A taxonomy of parallel strategies for deduction [*]

Maria Paola Bonacina [a,**]

[a] *Department of Computer Science*
The University of Iowa
Iowa City, IA 52242-1419, USA
E-mail: bonacina@cs.uiowa.edu

This paper presents a taxonomy of parallel theorem-proving methods based on the control of search (e.g., master-slaves versus peer processes), the granularity of parallelism (e.g., fine, medium and coarse grain) and the nature of the method (e.g., ordering-based versus subgoal-reduction). We analyze how the different approaches to parallelization affect the control of search: while fine and medium-grain methods, as well as master-slaves methods, generally do not modify the sequential search plan, parallel-search methods may combine sequential search plans (*multi-search*) or extend the search plan with the capability of subdividing the search space (*distributed search*). Precisely because the search plan is modified, the latter methods may produce radically different searches than their sequential base, as exemplified by the first distributed proof of the *Robbins theorem* generated by the *Modified Clause-Diffusion* prover *Peers-mcd*. An overview of the state of the field and directions for future research conclude the paper.

1. Introduction

The field of Strategies for Automated Deduction is concerned with the *control* component of deductive methods, that is, with the *search strategies* employed to search for a solution (e.g., a proof in theorem proving, a model in model building, a normal form in term rewriting).

The definition of the search problem in theorem proving is well known. The availability of a sound and refutationally complete inference system I guarantees the existence of a proof (a derivation of a contradiction), for any inconsistent set $H \cup \{\neg\varphi\}$, where H is a set of assumptions and φ a conjectured theorem. However, given an initial state with H and $\neg\varphi$, I can generate many derivations, because an inference system is *non-deterministic*. The search problem is how to control I so that a proof can be found (*fairness*) using as few resources as possible (*efficiency*).

[*] This is an extended version of "Ten years of parallel theorem proving: a perspective," invited paper, *Notes of the Workshop on Strategies in Automated Deduction, Federated Logic Conference*, 3–15, Trento, Italy, July 1999.

[**] Supported in part by the National Science Foundation with grants CCR-97-01508 and EIA-97-29807.

In this search problem, *states* contain partial proofs, *successful states* contain complete proofs, and the *transformation rules*, or *production rules*, are the given inference rules. The latter can be formalized as functions $f\colon \mathcal{P}(\mathcal{L}_\Theta) \rightarrow \mathcal{P}(\mathcal{L}_\Theta) \times \mathcal{P}(\mathcal{L}_\Theta)$, where Θ is a first-order signature, $\mathcal{L}_\Theta$ is a Θ-language of sentences, or clauses, or equations, depending on the problem, $\mathcal{P}(\mathcal{L}_\Theta)$ is its powerset, and f takes as argument a set of premises, and returns a set of elements to be added and a set of elements to be deleted in the current state.

Let *States* denote the set of all possible states of a theorem-proving search problem. Given a set of inference rules I, a search plan Σ is made of at least three components:

- a *rule-selecting function* $\zeta\colon States^* \rightarrow I$, which selects the next rule to be applied based on the history of the search so far;
- a *premise-selecting function* $\xi\colon States^* \rightarrow \mathcal{P}(\mathcal{L}_\Theta)$, which selects the elements of the current state the inference rule should be applied to;
- a *termination-detecting function* $\omega\colon States \rightarrow Bool$, which returns *true* if the given state is successful, $false$ otherwise.

If the current state is not successful, ζ selects a rule f and ξ selects premises $X = \{\psi_1, \ldots, \psi_n\}$, the next step will consist of adding $\pi_1(f(X))$ and/or deleting $\pi_2(f(X))$, where π_1 and π_2 are the projections $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ (e.g., $S_{i+1} = S_i \cup \pi_1(f(X)) - \pi_2(f(X))$). The function ζ also controls backtracking, in search plans that feature it. The sequence of states thus generated (e.g., $S_0 \vdash S_1 \vdash \cdots S_i \vdash \cdots$) forms the *derivation* by I controlled by Σ from the given input, and the combination of I and Σ forms a *deterministic* procedure called a *theorem-proving strategy.*

One approach to the problem of designing new forms of control of the inferences has been to investigate the parallelization of theorem proving. In this paper we give a taxonomy of parallel theorem-proving strategies and we study how the different approaches to parallelization affect the notion of search plan: given a sequential strategy $\mathcal{C} = \langle I, \Sigma \rangle$, and its parallelization $\mathcal{C}' = \langle I, \Sigma' \rangle$, we investigate what Σ' is, depending on the applied parallelization principle. Methods that parallelize the inner algorithms used by the strategy (e.g., parallel rewriting), and methods that schedule in parallel the inference steps prescribed by the sequential search plan, generally do not modify the search plan itself. On the other hand, methods that launch multiple search processes in parallel need to modify the search plan (otherwise they would execute n copies of the same search!), and are therefore the most interesting from the point of view of the control of search. We distinguish between *multi-search strategies*, that assign to each parallel process a different search plan, and *distributed-search strategies*, that assign to each parallel process a different portion of the search space. The two approaches are not mutually exclusive, and both require communication (multi-search without communication is trivial and distributed search without communication is incomplete).

