder to avoid overlapping searches, and therefore this issue does not arise. The observation of this phenomenon in parallel SAT-solving leads to the notion of subdividing the processes into groups [110]. Processes within a group cooperate, by sharing information such as learned clauses. Each group is devoted to search a different region of the search space, by letting all processes in the group start with the same cube, which is distinct from the cubes given to all other groups.

4.2 Parallelism and First-Order Model-Based Reasoning

Motivations for renewing the quest for parallel first-order theorem-proving methods are not different from those for injecting parallelism into SAT solvers: problems from applications get bigger and bigger; it is hard to improve sequential performance; and parallel hardware is available. In addition, the ATP problem is harder (only semi-decidable) and still far less understood than the SAT problem. Research into new approaches to ATP is certainly not over, and there are also approaches that are not new but are never or rarely considered for parallelization.

How to combine semantics and parallelism in theorem proving is still largely an open problem. *Semantically guided strategies* assume a *fixed* interpretation for semantic guidance. Among ordering-based strategies, a basic paradigm is that of *semantic resolution*, with *hyperresolution* and *resolution with set of support* as special cases (cf. Section 2.2). Among instance-based strategies, *ordered semantic hyperlinking* (OSHL) enriches hyperlinking with semantic guidance (cf. Section 2.3). A natural idea is to devise *multi-search* methods where the processes employ *different guiding interpretations* for semantic resolution or OSHL. A simple example is to have two parallel processes, one using positive and the other negative hyperresolution.

Another possibility is to design a method that combines distributed search as in Clause-Diffusion (see Section 3.5) with a multi-search scheme where the processes adopt different guiding interpretations. While Clause-Diffusion is a general

paradigm, it targets especially contraction-based strategies for equational theories and first-order logic with equality (cf. Section 2.2.2). Thus, the challenge is to combine multi-search with different guiding interpretations with distributed search for a logic including equality.

Model-based strategies build a candidate partial model and declare unsatisfiability when a contradiction arises, showing that no candidate can be completed in a model of the input set of clauses. Beside model elimination (ME) and ME-tableaux strategies (see Sections 2.1, 3.1.1, 3.2.1, 3.4.1), there are other classes of strategies that aim at being model-based for first-order logic and have not been considered for parallelization. This is the case for most model-oriented instance-based strategies and hybrid strategies that combine instance generation with tableaux (cf. Section 2.3 and Section 7.3 of [38]), as well as for the *model evolution calculus* that lifts the DPLL procedure to first-order logic [16, 17, 21, 22, 20].

Another example is the methods that integrate an ordering-based strategy for first-order logic with equality with a CDCL-based SAT [180] or SMT solver [56]. A straightforward approach to parallelization is to have two parallel processes, one executing the first-order strategy and one executing the solver. More ambitious schemes could devote multiple processes to both kinds of reasoning, parallelizing, in the sense of parallel search, both ordering-based strategy and solver. Such schemes could combine approaches to parallel search for SAT solvers (cf. Section 4.1, Chapter 1 on Parallel Satisfiability, and Chapter 2 on Cube and Conquer), SMT solvers (cf. Chapter 5 on Parallel Satisfiability Modulo Theories), and ordering-based first-order provers (cf. Sections 3.4.2 and 3.5).

SGGS (*Semantically-Guided Goal-Sensitive reasoning*) generalizes CDCL to first-order logic, and is both model-based and semantically guided [59, 58, 60, 61]. SGGS searches for a model of the input set S of clauses, starting from a given *initial Herbrand interpretation* I, and building interpretations $I[\Gamma_1]$, $I[\Gamma_2]$, $I[\Gamma_3], \ldots$, represented by *SGGS clause sequences* Γ_1, Γ_2, $\Gamma_3, \ldots$. An SGGS clause sequence is a sequence of constrained clauses with selected literals. An SGGS-derivation has the form $\Gamma_0 \vdash \Gamma_1 \vdash \Gamma_2 \vdash \Gamma_3 \vdash \ldots$, where Γ_0 is empty and $I[\Gamma_0] = I$. The current SGGS clause sequence corresponds to the current trail in CDCL. The main SGGS activities correspond to those of CDCL as follows.

The SGGS analogue of CDCL decision is *selection* of a literal in any clause added to the current SGGS clause sequence Γ. Selected literals differentiate $I[\Gamma]$ from Γ. SGGS is possibly the first method that features *clausal propagation at the first-order level*. Clausal propagation in SGGS relies on the concepts of *uniform falsity* and *dependence*. A literal is *uniformly false* in an interpretation if all its ground instances are *false* in that interpretation. For I, a literal is I-true if it is *true* in I, and I-false if it is uniformly false in I. SGGS requires that all literals in an SGGS clause sequence are either I-true or I-false. This invariant ensures that all ground instances of a literal in the sequence are in harmony with respect to I. A literal L *depends* on a selected literal M if M precedes L in Γ and all ground instances of L appear negated among the ground instances of M that M contributes to $I[\Gamma]$, so that M's selection makes L *uniformly false* in $I[\Gamma]$.

Most SGGS concepts and activities are defined *modulo semantic guidance* by I, because the system endeavors to make $I[\Gamma]$ different from I, since $I \not\models S$ (if $I \models S$, the problem is solved). For example, it is the I-false selected literals in Γ that differentiate $I[\Gamma]$ from I. Similarly, it is the dependence of I-true literals on I-false selected literals that is recorded by *assignment functions*; and it is I-all-true clauses, or clauses whose literals are all I-true, that are *conflict clauses* or *justifications* of an *implied literal*. When all literals of an I-all-true clause are assigned, it means that in an attempt to diversify $I[\Gamma]$ from I to satisfy other clauses, the system made that I-all-true clause uniformly false in $I[\Gamma]$. When all literals of an I-all-true clause but one are assigned, the non-assigned one must be selected, and it is an implied literal, as *all* its ground instances must be *true* in $I[\Gamma]$ to satisfy the clause.

The SGGS inference system includes *SGGS-extension*, *SGGS-splitting*, *SGGS-resolution*, *SGGS-move*, and *SGGS-deletion*. SGGS-extension is an *instance generation* mechanism. SGGS-extension extends the sequence Γ and the candidate model $I[\Gamma]$, by adding to Γ an instance of an input clause that covers ground instances not satisfied by $I[\Gamma]$. The clause is instantiated in a way that enforces the invariant whereby all literals in Γ are either I-true or I-false.