If the parallelization *changes* the search plan, the parallel strategy generates *different searches*, rather than executing faster the same steps of the sequential search. Thus, if the sequential search is not optimal, a super-linear speed-up may occur. As an example of this phaenomenon, we report on an experiment with our distributed prover *Peers-mcd*, which is the parallelization of McCune's prover EQP based on the Modified Clause-Diffusion methodology [16,14]. We tried Peers-mcd on the theorem *Robbins algebras are Boolean*, which was proved first by EQP [81], and we observe that the distributed prover shows super-linear speed-up with respect to EQP with all other factors being equal (same hardware, same version of EQP, same inference system).

Instances of super-linear speed-up by parallel provers may provide a lead into further research in sequential search plans, since one would like to find a sequential search plan that reproduces the faster search generated in parallel. We emphasize that finding a general and fully automated solution – without giving "hints" to the sequential prover in the form of additional lemmas or patterns in the input file – is hard. It is the inverse of the problem we are studying in this survey: given a parallel plan Σ, find a sequential plan Σ' that simulates it.

Previous surveys or collections on parallel deduction include [54,94,21]. The main classification criterion of [94] is whether the parallel components *cooperate* or *compete* to find a solution. This criterion appears to have been inspired by the classical distinction between *AND-parallelism* and *OR-parallelism* in subgoal-reduction strategies (and logic programming), and was designed to supersede it. One of the criteria of our taxonomy is the distinction of *parallelism at the term level*, *at the clause level*, and *at the search level*, introduced in [21]. If AND-parallelism (try in parallel the conjuncts of the current goal) or OR-parallelism (try in parallel different clauses with the selected literal of the current goal) is applied within one search process, it is an instance of parallelism at the clause level[1]. If, on the other hand, parallel search processes, each developing its own derivation, employ different AND-rules or different OR-rules, then we have an instance of parallel search.

The survey presented in this paper differs from the one in [21] in several ways. First, it studies the effect of parallelization on the search plan, which was not investigated in [21]. For this purpose, it distinguishes between those approaches that separate the control of parallelism from the control of deduction and those approaches that combine them. Furthermore, it distinguishes between *multi-search* and *distributed search* within parallel search. Both dichotomies did not appear in [21]. Second, since approaches based on parallel search modify the sequential notion of search plan, it proposes formal definitions of *multi-search plan* and *distributed-search plan* that match the surveyed strategies. Third, it covers many papers (e.g., [1,31,30,42,40,52,55,57,65,72,74,83,91,93,95,99,102,104]) that

[1] AND-parallelism may also be considered as parallelism at the term level, since the data accessed in parallel are the literals of a goal clause, hence subexpressions of formulae.

have appeared since [21] was written in 1992, offering an up to date survey of the field. Last, the analysis in [21] emphasized the difficulty of parallelizing contraction-based strategies, and was more optimistic on the parallelization of subgoal-reduction strategies. After eight years of additional research by several authors, we reassess our evaluation of the difficulty of parallelizing theorem proving, including also subgoal-reduction strategies.

2. Sequential theorem-proving strategies

We begin by recalling basic concepts and terminology that will be referred to in the following. Surveys on the subject include [12,47,53,85,11,49,84,59,24,6, 34,20], where the interested reader may find extensive bibliographies.

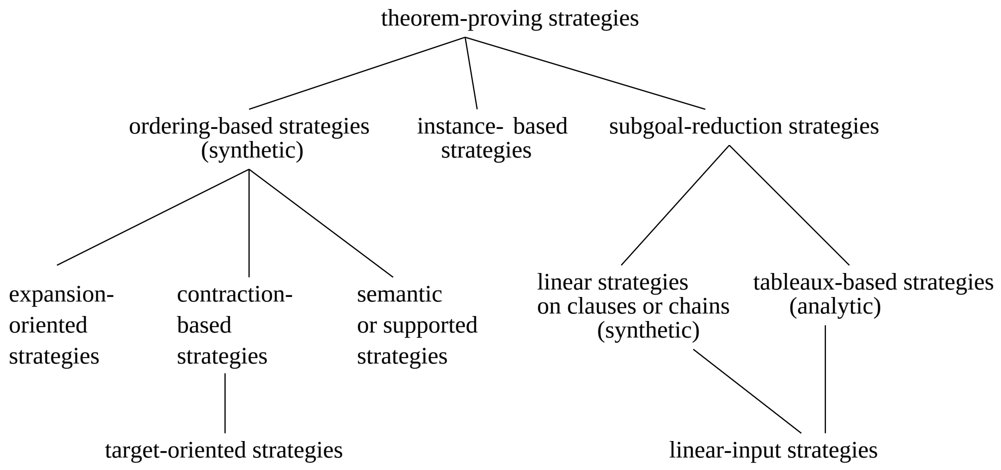
theorem-proving strategies
ordering-based strategies (synthetic)
instance- based strategies
subgoal-reduction strategies
expansion-oriented strategies
contraction-based strategies
semantic or supported strategies
linear strategies on clauses or chains (synthetic)
tableaux-based strategies (analytic)
target-oriented strategies
linear-input strategies

Figure 1. A taxonomy of theorem-proving strategies.

Ordering-based strategies (see Figure 1) include the strategies resulting from the merging of the resolution-paramodulation paradigm with the term-rewriting and Knuth-Bendix paradigm. We call them *ordering-based* to emphasize that exactly because the strategy works with a set of clauses, it can use a *well-founded ordering* to order them, and possibly *delete* clauses that are greater than and entailed by others. These strategies work by adding clauses to the set – *expansion* inferences such as resolution and paramodulation – and deleting clauses or replacing them by smaller ones – *contraction* inferences, such as subsumption and normalization by rewriting. The state of a derivation contains the set of clauses that have been generated and kept (i.e., not deleted by contraction). Since this set typically grows very large, the strategy may employ *eager-contraction* search plans to keep it maximally reduced, and *indexing techniques* to retrieve elements.

Strategies that feature only expansion rules, or apply contraction rules only for *forward contraction* (i.e., normalization of newly generated data with respect to the existing set), are called *expansion-oriented*. Strategies with both forward and *backward contraction* (i.e., normalization of data already in the set with respect to the new insertions), and an eager-contraction search plan, are called *contraction-based*. Ordering-based strategies are not goal-sensitive in general: *semantic* or *supported strategies* use semantics to introduce some goal-sensitivity; *target-oriented strategies* are equational strategies that may employ heuristic syntactic criteria to enhance goal-sensitivity.

By generating and keeping clauses, an ordering-based strategy *builds many proof attempts implicitly*; only when an empty clause is generated, the strategy reconstructs the proof (i.e., an ancestor-graph of $\Box$) from the set of generated clauses, starting from the empty clause and proceeding backward towards the input clauses.

Subgoal-reduction strategies include those based on linear resolution, model elimination, hence Prolog Technology Theorem Proving (PTTP), problem reduction format methods, and tableaux-based methods. The name *subgoal-reduction strategies* emphasizes that each step in the derivation consists in reducing a goal to subgoals (e.g., the current center clause in linear resolution, the current chain in model elimination, the selected leaf in a tableau), so that these strategies are goal-sensitive. The state of a derivation contains the current proof attempt (e.g., a linear deduction, a tableau). Since subgoals may repeat and cause redundant inferences, various forms of *lemmatization*, or *caching*, may be used to save already computed solutions and limit repetitions.

Linear resolution and model elimination are *linear* strategies: they search for a *linear* ancestor-graph of $\Box$. While ordering-based strategies restrict the search, by imposing local requirements on each inference step (e.g., resolve only on maximal literals), linear strategies restrict the search by imposing requirements on the shape of the proof (i.e., linear). Furthermore, model elimination is a *linear-input* strategy. Linear resolution and model elimination with chains are *synthetic* strategies, because they generate clauses from clauses and chains from chains, respectively. All ordering-based strategies are synthetic. Subgoal-reduction strategies based on tableaux, on the other hand, are *analytic*, because they build a survey of interpretations by decomposing formulae into subformulae, and obtain a proof of unsatisfiability by showing that no interpretation is a model (closed tableau)[2]. In synthetic strategies, most general unifiers apply to the generated data (clauses or chains), while in analytic strategies most general unifiers apply to the entire tableau, because the variables of a formula may appear on more than one branch in the tableau. Some authors refer to this characteristic by saying that variables in tableaux are *rigid*. Some features of ordering-based and subgoal-reduction strategies are summarized in Table 1.

Instance-based strategies implement directly the Herbrand theorem by gen-

[2] Model elimination with chains can be interpretated in this fashion also, since a chain encodes a candidate model.

	Ordering-based	Subgoal-reduction
Data	set of objects	one goal-object at a time
Proof attempts built	many implicitly	one at a time
Proof-confluent	yes	no
Goal-sensitive	no	yes
Contraction	yes	no
Generated search space	all generated clauses	all tried proof attempts
Active search space	all kept clauses	the current proof attempt
Generated proof	the ancestor-graph of $\Box$	the closed proof attempt

Table 1
Some features of ordering-based and subgoal-reduction strategies.

erating sets of ground instances of the existing clauses and testing them for unsatisfiability by propositional methods. We place them in a separate class in the middle between ordering-based and subgoal-reduction strategies (see Figure 1), because they have something in common with both. On one hand, they are synthetic (e.g., generate instances) like ordering-based strategies; on the other hand, they resemble tableau-based strategies because of the way the latter close the tableau by progressively instantiating the formulae.

We consider next the search plans. Ordering-based strategies typically use best-first search, with various heuristic functions. Because they accumulate all generated non-redundant data, ordering-based strategies are *proof-confluent*, meaning that if the inference system (including contraction) is complete, and the search plan is fair, the order of inferences does not affect completeness[3]. These strategies *do not need backtracking*, and whatever they do may further one of the implicit proof attempts. As an example, consider the basic control algorithm used in ordering-based provers such as Otter, with a list of clauses *to-be-selected* (conceptually, the frontier of the search) and a list of clauses *already-selected*. Assume that the search plan is depth-first, so that *to-be-selected* is handled as a stack[4]. Even with depth-first search this kind of strategy does not need backtracking, because if it selects a clause φ that produces no children (i.e., a clause that "fails"), all it has to do is to move φ to *already-selected* and select another clause.

Some authors call this control *saturation-based*, but we find this name po-

[3] The notion of proof-confluence is analogous to the notion of confluence in rewriting: in a confluent term rewriting system the order of rewrites does not affect the normal form, if one exists.