SGGS-splitting has nothing to do with DPLL splitting. SGGS-splitting of a clause φ by a clause ψ replaces φ with a *partition*, where all ground instances that a specified literal in φ has in common with ψ's selected literal are confined to one clause of the partition. This enables SGGS-resolution or SGGS-deletion to remove such intersections between literals, eliminating duplications or contradictions in the representation of the candidate model. *SGGS-resolution* is a restricted form of first-order resolution, where an implied literal in a justification resolves away a literal that depends on it: for this reason it uses *matching* rather than unification, and allows the resolvent to *replace* the parent that is not a justification. *SGGS-deletion* removes *disposable* clauses, which are redundant because satisfied by the interpretation induced by the clauses on their left in Γ. In a model-based approach a satisfied clause is redundant.

If SGGS-extension adds a clause in conflict with $I[\Gamma]$, the first-order CDCL mechanism of SGGS applies. It comprises *explanation* and *solving* inferences. If the conflict clause includes I-false literals, SGGS-resolution *explains* the conflict by resolving away those I-false literals with implied literals in Γ. An SGGS-extension adding such a clause makes sure that this is possible by applying an appropriate substitution. The explanation inferences yield either $\square$ or an I-all-true conflict clause, which is then subject to the solving inferences.

If the conflict clause does not include I-false literals, only the solving inferences are applied: the conflict clause is *moved* to the left of the clause to which its selected literal is assigned. This *SGGS-move* solves the conflict by *flipping* the truth value in $I[\Gamma]$ of *all* ground instances of this selected literal. It corresponds to backjumping in CDCL. The moved clause is *learned* in the sense that it becomes the justification of its selected literal. Prior to the move, splitting inferences may apply to make the selected literal of the clause to be moved so precise that the move will indeed flip the truth value of all its ground instances. Every SGGS-extension with a conflict clause is followed by the explanation and solving inference that solve the conflict.

Because of the novelty of SGGS, its parallelization is a research goal for the long term. Since SGGS is semantically guided by the initial interpretation I, the notion of a parallel search with multiple SGGS processes, each using a different I for semantic guidance, applies here too. Similarly to hyperresolution, the simplest example is to have two parallel SGGS processes, one using an I where all negative literals are *true*, and the other using an I where all positive literals are *true*. Most excitingly, SGGS opens the possibility of lifting to the first-order level the ideas for distributed search (e.g., cubes) or multi-search put forth for CDCL.

References

1. Martin Aigner, Armin Biere, Christoph M. Kirsch, Aina Niemetz, and Mathias Preiner. Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In Daniel Le Berre and Allen Van Gelder, editors, *Notes of the Fourth Workshop on Pragmatics of SAT (POS), Sixteenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 28–40, 2013.
2. Iliès Alouini. Concurrent garbage collector for concurrent rewriting. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications (RTA)*, volume 914 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 1995.
3. Iliès Alouini. *Étude et mise en oeuvre de la réecriture conditionnelle concurrente sur des machines parallèles à mémoire distribuée*. PhD thesis, Université Henri Poincaré Nancy 1, May 1997.
4. Siva Anantharaman and Nirina Andrianarivelo. Heuristical criteria in refutational theorem proving. In Alfonso Miola, editor, *Proceedings of the First International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO)*, volume 429 of *Lecture Notes in Computer Science*, pages 184–193. Springer, 1990.
5. Siva Anantharaman and Jieh Hsiang. Automated proofs of the Moufang identities in alternative rings. *Journal of Automated Reasoning*, 6(1):76–109, 1990.
6. Owen L. Astrachan and Donald W. Loveland. METEORS: high performance theorem provers using model elimination. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 31–60. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1991.
7. Owen L. Astrachan and Mark E. Stickel. Caching and lemmaizing in model elimination theorem provers. In Deepak Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 224–238. Springer, 1992.
8. Jürgen Avenhaus and Jörg Denzinger. Distributing equational theorem proving. In Claude Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications (RTA)*, volume 690 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 1993.
9. Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: a system for distributed equational deduction. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications (RTA)*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402. Springer, 1995.
10. Leo Bachmair and Nachum Dershowitz. Critical pair criteria for completion. *Journal of Symbolic Computation*, 6(1):1–18, 1988.
11. Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure. In Hassam Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume II: Rewriting Techniques, pages 1–30. Academic Press, Cambridge, England, 1989.

12. Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
13. Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.
14. Leo Bachmair, Harald Ganzinger, David McAllester, and Christopher A. Lynch. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 535–610. Elsevier, Amsterdam, The Netherlands, 2001.
15. Peter Baumgartner. Hyper tableaux – the next generation. In Harrie de Swart, editor, *Proceedings of the Seventh International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 60–76. Springer, 1998.
16. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the model evolution calculus. *International Journal on Artificial Intelligence Tools*, 15(1):21–52, 2006.
17. Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the Thirteenth Conference on Logic, Programming and Automated Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Artificial Intelligence*, pages 572–586. Springer, 2006.
18. Peter Baumgartner and Ulrich Furbach. Consolution as a framework for comparing calculi. *Journal of Symbolic Computation*, 16(5):445–477, 1993.
19. Peter Baumgartner and Ulrich Furbach. Variants of clausal tableaux. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, volume I: Foundations - Calculi and Methods, chapter 3, pages 73–102. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1998.
20. Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model evolution calculus with equality - revised and implemented. *Journal of Symbolic Computation*, 47(9):1011–1045, 2012.
21. Peter Baumgartner and Cesare Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172(4/5):591–632, 2008.
22. Peter Baumgartner and Uwe Waldmann. Superposition and model evolution combined. In Renate Schmidt, editor, *Proceedings of the Twenty-Second International Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 17–34. Springer, 2009.
23. Markus Bender, Björn Pelzer, and Claudia Schon. E-KRHyper 1.4: extensions for unique names and description logic. In Maria Paola Bonacina, editor, *Proceedings of the Twenty-Fourth International Conference on Automated Deduction (CADE)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 126–134. Springer, 2013.
24. Wolfgang Bibel and Elmer Eder. Methods and calculi for deduction. In Dov M. Gabbay, Christopher J. Hogger, and John Alan Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume I: Logical Foundations, pages 68–183. Oxford University Press, Oxford, England, 1993.
25. Jean-Paul Billon. The disconnection method. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Proceedings of the Fifth International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 110–126. Springer, 1996.
26. Maria Paola Bonacina. *Distributed automated deduction*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.
27. Maria Paola Bonacina. On the reconstruction of proofs in distributed theorem proving with contraction: a modified Clause-Diffusion method. In Hoon Hong, editor, *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO)*, volume 5 of *Lecture Notes Series in Computing*, pages 22–33. World Scientific, 1994.
28. Maria Paola Bonacina. On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method. *Journal of Symbolic Computation*, 21(4–6):507–522, 1996.