[4] This strategy is obviously incomplete.

tentially misleading in a few ways. First, it suggests that the goal of the strategy is to generate a saturated set of clauses[5], whereas the goal of the strategy is to find a proof. Second, it suggests that all searches are exhaustive (e.g., *level saturation* in resolution-based theorem proving means breadth-first search), whereas all non-trivial results by these strategies were obtained by using non-exhaustive heuristics (e.g., best-first search with heuristic function *weight* and deletion of all clauses above a certain weight in Otter and similar provers). Third, the strategy terminates with a non-trivial saturated set, only if it fails to find a proof, because the input set is satisfiable and has a finite saturated set. In many satisfiable cases the saturated set is infinite, so that the strategy does not halt. In either case, it seems odd to name a theorem-proving strategy by a feature that it displays when it fails to find a proof. Of course, one can say that the strategy generates a saturated set also when it finds a proof, because $\Box$ subsumes all other clauses, and $\{\Box\}$ is saturated. This is a correct and elegant solution from a purely theoretical point of view that considers the inference system only, but it is not satisfactory from a point of view that aims at combining theory and practice, and also takes the search plan into account: when the strategy halts having generated $\Box$, it is not true in general that it has saturated the search space, and no theorem prover subsumes all generated clauses by $\Box$. Rather, it uses them to reconstruct the proof.

Subgoal-reduction strategies typically use *depth-first search with backtracking and iterative deepening* (DFID). In principle, one can search for a linear refutation, or a closed tableau, by any search plan, including breadth-first and best-first search plans. Such a search plan, however, requires the strategy to generate and keep in memory a set of proof attempts (linear deductions or tableaux), which represents the frontier of the search. Depth-first search, with *iterative deepening* to preserve fairness, is usually preferred because it requires the program to keep in memory only one proof attempt at a time, and the others will be considered upon backtracking. If no step applies to the current linear deduction or tableau, or the current limit of iterative deepening has been reached, the search plan backtracks. The use of backtracking means that the strategy is *not proof-confluent*.

Ordering-based strategies use best-first search because each node in the search space is a clause, but keeping in memory the whole frontier when each node in the search space is a deduction, or a tableau, may be more onerous.

In addition, the linearity requirement leads to depth-first search and appears at odd with other search plans: take best-first search, let $D_1, \ldots, D_n$ be the linear deductions being pursued, and let $\varphi_1, \ldots, \varphi_n$ be the current goals (i.e., current center clauses) of $D_1, \ldots, D_n$, respectively. Assume that φ_1 and an ancestor of φ_2 generate the empty clause: it would be natural to generate it and conclude the

[5] A set of clauses is *saturated* with respect to an inference system and a notion of redundancy if all clauses that could be derived by the system are redundant.

search, but the strategy forbids such a move, because the generated refutation would not be linear.

In analytic tableaux [89] (or equivalently the basic cut-free Gentzen system $\mathcal{G}$ [89,90]), which may be considered as an ancestor of subgoal-reduction strategies, backtracking is not an issue, because after trying a proof attempt (e.g., a set of applications of the γ-rule to instantiate the universally quantified variables), one switches to the next one simply by extending the tableau with fresh copies of the universally quantified formulae. Mechanical subgoal-reduction strategies instantiate universally quantified variables by unification. If they use depth-first search and therefore keep in memory only one proof attempt at a time, they need to undo the unsuccessful instantiation of variables and switch to a new proof attempt by backtracking[6].

We have emphasized best-first search for ordering-based strategies and DFID for subgoal-reduction strategies, because they are the most commonly used, and in this paper we are interested in the parallelization of the strategies used in practice. In principle, however, all types of inference can be matched with search plans other than the dominating ones, and one of the purposes of studying parallelism is precisely to investigate new forms of control.

3. Principles of parallelization

In *parallelism at the term level*, the data accessed in parallel are subexpressions of a formula such as terms or literals (here a "formula" means the basic unit of the language the inference system is for), and the parallel operations are subtasks of an inference step. Thus, parallelism happens *below the inference level*, as in *parallel matching*, *parallel unification* and *parallel term rewriting*. The rationale for parallelism at the term level is that since a strategy executes these low-level operations very frequently, if one can make them very fast by parallelism, the overall performance of the strategy should improve.

In *parallelism at the clause level*, the data accessed in parallel are formulae, and the parallel operations are inferences, so that parallelism is *at the inference level*. Theorem-proving methods with *parallel inferences within a single search* (e.g., parallel resolution steps) belong to this category. The motivation is to speed-up the execution of the strategy by doing many inferences at each step.

In *parallelism at the search level*, multiple deductive processes search in parallel the space of the problem until one of them finds a proof. Each process executes a strategy, develops a derivation and builds its own set of data.

[6] In recent years, there has been some progress in the design of proof-confluent subgoal-reduction strategies [13,8,10]: the approach of [10], however, still needs to rely on the γ-rule for universally quantified variables; the approach of [13,8] resembles the instance-based approach of [75]: unification is used to generate instances, but subsumption cannot be applied to remove redundant instances.

Parallelism at the term, clause, and search level aim at representing fine-grain, medium-grain and coarse-grain parallelism for deduction, respectively, because the distinction is based on the granularity of the parallel operations and the data they manipulate, as shown in Table 2.

Parallelism	Data accessed in parallel	Parallel operations
At the term level	subexpressions of formulae	subtasks of inference
At the clause level	formulae	inferences
At the search level	sets of formulae	derivations

Table 2
Granularities of parallelism in deduction.

The border between parallelism at the term and clause level may not be always clear cut. For instance, *parallel term rewriting* overlaps with both parallelism at the term level (the data accessed in parallel are terms), and parallelism at the clause level (if each rewrite step is regarded as an inference). Thus, its classification is affected also by the application: in a context where the whole computation is a reduction one may consider parallel term rewriting as parallelism at the clause level; this may be the case for parallel rewriting machines (e.g., [62,48,73,1,2]) and parallel interpreters of functional languages (e.g., [71]). In the context of theorem proving, where the whole computation is better seen as a search, and each normalization, rather than each rewrite step, may constitute an inference, it is more natural to consider parallel term rewriting as parallelism at the term level.

A key factor in classifying methods with parallel search is how they differentiate and combine the activities of the deductive processes. A possibility is to subdivide the search space among the processes by subdividing the inferences or decomposing the problem: we call this principle *distributed search*, using the word "distributed" in its literal meaning of "giving each a share of something". Distributed search needs communication to preserve completeness and load-balance; its aim is to obtain a speed-up over the sequential search by ensuring that each parallel process has to search only a part of the whole space.

Another possibility is to let each process handle the problem in its entirety and differentiate them by having each process use a different search plan: we call this principle *multi-search*, because multiple plans are applied. Multi-search needs communication to allow every process to take advantage of the results of others, and also for completeness, if some processes employ unfair search plans. The intuition behind multi-search is that parallel processes executing different search plans will search the space of the problem in a different order. Thus, multi-search aims at obtaining a speed-up over sequential search by letting each process take advantage of data earlier than its search plan would allow, because such data has been generated and communicated by other processes following different plans. Note that also distributed search may induce this effect, because a process barred from exploring a certain part of the search space may reach sooner deeper parts of its allowed portion, and send to other processes not only data that they would not generate because of the partition, but also data that they would generate much later. Distributed search and multi-search are not mutually exclusive: a strategy may feature instances of both principles.

A third option is to let each process have the same problem and search plan, but assign them different inference systems, leading to what is called a *heterogeneous* system. A motivation for heterogeneous systems is to use parallelism to combine subgoal-reduction and ordering-based strategies, typically by enabling subgoal-reduction processes to use clauses generated by the ordering-based processes as *lemmas* (e.g., see [25] for a study of lemmatization as combination of forward and backward reasoning and more references). Another possibility is to combine a theorem-proving strategy with a model-building strategy (e.g., [32] for the idea of simultaneous search for refutations and models, with a sequential control). If a heterogeneous system is combined with multi-search, each process executes a different theorem-proving strategy. On the other hand, parallel search approaches where all processes have the same inference system are called *homogeneous*. Figure 2 summarizes this classification of parallelization principles.

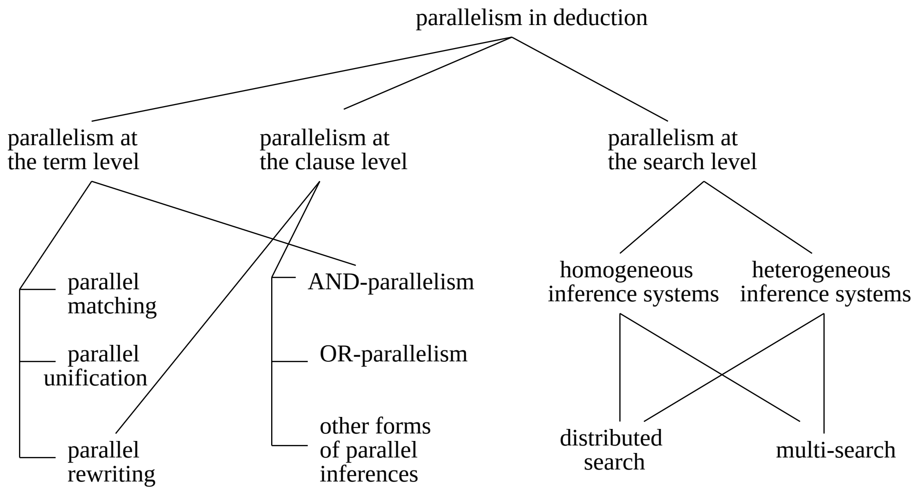
parallelism in deduction
parallelism at the term level
parallelism at the clause level
parallelism at the search level
parallel matching
parallel unification
parallel rewriting
AND-parallelism
OR-parallelism
other forms of parallel inferences
homogeneous inference systems
heterogeneous inference systems
distributed search
multi-search

Figure 2. Types of parallelism in deduction.

While in theory everything can be implemented in either shared memory or distributed memory, most methods with parallelism at the term or clause level use shared memory, and most parallel search methods use distributed memory. For this reason, *distributed deduction* and *distributed strategies* have been used for methods with parallelism at the search level, implemented in environments with distributed memory (e.g., networks of workstations), regardless of whether the method employs distributed search or multi-search. In this taxonomy, we use "distributed" for distributed search. Tools for parallel programming often let the high-level application programmer ignore whether the memory is physically shared or distributed. For example, the definition of a parallel-search strategy typically requires separate data bases and communication by message passing; however, the latter can be implemented over a network or in a shared memory, depending on the architecture (e.g., network of workstations or shared-memory multi-processor) specified at compilation time. Thus, what is relevant is not the physical memory, but the logical view of the memory: for instance, for ordering-based strategies, whether the method assumes a shared data base of clauses, or separate data bases and communication by message passing. In the following, we use shared and distributed referring to the definition of a method, not its implementation on either type of physical memory.