29. Maria Paola Bonacina. The Clause-Diffusion theorem prover Peers-mcd. In William W. McCune, editor, *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 53–56. Springer, 1997.
30. Maria Paola Bonacina. Experiments with subdivision of search in distributed theorem proving. In Markus Hitz and Erich Kaltofen, editors, *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO)*, pages 88–100. ACM Press, 1997.
31. Maria Paola Bonacina. Analysis of distributed-search contraction-based strategies. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *Proceedings of the Sixth European Workshop on Logics in Artificial Intelligence (JELIA)*, volume 1489 of *Lecture Notes in Artificial Intelligence*, pages 107–121. Springer, 1998.
32. Maria Paola Bonacina. Mechanical proofs of the Levi commutator problem. In Peter Baumgartner et al., editor, *Notes of the Workshop on Problem Solving Methodologies with Automated Deduction, Fifteenth International Conference on Automated Deduction (CADE)*, pages 1–10, 1998.
33. Maria Paola Bonacina. A model and a first analysis of distributed-search contraction-based strategies. *Annals of Mathematics and Artificial Intelligence*, 27(1–4):149–199, 1999.
34. Maria Paola Bonacina. A taxonomy of theorem-proving strategies. In Michael J. Wooldridge and Manuela Veloso, editors, *Artificial Intelligence Today - Recent Trends and Developments*, volume 1600 of *Lecture Notes in Artificial Intelligence*, pages 43–84. Springer, Berlin, Germany, 1999.
35. Maria Paola Bonacina. Ten years of parallel theorem proving: a perspective. In Bernhard Gramlich, Hélène Kirchner, and Frank Pfenning, editors, *Notes of the Third Workshop on Strategies in Automated Deduction (STRATEGIES), Second Federated Logic Conference (FLoC)*, pages 3–15, 1999.
36. Maria Paola Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257, 2000.
37. Maria Paola Bonacina. Combination of distributed search and multi-search in Peers-mcd.d. In Rajeev P. Gore, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 448–452. Springer, 2001.
38. Maria Paola Bonacina. Towards a unified model of search in theorem proving: subgoal-reduction strategies. *Journal of Symbolic Computation*, 39(2):209–255, 2005.
39. Maria Paola Bonacina. On theorem proving for program checking – Historical perspective and recent developments. In Maribel Fernàndez, editor, *Proceedings of the Twelfth International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 1–11. ACM Press, 2010.
40. Maria Paola Bonacina and Nachum Dershowitz. Abstract canonical inference. *ACM Transactions on Computational Logic*, 8(1):180–208, 2007.
41. Maria Paola Bonacina and Nachum Dershowitz. Canonical ground Horn theories. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Artificial Intelligence*, pages 35–71. Springer, 2013.
42. Maria Paola Bonacina and Mnacho Echenim. Theory decision by decomposition. *Journal of Symbolic Computation*, 45(2):229–260, 2010.
43. Maria Paola Bonacina, Ulrich Furbach, and Viorica Sofronie-Stokkermans. On first-order model-based reasoning. In Narciso Martí-Oliet, Peter Olveczky, and Carolyn Talcott, editors, *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer*, volume 9200 of *Lecture Notes in Computer Science*, pages 181–204. Springer, Berlin, Germany, 2015.
44. Maria Paola Bonacina and Jieh Hsiang. High performance simplification-based automated deduction. In *Transactions of the Ninth U.S. Army Conference on Applied Mathematics and Computing*, number 92-1, pages 321–335. Army Research Office, 1991.
45. Maria Paola Bonacina and Jieh Hsiang. A system for distributed simplification-based theorem proving. In Bertrand Fronhöfer and Graham Wrightson, editors, *Proceedings of the First*

International Workshop on Parallelization in Inference Systems (December 1990), volume 590 of *Lecture Notes in Artificial Intelligence*, pages 370–370. Springer, Berlin, Germany, 1992.