4. The search plan and parallelism at the term level

We begin by considering *parallel term rewriting.* Strategies for *term rewriting* can be seen as linear-input subgoal-reduction strategies: given a set R of rewrite rules, and a term t_0 to be normalized, states are pairs (R, t_i), where t_i is the reduced form of t_0 after i steps, and the search plan has the form: $\Sigma = \langle \xi, \omega \rangle$, where ξ selects the redex for the next step and $\omega((R, t_i)) = \mathit{true}$ if t_i is R-irreducible. If R is *confluent*, the normal form is unique regardless of the order of rewriting, and therefore there is no need for backtracking. In theorem proving, in general, one cannot assume that R is confluent, and seeks any R-irreducible form without backtracking. Since there is only one inference rule (simplification by rewriting) and no backtracking, there is no need for ζ.

A parallel search plan for rewriting could be defined as a pair $\Sigma' = \langle \xi', \omega \rangle$, where ξ' selects a set of redexes to be rewritten in parallel. The question is how to determine such a set. Given a term to be rewritten t, and two rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ that apply to t, their redexes are *overlapping* if the two rules *overlap*, that is, l_1 unifies with a non-variable subterm of l_2 (e.g., given the term $f(h(a,a),a)$, the subterms $f(h(a,a),a)$ and $h(a,a)$ are overlapping redexes for the rewrite rules $h(x,x) \to x$ and $f(h(z,y),z) \to y$, because $h(x,x)$ unifies with $h(z,y)$). Otherwise, the redexes are *non-overlapping.* Non-overlapping redexes include *disjoint* redexes (e.g., given the term $f(h(a,a),g(b))$, the subterms $h(a,a)$ and $g(b)$ are disjoint redexes for the rewrite rules $h(x,x) \to x$ and $g(b) \to c$), and redexes such that the two rules overlap at a variable position (e.g., the rewrite rules $h(x,x) \to x$ and $f(y,c) \to y$ overlap at a variable position in $f(h(a,a),c)$, so that $f(h(a,a),c)$ and $h(a,a)$ are non-overlapping redexes).

If terms are represented as trees, all and only the *disjoint* redexes can be rewritten in parallel without causing conflicts. In *concurrent rewriting* [73] terms are represented as dags (direct acyclic graphs); in a dag, all and only the *non-overlapping* redexes can be rewritten in parallel without causing conflicts. It follows that dags allow more parallelism than trees. One could then say that tree-behavior corresponds to a ξ' that selects all disjoint redexes, and dag-behavior corresponds to a ξ' that selects all non-overlapping redexes (*maximal concurrent rewriting*). However, it appears that ξ', hence Σ', are *superfluous*, in the sense that the data structure chosen for terms dictates what is done in parallel.

The data-driven nature of this sort of parallelism becomes even more apparent considering the implementations of concurrent rewriting in [73,1] and conditional concurrent rewriting in [2]: concurrent rewriting was realized by a network where processes and communication channels correspond, respectively, to nodes and arcs of the dag of the term to be rewritten. Thus, there is a process for each symbol in the signature, and each process performs actions based on messages received from its neighbors in the term structure.

This fine-grained philosophy was extended to *concurrent ground completion* in [72,86]: the network implements the dags representing the sides of the equations and all their subterms, and, in addition to subterm-arcs, there are equality-arcs to represent the equations (since a total ordering on ground terms is assumed, all equations can be oriented into rewrite rules, and the equality-arcs are rewrite-arcs). The entire computation is driven by message-passing. By exchanging messages, neighbor processes detect configurations that satisfy the conditions to perform inference steps (i.e., a superposition, or a rewrite, although in the ground case superposition steps also reduce to rewrite steps). Once such a configuration has been detected, the inference is executed: new equations are generated by adding arcs; symmetrically, deletions of equations by backward contraction would correspond to deletions of arcs. However, arcs cannot be actually deleted, but a time-stamp mechanism is in place to enable the processes to recognize arcs that were made obsolete by contraction, and are therefore disabled. From the point of view of our analysis, the most relevant aspect is that there is no notion of a process executing a search plan: given a network of terms all conflict-free inferences happen concurrently, so that concurrency *replaces* the control provided by a search plan.

The method of [52] may be considered parallelism at the clause level rather than at the term level, because each node holds a clause. However, its approach to the control issue resembles that of [73,1,2,72,86], in the sense that it is another case where *data-driven concurrency replaces the search plan.* For instance, unit resolution for propositional Horn clauses (the method is described for expansion-oriented strategies, mostly for propositional logic) is realized by having nodes broadcast unit clauses; all nodes resolve received unit clauses with their clauses and broadcast the unit clauses thus generated, until some node generates an empty clause.

While using the concurrency of the data structures to replace the search plan may be attractive for *ground* inferences, it is problematic for general theorem proving, where the role of the search plan is fundamental. The fine-grained concurrent completion of [72,86] has not been extended to the non-ground case for several reasons. As pointed out also in [86], this approach is very demanding in terms of communication, because even subtasks of inferences, such as unifiability tests (which reduce to identity tests in the ground case), require the nodes to exchange messages. In the implementation of this method in the CWD system [86] care has been taken to reduce the amount of message-passing with respect to the theoretical definition, but it remains very high. This problem would get much worse in the general case, where substitutions also need to be computed and communicated. Another obstacle mentioned in [86] is that in the representation of terms as dags variables are shared, so that a rather complicated mechanism should be designed for variable renaming.

In our analysis these difficulties depend on a granularity of parallelism that is too small for theorem proving. The amount of communication is too high, because each node holds too little information (i.e., a single symbol), and unification, orientation, and rewriting are small tasks with respect to a theorem-proving derivation, so that having to perform communication to achieve these basic operations represents too much overhead. The difficulty with variable renaming is an aspect of the problem of managing the growth of the network of terms because of expansion. In term rewriting, there is no generation of new equations by expansion inferences. In ground completion, the generation of new equations consists in recognizing that certain ground terms are in the same congruence class[7], and this can still be handled by adding arcs between nodes. In theorem proving also, at least in principle, this might be done, because the signature is static, and all new terms are formed with the given function symbols, but there are at least two difficulties. First, each equation needs its own variables, leading to the variable renaming problem mentioned above. Second, a very high number of equations can be generated, yielding an extremely complex network, since each new equation means an additional arc and arcs are communication channels.

A third potential problem is related to *backward contraction.* In theorem proving, and completion, every equation normalized by backward-contraction needs to be applied to backward-contract the other clauses in the data base, so that each backward contraction step may induce many. This avalanche growth of contraction steps may cause a *backward-contraction bottleneck* [21], if many processes ask for write-access to a shared data base to do backward contraction. The method of [72,86] is completely distributed, and therefore the problem with backward-contraction could not appear in form of a bottleneck in shared memory.

[7] Computing the completion of a set of ground equations consists in computing a congruence closure [60].

However, backward-contraction in the non-ground case would represent a problem for the time-stamp mechanism devised in [72,86] to keep track of deletions: the network would not only grow very complex because of additions of equations, but many arcs would become obsolete because of backward-contraction, resulting in high overhead from the time-stamp mechanism.

A different approach was exemplified by the fine-grained parallelization of rewriting and completion in PaReDux [29,31]. PaReDuX implements completion on top of a low-level implementation of parallel rewriting using threads. The philosophy of PaReDux is to be *strategy-compliant*, in the sense that inferences are guaranteed to occur in the same order as in the sequential strategy. In other words, if $\mathcal{C} = \langle I, \Sigma \rangle$ is a sequential strategy, and $\mathcal{C}'$ is the strategy executed by a strategy-compliant parallel implementation of $\mathcal{C}$, then $\mathcal{C}' = \mathcal{C}$. The aim of this type of approach is to obtain a speed-up by executing faster the same steps in the same order, but the downside is excluding the use of parallelism to explore the search space in different ways[8].

In summary, exactly because parallelism at the term level is *below* the level where the search plan makes decisions, either the search plan is *replaced* by a low-level data-driven form of concurrency, as in concurrent term rewriting, or the search plan is left *untouched*, as in strategy-compliant parallelizations.

5. The search plan and parallelism at the clause level

In parallelism at the clause level, the common philosophy is to consider inferences, or groups thereof, as *tasks*, and conceive the control problem as a *task scheduling* problem. This approach emerges from studying clause-level parallelizations of both ordering-based and subgoal-reduction strategies.

For the ordering-based strategies, we consider the parallel prover ROO [77,78] as a paradigmatic example. ROO was designed to parallelize the well-known Otter theorem prover [79]. The core of Otter's control is a main loop, which works with a list of clauses to be selected, called `sos` for historical reasons (the set of support strategy), and a list of clauses already selected, called `usable`, because these clauses can be used for inferences. At each iteration, Otter selects a clause φ – called the *given-clause* – from `sos`, generates all the clauses that can be generated by the active expansion inference rules from φ and clauses in `usable`, inserts the non-trivial normal forms of these clauses into `sos` (forward contraction), applies them to contract clauses in `sos` and `usable` (backward contraction), and appends φ to `usable`. In the backward-contraction phase, reducible clauses are removed from `sos` and `usable`, and their non-trivial normal forms are inserted into `sos` and also applied as backward-contractors. The iteration halts when `sos` is empty. By default, the given-clause is the lightest in `sos`, corresponding to

[8] See also Section 6.1 for later developments of PaReDuX.

a best-first search with the weight of clauses as evaluation function, where the default definition of “weight” is the number of symbols.

The basic idea of ROO was to have several given-clauses active in parallel. Let *Task A* be the task of performing the body of the main loop of Otter for a given-clause: in ROO multiple parallel processes execute Task A, each with a different given-clause, on shared `sos` and `usable`. Clearly, this causes *conflicts* among the parallel inferences, hence among the parallel processes accessing the shared lists. A first type of conflict arises when different processes executing Task A need to append generated clauses to `sos`. A second type of conflict happens between expansion and contraction: assume processes p_1 and p_2 generate a set S_1 and a set S_2, respectively, of new clauses; the clauses in S_1 are not reduced with respect to the clauses in S_2 and vice versa; if the processes are allowed to insert them into `sos`, the conflict is solved in favor of expansion (eager parallel expansion), but the data base is not inter-reduced. A third type of conflict (between contraction inferences) occurs when different processes executing Task A need to delete backward-contracted clauses in `sos` and `usable`.