46. Maria Paola Bonacina and Jieh Hsiang. Distributed deduction by Clause-Diffusion: the Aquarius prover. In Alfonso Miola, editor, *Proceedings of the Third International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO)*, volume 722 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 1993.
47. Maria Paola Bonacina and Jieh Hsiang. On fairness in distributed deduction. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *Proceedings of the Tenth Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 665 of *Lecture Notes in Computer Science*, pages 141–152. Springer, 1993.
48. Maria Paola Bonacina and Jieh Hsiang. On subsumption in distributed derivations. *Journal of Automated Reasoning*, 12:225–240, 1994.
49. Maria Paola Bonacina and Jieh Hsiang. Parallelization of deduction strategies: an analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.
50. Maria Paola Bonacina and Jieh Hsiang. The Clause-Diffusion methodology for distributed deduction. *Fundamenta Informaticae*, 24(1–2):177–207, 1995.
51. Maria Paola Bonacina and Jieh Hsiang. Distributed deduction by Clause-Diffusion: distributed contraction and the Aquarius prover. *Journal of Symbolic Computation*, 19:245–267, 1995.
52. Maria Paola Bonacina and Jieh Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146:199–242, 1995.
53. Maria Paola Bonacina and Jieh Hsiang. On semantic resolution with lemmaizing and contraction and a formal treatment of caching. *New Generation Computing*, 16(2):163–200, 1998.
54. Maria Paola Bonacina and Jieh Hsiang. On the modelling of search in theorem proving – towards a theory of strategy analysis. *Information and Computation*, 147:171–208, 1998.
55. Maria Paola Bonacina, Christopher A. Lynch, and Leonardo de Moura. On deciding satisfiability by DPLL($\Gamma + \mathscr{T}$) and unsound theorem proving. In Renate Schmidt, editor, *Proceedings of the Twenty-second International Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 35–50. Springer, 2009.
56. Maria Paola Bonacina, Christopher A. Lynch, and Leonardo de Moura. On deciding satisfiability by theorem proving with speculative inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.
57. Maria Paola Bonacina and William W. McCune. Distributed theorem proving by Peers. In Alan Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction (CADE)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 841–845. Springer, 1994.
58. Maria Paola Bonacina and David A. Plaisted. Constraint manipulation in SGGS. In Temur Kutsia and Christophe Ringeissen, editors, *Proceedings of the Twenty-Eighth Workshop on Unification (UNIF), Sixth Federated Logic Conference (FLoC)*, Technical Reports of the Research Institute for Symbolic Computation, pages 47–54. Johannes Kepler Universität, 2014.
59. Maria Paola Bonacina and David A. Plaisted. SGGS theorem proving: an exposition. In Stephan Schulz, Leonardo De Moura, and Boris Konev, editors, *Proceedings of the Fourth Workshop on Practical Aspects in Automated Reasoning (PAAR), Sixth Federated Logic Conference (FLoC), July 2014*, volume 31 of *EasyChair Proceedings in Computing (EPiC)*, pages 25–38, 2015.
60. Maria Paola Bonacina and David A. Plaisted. Semantically-guided goal-sensitive reasoning: model representation. *Journal of Automated Reasoning*, 56(2):113–141, 2016.
61. Maria Paola Bonacina and David A. Plaisted. Semantically-guided goal-sensitive reasoning: inference system and completeness. *Journal of Automated Reasoning*, 59:165–218, 2017.
62. Soumitra Bose, Edmund M. Clarke, David E. Long, and Spiro Michaylov. Parthenon: A parallel theorem prover for non-Horn clauses. *Journal of Automated Reasoning*, 8(2):153–182, 1992.

63. Bruno Buchberger. *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal (in German)*. PhD thesis, Department of Mathematics, Universität Innsbruck, 1965.
64. Bruno Buchberger. History and basic features of the critical-pair/completion procedure. *Journal of Symbolic Computation*, 3:3–38, 1987.
65. Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. Strategy-compliant multi-threaded term completion. *Journal of Symbolic Computation*, 21(4–6):475–506, 1996.
66. Ralph M. Butler and Ewing L. Lusk. User's guide to the p4 programming system. Technical Report 92/17, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, October 1992.
67. Soumen Chakrabarti and Katherine A. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 169–178, 1993.
68. Soumen Chakrabarti and Katherine A. Yelick. On the correctness of a distributed memory Gröbner basis algorithm. In Claude Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications (RTA)*, volume 690 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 1993.
69. K. Many Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett, Burlington, Massachusetts, 1991.
70. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Cambridge, England, 1973.
71. P. Daniel Cheng and J. Y. Juang. A parallel resolution procedure based on connection graph. In *Proceedings of the Sixth Annual Conference of the American Association for Artificial Intelligence (AAAI)*, pages 13–17, 1987.
72. Heng Chu and David A. Plaisted. Model finding in semantically guided instance-based theorem proving. *Fundamenta Informaticae*, 21(3):221–235, 1994.
73. Heng Chu and David A. Plaisted. CLINS-S: a semantically guided first-order theorem prover. *Journal of Automated Reasoning*, 18(2):183–188, 1997.
74. Edmund M. Clarke, David E. Long, Spiro Michaylov, Stephen A. Schwab, Jean-Philippe Vidal, and Shinji Kimura. Parallel symbolic computation algorithms. Technical Report CMU-CS-90-182, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1990.
75. Susan E. Conry, Douglas J. MacIntosh, and Robert A. Meyer. DARES: a Distributed Automated REasoning System. In *Proceedings of the Eleventh Annual Conference of the American Association for Artificial Intelligence (AAAI)*, pages 78–85, 1990.
76. Simon Cruanes. *Extending superposition with integer arithmetic, structural induction, and beyond*. PhD thesis, École Polytechnique, Université Paris-Saclay, September 2015.
77. Bernd I. Dahn. Robbins algebras are Boolean: a revision of McCune's computer-generated solution of Robbins problem. *Journal of Algebra*, 208:526–532, 1998.
78. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
79. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
80. Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings of the Fourth International Conference on Automated Reasoning (IJCAR)*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 475–490. Springer, 2008.
81. Leonardo de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In Jürgen Giesl and Reiner Hähnle, editors, *Proceedings of the Fifth International Conference on Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 400–411. Springer, 2010.
82. Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.
83. Jörg Denzinger. *Team-Work: a method to design distributed knowledge based theorem provers*. PhD thesis, Department of Computer Science, Universität Kaiserslautern, 1993.

84. Jörg Denzinger and Bernd Ingo Dahn. Cooperating theorem provers. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation, chapter 14, pages 383–416. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1998.
85. Jörg Denzinger and Dirk Fuchs. Cooperation of heterogeneous provers. In Thomas Dean, editor, *Proceedings of the Sixeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10–15. Morgan Kaufmann Publishers, 1999.
86. Jörg Denzinger, Marc Fuchs, and Matthias Fuchs. High performance ATP systems by combining several AI methods. In Martha E. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 102–107. Morgan Kaufmann Publishers, 1997.
87. Jörg Denzinger and Matthias Fuchs. Goal-oriented equational theorem proving using Team-Work. In Bernhard Nebel and Leonie Dreschler-Fischer, editors, *Proceedings of the Eighteenth German Conference on Artificial Intelligence (KI)*, volume 861 of *Lecture Notes in Artificial Intelligence*, pages 343–354. Springer, 1994.
88. Jörg Denzinger and Martin Kronenburg. Planning for distributed theorem proving: the Team-Work approach. In Steffen Hölldobler, editor, *Proceedings of the Twentieth German Conference on Artificial Intelligence (KI)*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 43–56. Springer, 1996.
89. Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT: a distributed and learning equational prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.
90. Jörg Denzinger and Jürgen Lind. TWlib: a library for distributed search applications. In Chu-Sing Yang, editor, *Proceedings of the International Conference on Artificial Intelligence, International Computer Symposium (ICS)*, pages 101–108. National Sun-Yat Sen University, 1996.
91. Jörg Denzinger and Stephan Schulz. Recording and analyzing knowledge-based distributed deduction processes. *Journal of Symbolic Computation*, 21(4–6):523–541, 1996.
92. Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
93. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, Amsterdam, The Netherlands, 1990.
94. Nachum Dershowitz and Naomi Lindenstrauss. An abstract concurrent machine for rewriting. In Hélène Kirchner and W. Wechler, editors, *Proceedings of the Second International Conference on Algebraic and Logic Programming (ALP)*, volume 463 of *Lecture Notes in Computer Science*, pages 318–331. Springer, 1990.
95. Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465476, 1979.
96. Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier, Amsterdam, The Netherlands, 2001.
97. Norbert Eisinger and Hans Jürgen Ohlbach. Deduction systems based on resolution. In Dov M. Gabbay, Christopher J. Hogger, and John Alan Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume I: Logical Foundations, pages 184–273. Oxford University Press, Oxford, England, 1993.
98. Zachary Ernst and Seth Kurtenbach. Toward a procedure for data mining proofs. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Artificial Intelligence*, pages 229–239. Springer, 2013.
99. Michael Fisher. An alternative approach to concurrent theorem proving. In James Geller, Hiroaki Kitano, and Christian B. Suttner, editors, *Parallel Processing for Artificial Intelligence 3*, pages 209–230. Elsevier, Amsterdam, The Netherlands, 1997.
100. Ian Foster and Steve Tuecke. Parallel programming with PCN. Technical Report 91/32, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, December 1991.

101. Dirk Fuchs. Requirement-based cooperative theorem proving. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *Proceedings of the Sixth Joint European Workshop on Logic in Artificial Intelligence (JELIA)*, volume 1489 of *Lecture Notes in Artificial Intelligence*, pages 139–153. Springer, 1998.
102. Marc Fuchs. Controlled use of clausal lemmas in connection tableau calculi. *Journal of Symbolic Computation*, 29(2):299–341, 2000.
103. Marc Fuchs and Andreas Wolf. Cooperation in model elimination: CPTHEO. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the Fifteenth International Conference on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 42–46. Springer, 1998.
104. Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *Proceedings of the Eighteenth IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–64. IEEE Computer Society Press, 2003.
105. Harald Ganzinger and Konstantin Korovin. Theory instantiation. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the Thirteenth Conference on Logic, Programming and Automated Reasoning (LPAR)*, volume 4246 of *Lecture Notes in Artificial Intelligence*, pages 497–511. Springer, 2006.
106. Luís Gil, Paulo F. Flores, and Luis Miguel Silveira. PMSat: a parallel version of Minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.
107. Joseph A. Goguen, Sany Leinwand, José Meseguer, and Timothy Winkler. The rewrite rule machine 1988. Technical Report PRG-76, Oxford University Computing Laboratory, Oxford, England, August 1989.
108. Joseph A. Goguen, José Meseguer, Sany Leinwand, Timothy Winkler, and Hitoshi Aida. The rewrite rule machine. Technical Report SRI-CSL-89-6, Computer Science Laboratory, SRI International, Menlo Park, California, March 1989.
109. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.
110. Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In Dave Cohen, editor, *Proceedings of the Sixteenth International Conference on Principles and Practice of Constraint Programming (CP)*, volume 6308 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2010.
111. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In Craig Boutilier, editor, *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 409–504. AAAI Press, 2009.
112. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.
113. Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34(2):99–106, 2013.
114. D. J. Hawley. A Buchberger algorithm for distributed memory multi-processors. In Hans P. Zima, editor, *Proceedings of the First International Conference of the Austrian Center for Parallel Computation (ACPC)*, volume 591 of *Lecture Notes in Computer Science*. Springer, 1991.
115. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn M. Shehory, editors, *Proceedings of the Seventh International Haifa Verification Conference (HVC)*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2012.
116. Thomas Hillenbrand. Citius, altius, fortius: lessons learned from the theorem prover WALDMEISTER. In Ingo Dahn and Laurent Vigneron, editors, *Proceedings of the Fourth International Workshop On First-Order Theorem Proving (FTP)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
117. Christoph M. Hoffmann and Michael J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.
118. Alfred Horn. On sentences which are true in direct unions of algebras. *Journal of Symbolic Logic*, 16:14–21, 1951.

119. Jieh Hsiang and Michaël Rusinowitch. On word problems in equational theories. In Thomas Ottman, editor, *Proceedings of the Fourteenth International Colloquium on Automta, Languages, and Programming (ICALP)*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 1987.
120. Jieh Hsiang and Michaël Rusinowitch. Proving refutational completeness of theorem proving strategies: the transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, 1991.
121. Jieh Hsiang, Michaël Rusinowitch, and Ko Sakai. Complete inference rules for the cancellation laws. In John McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 990–992. Morgan Kaufmann Publishers, 1987.
122. Antti E. J. Hyvärinen, Tommi Junttila, and Ilka Niemelä. Incorporating clause learning in grid-based randomized SAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:223–244, 2009.
123. Swen Jacobs and Uwe Waldmann. Comparing instance generation methods for automated reasoning. *Journal of Automated Reasoning*, 38:57–78, 2007.
124. Himanshu Jain. *Verification using satisfiability checking, predicate abstraction and Craig interpolation*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 2008.
125. Anita Jindal, Ross Overbeek, and Waldo C. Kabat. Exploitation of parallel processing for implementing high-performance deduction systems. *Journal of Automated Reasoning*, 8:23–38, 1992.
126. Deepak Kapur, David Musser, and Paliath Narendran. Only prime superposition need be considered in the Knuth-Bendix completion procedure. *Journal of Symbolic Computation*, 6:19–36, 1988.
127. Owen Kaser, Shaunak Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Fast parallel implementations of lazy languages – the EQUALS experience. In John L. White, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 335–344. ACM Press, 1992.
128. Claude Kirchner, Christopher Lynch, and Christelle Scharff. Fine-grained concurrent completion. In Harald Ganzinger, editor, *Proceedings of the Seventh International Conference on Rewriting Techniques and Applications (RTA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 1996.
129. Claude Kirchner and Patrick Viry. Implementing parallel rewriting. In Bertrand Fronhöfer and Graham Wrightson, editors, *Proceedings of the First International Workshop on Parallelization in Inference Systems (December 1990)*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 123–138. Springer, Berlin, Germany, 1992.
130. Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Proceedings of the Conference on Computational Problems in Abstract Algebras*, pages 263–298. Pergamon Press, Oxford, England, 1970.
131. Richard E. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
132. Konstantin Korovin. An invitation to instantiation-based reasoning: from theory to practice. In Renate Schmidt, editor, *Proceedings of the Twenty-Second International Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 163–166. Springer, 2009.
133. Konstantin Korovin. Inst-Gen: a modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Artificial Intelligence*, pages 239–270. Springer, 2013.
134. Konstantin Korovin and Christoph Sticksel. iProver-Eq: An instantiation-based theorem prover with equality. In Jürgen Giesl and Reiner Hähnle, editors, *Proceedings of the Fifth International Conference on Automated Reasoning (IJCAR)*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 196–202. Springer, 2010.
135. Laura Kovàcs and Andrei Voronkov. First order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of the Twenty-Fifth International Con-*

ference on Computer-Aided Verification (CAV), volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.
136. Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.
137. Dallas S. Lankford and A. M. Ballantyne. The refutation completeness of blocked permutative narrowing and resolution. In William H. Joyner Jr., editor, *Proceedings of the Fourth Conference on Automated Deduction (CADE)*, pages 168–174, 1979. Available at `http://www.cadeinc.org/`.
138. Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyperlinking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.
139. K. Rustan M. Leino and Aleksandar Milicevic. Program extrapolation with Jennisys. In *Proceedings of the Twenty-Seventh Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 411–430. ACM, 2012.
140. Reinhold Letz. Clausal tableaux. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, volume I: Foundations - Calculi and Methods, chapter 2, pages 43–72. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1998.
141. Reinhold Letz, Klaus Mayr, and Christian Goller. Controlled integration of the cut rule into connection tableau calculi. *Journal of Automated Reasoning*, 13(3):297–338, 1994.
142. Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. SETHEO: a high performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.
143. Reinhold Letz and Gernot Stenz. DCTP - a disconnection calculus theorem prover. In Rajeev P. Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 381–385. Springer, 2001.
144. Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 28, pages 2015–2114. Elsevier, Amsterdam, The Netherlands, 2001.
145. Reinhold Letz and Gernot Stenz. Proof and model generation with disconnection tableaux. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the Eighth International Conference on Logic, Programming and Automated Reasoning (LPAR)*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 142–156. Springer, 2001.
146. Reinhold Letz and Gernot Stenz. Integration of equality reasoning into the disconnection calculus. In Uwe Egly and Christian G. Fermüller, editors, *Proceedings of the Fifteenth International Conference on Analytic Tableaux and Related Methods (TABLEAUX)*, volume 2381 of *Lecture Notes in Artificial Intelligence*, pages 176–190. Springer, 2002.
147. Vladimir Lifschitz, Leora Morgenstern, and David A. Plaisted. Knowledge representation and classical logic. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 1, pages 3–88. Elsevier, Amsterdam, The Netherlands, 2008.
148. Rasiah Loganantharaj. *Theoretical and implementational aspects of parallel link resolution in connection graphs*. PhD thesis, Department of Computer Science, Colorado State University, 1985.
149. Rasiah Loganantharaj and Robert A. Müller. Parallel theorem proving with connection graphs. In Jörg Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction (CADE)*, volume 230 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 1986.
150. Donald W. Loveland. A simplified format for the model elimination procedure. *Journal of the ACM*, 16(3):349–363, 1969.
151. Donald W. Loveland. A unifying view of some linear Herbrand procedures. *Journal of the ACM*, 19(2):366–384, 1972.
152. Ewing L. Lusk and William W. McCune. Experiments with ROO: a parallel automated deduction system. In Bertrand Fronhöfer and Graham Wrightson, editors, *Proceedings of the First International Workshop on Parallelization in Inference Systems (December 1990)*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 139–162. Springer, Berlin, Germany, 1992.

153. Ewing L. Lusk, William W. McCune, and John K. Slaney. Parallel closure-based automated reasoning. In Bertrand Fronhöfer and Graham Wrightson, editors, *Proceedings of the First International Workshop on Parallelization in Inference Systems (December 1990)*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 347–347. Springer, Berlin, Germany, 1992.
154. Ewing L. Lusk, William W. McCune, and John K. Slaney. ROO: a parallel theorem prover. In Deepak Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 731–734. Springer, 1992.
155. Sharad Malik and Lintao Zhang. Boolean satisfiability: from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.
156. Norbert Manthey. Towards next generation sequential and parallel SAT solvers. *Constraints*, 20(4):504–505, 2015.
157. Rainer Manthey and François Bry. SATCHMO: a theorem prover implemented in Prolog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, pages 415–434. Springer, 1988.
158. João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marjin Heule, Hans Van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press, Amsterdam, The Netherlands, 2009.
159. João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, 1997.
160. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
161. Ruben Martins, Vasco M. Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.
162. William W. McCune. OTTER 2.0 users guide. Technical Report 90/9, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, March 1990.
163. William W. McCune. What's new in OTTER 2.2. Technical Report TM-153, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, July 1991.
164. William W. McCune. OTTER 3.0 reference manual and guide. Technical Report 94/6, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, January 1994. Revised August 1995.
165. William W. McCune. 33 Basic test problems: a practical evaluation of some paramodulation strategies. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 71–114. MIT Press, Cambridge, Massachusetts, 1997.
166. William W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.
167. William W. McCune. OTTER 3.3 reference manual. Technical Report TM-263, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, August 2003.
168. William W. McCune. PROVER9 and MACE4, 2005–2010. `http://www.cs.unm.edu/~mccune/prover9/`, last seen on July 3, 2017.
169. Max Moser, Ortrun Ibens, Reinhold Letz, Joachim Steinbach, Christoph Goller, Johann Schumann, and Klaus Mayr. The model elimination provers SETHEO and E-SETHEO. *Journal of Automated Reasoning*, 18(2):237–246, 1997.
170. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In David Blaauw and Luciano Lavagno, editors, *Proceedings of the Thirty-Ninth Design Automation Conference (DAC)*, pages 530–535, 2001.
171. Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 7, pages 371–443. Elsevier, Amsterdam, The Netherlands, 2001.

172. Robert Niewenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.
173. Gerald E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM Journal of Computing*, 12(1):82–100, 1983.
174. Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.
175. David A. Plaisted. Mechanical theorem proving. In Ranan B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pages 269–320. Elsevier, Amsterdam, The Netherlands, 1990.
176. David A. Plaisted. Equational reasoning and term rewriting systems. In Dov M. Gabbay, Christopher J. Hogger, and John Alan Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume I: Logical Foundations, pages 273–364. Oxford University Press, Oxford, England, 1993.
177. David A. Plaisted. Automated theorem proving. *Wiley Interdisciplinary Reviews: Cognitive Science*, 5(2):115–128, 2014.
178. David A. Plaisted and Swaha Miller. The relative power of semantics and unification. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Artificial Intelligence*, pages 317–344. Springer, 2013.
179. David A. Plaisted and Yunshan Zhu. Ordered semantic hyper linking. *Journal of Automated Reasoning*, 25:167–217, 2000.
180. Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In Amy P. Felty and Aart Middeldorp, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Deduction (CADE)*, volume 9195 of *Lecture Notes in Artificial Intelligence*, pages 399–415. Springer, 2015.
181. George A. Robinson and Larry Wos. Paramodulation and theorem-proving in first-order theories with equality. In Donald Michie and Bernard Meltzer, editors, *Machine Intelligence*, volume 4, pages 135–150. Edinburgh University Press, Edinburgh, Scotland, 1969.
182. John Alan Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.
183. John Alan Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
184. Michaël Rusinowitch. Theorem-proving with resolution and superposition. *Journal of Symbolic Computation*, 11(1 & 2):21–50, 1991.
185. Tobias Schubert, Matthew Lewis, and Bernd Becker. PaMiraXT: parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.
186. Stephan Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2–3):111–126, 2002.
187. Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Artificial Intelligence*, pages 45–67. Springer, 2013.
188. Stephan Schulz. System description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proceedings of the Nineteenth International Conference on Logic, Programming and Automated Reasoning (LPAR)*, volume 8312 of *Lecture Notes in Artificial Intelligence*, pages 735–743. Springer, 2013.
189. Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Proceedings of the Eighth International Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Artificial Intelligence*, pages 330–345. Springer, 2016.
190. Johan Schumann. Parallel theorem provers – an overview. In Bertrand Fronhöfer and Graham Wrightson, editors, *Proceedings of the First International Workshop on Parallelization in Inference Systems (December 1990)*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 26–50. Springer, Berlin, Germany, 1992.

191. Johann Schumann. Delta: a bottom-up pre-processor for top-down theorem provers. In Alan Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction (CADE)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 774–777. Springer, 1994.
192. Johann Schumann and Reinhold Letz. PARTHEO: a high-performance parallel theorem prover. In Mark E. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (CADE)*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 28–39. Springer, 1990.
193. Robert E. Shostak. Refutation graphs. *Artificial Intelligence*, 7:51–64, 1976.
194. Kurt Siegl. Gröbner bases computation in STRAND: a case study for concurrent symbolic computation in logic programming languages (Master thesis). Technical Report 90-54.0, Research Institute for Symbolic Computation (RISC), Linz, Austria, November 1990.
195. Carsten Sinz, Jörg Denzinger, Jürgen Avenhaus, and Wolfgang Küchlin. Combining parallel and distributed search in automated equational deduction. In *Proceedings of the Fourth International Conference on Parallel Processing and Applied Mathematics (PPAM) – Revised Papers*, pages 819–832, 2001.
196. James R. Slagle. Automatic theorem proving with renamable and semantic resolution. *Journal of the ACM*, 14(4):687–697, 1967.
197. James R. Slagle. Automated theorem proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21:622–642, 1974.
198. Mark E. Stickel. A Prolog technology theorem prover. *New Generation Computing*, 2(4):371–383, 1984.
199. Mark E. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.
200. Mark E. Stickel. PTTP and linked inference. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 283–296. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1991.
201. Mark E. Stickel. A Prolog technology theorem prover: new exposition and implementation in Prolog. *Theoretical Computer Science*, 104:109–128, 1992.
202. Mark E. Stickel and W. Mabry Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1073–1075. Morgan Kaufmann Publishers, 1985.
203. David Sturgill and Alberto Maria Segre. Nagging: a distributed, adversarial search-pruning technique applied to first-order inference. *Journal of Automated Reasoning*, 19(3):347–376, 1997.
204. Geoff Sutcliffe. A heterogeneous parallel deduction system. In Ryuzo Hasegawa and Mark E. Stickel, editors, *Proceedings of the FGCS Workshop on Automated Deduction: Logic Programming and Parallel Computing Approaches*, pages 5–13, 1992.
205. Christian B. Suttner. SPTHEO: a parallel theorem prover. *Journal of Automated Reasoning*, 18(2):253–258, 1997.
206. Christian B. Suttner and Johann Schumann. Parallel automated theorem proving. In Laveen N. Kanal, Vipin Kumar, Hiroaki Kitano, and Christian B. Suttner, editors, *Parallel Processing for Artificial Intelligence*. Elsevier, Amsterdam, The Netherlands, 1994.
207. Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
208. Stephen Taylor. *Parallel Logic Programming Techniques*. Prentice Hall, Upper Saddle River, New Jersey, 1989.
209. Josef Urban and Jirí Vyskocil. Theorem proving in large formal mathematics as an emerging AI field. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Artificial Intelligence*, pages 240–257. Springer, 2013.
210. Jean-Philippe Vidal. The computation of Gröbner bases on a shared memory multiprocessor. In Alfonso Miola, editor, *Proceedings of the First International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO)*, volume 429 of *Lecture Notes in Computer Science*, pages 81–90. Springer, 1990.

211. Kevin Wallace and Graham Wrightson. Regressive merging in model elimination tableau-based theorem provers. *Journal of the IGPL*, 3(6):921–937, 1995.
212. David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.
213. David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):94–111, 1992.
214. Christoph Weidenbach, Dylana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate Schmidt, editor, *Proceedings of the Twenty-Second International Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer, 2009.
215. Andreas Wolf. P-SETHEO: strategy parallelism in automated theorem proving. In Harrie de Swart, editor, *Proceedings of the Seventh International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 320–324. Springer, 1998.
216. Larry Wos. Searching for open questions. *Newsletter of the Association for Automated Reasoning*, 15, May 1990.
217. Larry Wos, Daniel F. Carson, and George A. Robinson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12:536–541, 1965.
218. Larry Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4):698–709, 1967.
219. Chih-Hung Wu and Shie-Jue Lee. Parallelization of a hyper-linking based theorem prover. *Journal of Automated Reasoning*, 26(1):67–106, 2001.
220. Katherine A. Yelick. *Using abstraction in explicitly parallel programs*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.
221. Katherine A. Yelick and Steven J. Garland. A parallel completion procedure for term rewriting systems. In Deepak Kapur, editor, *Proceedings of the Eleventh International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 109–123. Springer, 1992.
222. Hantao Zhang. SATO: an efficient propositional prover. In William W. McCune, editor, *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275. Springer, 1997.
223. Hantao Zhang and Maria Paola Bonacina. Cumulating search in a distributed computing environment: a case study in parallel satisfiability. In Hoon Hong, editor, *Proceedings of the First International Symposium on Parallel Symbolic Computation (PASCO)*, volume 5 of *Lecture Notes Series in Computing*, pages 422–431. World Scientific, 1994.
224. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4–6):543–560, 1996.
225. Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.
226. Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the Eighteenth International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 295–313. Springer, 2002.

Index

Aquarius, 1, 3, 25, 30
assignment, 2, 36
assignment function, 39
associative-commutative symbol, 30, 31

backjumping, 35, 39
backtrack, 6, 7, 15
backward contraction, 8–10, 12, 17–19, 26–31, 35
backward-contraction bottleneck, 18, 19, 30
best-first search, 7, 16, 32
blocking, 31
breadth-first search, 32
Buchberger algorithm, 9, 18

C-reduction, 5
caching, 5, 6
CDCL, 35, 36, 38–40
CL-SDSAT, 36
clausal simplification, 7, 8, 11, 28
clause learning, 35
Clause-Diffusion, 1, 3, 13, 25–37
completion procedure, 9, 15, 18, 30
concurrent rewriting, 14
conflict clause, 35, 37, 39
contraction-based strategies, 3, 8, 9, 12, 24, 38
cooperative parallelism, 20, 22, 23
CPTHEO, 21, 22, 24
cube, 36, 37, 40

DFS, 6, 21
DFS with iterative deepening, 6, 21
distributed fairness, 29, 30
distributed global contraction, 29
distributed proof reconstruction, 29
distributed search, 13, 20, 24, 25, 29–34, 36, 38, 40
divide and conquer, 1, 20, 34
DPLL, 11, 34–36, 38

EQP, 31, 32
expansion-oriented strategies, 3, 8–10, 12, 35
explanation, 35, 39

factoring, 7, 27, 28
fairness, 11, 23, 29
folding-up, 5, 6
forward contraction, 8, 10, 11, 16, 17, 19, 26, 28, 29, 35, 36

given-clause algorithm, 7, 16–18, 22, 30–32
grounding, 12
guiding path, 34, 36

Herbrand model, 8, 38
heterogeneous systems, 21–25
heuristic function, 16, 22, 30, 32
homogeneous systems, 21, 25
Horn clause, 4
HPDS, 21
hybrid strategies, 12, 38
hyperresolution, 8, 21, 28, 37, 40

instance generation, 3, 12, 16, 38, 39
instance-based strategies, 2, 3, 11, 12, 16, 19, 37, 38

learning, 24, 35, 37
lemmatization, 5, 35
linear resolution, 4, 6
load balancing, 26, 28

ManySAT, 36
METEOR, 13, 16

model, 2–5, 7, 34–39
model building, 12, 16
model elimination, 2, 4, 5, 21, 22, 38
model-based reasoning, 2, 3, 33, 34, 36, 38
model-elimination tableaux, 2, 4, 38
Moufang identities, 32
multi-search, 13, 20–24, 30–34, 36, 37, 40

non-variable overlap, 14
normalization, 7, 12, 15, 27, 28

ordering-based strategies, 2, 3, 7–9, 12, 13, 16, 19, 20, 22, 25, 29, 37
OTTER, 16, 17, 23, 27, 30, 32

pairs algorithm, 30–32
PaMiraXT, 36
parallel rewriting, 12, 14, 15, 19
parallel speedup, 31–33
parallelism at the clause level, 12, 15, 16, 18, 19
parallelism at the search level, 12, 19, 20
parallelism at the term/literal level, 12, 13
paramodulation, 2, 6, 7, 27, 28, 30, 31
Parthenon, 13, 16
PARTHEO, 13, 16, 21
partitioning, 29, 39
Peers, 3, 25, 27, 30, 31
Peers-mcd, 3, 25, 31, 32
PMSat, 37
portfolio solving, 3, 20, 24, 34
preprocessing, 15
Prolog Technology Theorem Proving, 6, 21
propositional satisfiability, 1, 2, 11, 34

PSATO, 1, 34

redundancy, 3, 5, 10, 11, 20, 29, 32, 36, 39
regressive merging, 5
resolution, 2, 6, 7, 22, 27, 28, 34, 35, 39
Robbins algebras, 31, 32
ROO, 3, 16–18

SAT solver, 1, 3, 12, 34, 36–38
satisfiability, 3, 33, 36
satisfiability modulo assignment, 36
scalability, 33
search overlap, 20, 26, 27, 33, 35–37
selection heuristics, 24
semantic guidance, 8, 12, 37–39
semantic resolution, 8, 37
set of support, 8, 16, 37
SGGS, 2, 38–40
simplification, 7, 9, 15, 28, 29, 31
SMT, 2
SMT solver, 2, 3, 38
subgoal-reduction strategies, 2, 3, 5, 6, 12, 13, 15, 19, 20, 24, 35
subsumption, 7, 8, 11, 21, 28
superposition, 6, 7, 9, 14, 15, 27, 28, 31

task stealing, 15
Team-Work, 3, 22–24, 36
TECHS, 24

unit-resulting resolution, 21, 28

VSIDS, 35