The first two problems were addressed in ROO by having a third list, called `k-list`, and establishing that all processes append clauses to `k-list` rather than `sos`. The third one was addressed by establishing that the processes executing Task A do not perform deletions on `sos` and `usable`, but append the identifiers of the clauses they need to delete to a fourth list, called `to-be-deleted`. A different task, Task B, was defined to handle `k-list` and `to-be-deleted`, with the provision that only one process can execute Task B at any given time. Thus, Task B consisted of extracting a clause from `k-list`, forward-contracting it, inserting its normal form into `sos`, using it for backward-contraction, and deleting the clauses whose identifier is in `to-be-deleted`. In essence, in order to avoid conflicts among parallel inferences, backward-contraction inferences were handled sequentially (i.e., only one process executes Task B). The problem with this solution was that if a high amount of backward contraction was required (e.g., in equational theories), Task B would become a *backward-contraction bottleneck* and all the other processes scheduled to execute Task A would starve.

From the point of view of the search plan, the control of ROO is obtained by breaking the activities of Otter into *tasks*: each given-clause is a task of type A (the expansion inferences with that given-clause), and each clause in the `k-list` is a task of type B (the contraction inferences with that clause). Then, a parallel search plan is a *scheduler* that assigns tasks to parallel processes. Similar considerations apply to other approaches based on parallelism at the clause level such as PARROT [70], which was a predecessor of ROO for expansion-oriented strategies, and the parallel implementation of completion in [103], where each inference rule of completion was considered a type of task or *transition*.

A more recent example in the instance-based family is the clause-level parallelization of hyper-linking in [102]. Hyper-linking [75] interleaves instance generation by *hyperlinking* (similar to hyperresolution, but every step generates an instance of the nucleus, not a hyperresolvent) and unsatisfiability testing by the *Davis-Putnam algorithm* [38,37]. Two approaches for introducing parallelism were considered in [102]: one consisted in running the Davis-Putnam phase in parallel with the subsequent hyperlinking phase (e.g., generate S_{i+1} while testing the unsatisfiability of S_i); the other one consisted in performing parallel hyperlinking steps within each phase of hyperlinking.

The second approach yields in turn three variants, depending on the sequential implementation of hyperlinking being parallelized. If the sequential implementation is list-based (clauses and literals stored in lists, with hyperinstances generated by list traversal), the parallelization adopts *clause-level parallelism*, where each parallel process selects a nucleus and generates all its hyperinstances, with the lists in shared memory. If the sequential implementation is net-based (a net of literals, similar in spirit to discrimination nets, with hyperinstances generated by having each literal traversing the net), the parallelization may adopt either *literal-level parallelism* or *flow-level parallelism*. In the former, each parallel process takes care of one literal, and in the latter, each parallel process takes care of a path in the net. In both cases, the net is held in shared memory and the two approaches can also be combined. In all three approaches critical regions and semaphores are used to control access to the shared memory.

In hyperlinking, contraction is limited to forward unit subsumption and clausal simplification. These contraction inferences can be implemented as tests during the generation of a hyperinstance, and a backward-contraction bottleneck does not occur. However, there are still the problems of concurrent insertion of hyperinstances in the shared data base and insertion of duplicate hyperinstances. These are solved by endowing each process with a local memory, where it stores temporarily its hyperinstances, and updating the shared list, or net, sequentially. What is most relevant to our discussion is that in all three medium-grain parallel hyperlinking techniques (clause-level, literal-level and flow-level), the control component is a *scheduler* that assigns to processes clauses, literals or paths, respectively.

The notion of controlling parallel inferences via task scheduling is common also to subgoal-reduction strategies with parallelism at the clause level, where the tasks are the subgoals. Since subgoal-reduction strategies try one proof at a time, backtrack upon failure and try another one, the first idea to introduce parallelism in such strategies was to try in parallel the proof attempts that would be tried sequentially. Because upon backtracking the strategy tries another proof by trying another clause (another rule in Prolog terminology), this idea leads to trying clauses in parallel, that is *OR-parallelism*. The first prover based on this principle was PARTHENON [28], soon followed by PARTHEO [88], in a tableau-based context, and METEOR [3], in the context of PTTP. In these systems, the shared structure is the stack of goal literals (e.g., in PARTHENON and the version of METEOR in shared memory), each goal literal is a *task*, and the essential part of the parallel search plan is the mechanism that assigns tasks to processes (e.g., *task stealing*).

Two more recent instances of parallelism at the clause level are the HOT prover [74] for higher-order analytic tableaux (as opposed to first-order model-elimination tableaux as in PARTHEO), and the system of [65] to parallelize the Gentzen-style (propositional) proofs generated as subproofs during an interactive proof with the Nuprl system [51]. The above considerations on the search plan as a scheduler still apply, as well as the notion of *task*: regardless of whether we consider a model-elimination tableau, or an analytic tableau, or a Gentzen-style proof, the leaf of every open branch is a task. The shared structure is a *blackboard* [50] in HOT, whereas the system of [65] uses both messages and shared memory with locks.

In summary, because parallelism at the clause level parallelizes the inferences within one derivation, and the inferences within one derivation are precisely what the search plan is supposed to order, a parallel search plan for parallelism at the clause level is essentially a *scheduler* that assigns inferences to parallel processes.

6. The search plan and parallelism at the search level

In parallelism at the search level, each process generates a derivation, and therefore needs to execute a search plan. In addition to deduction, parallel search involves controlling *communication* (for both multi-search and distributed search) and *subdivision of the work* (for distributed search). If the control of deduction and the control of parallelism are *separate*, the search plan is responsible only for the control of deduction like in the sequential case, and therefore *the parallelization does not modify the search plan.* This happens in *master-and-slaves* approaches, where the master is responsible for communication, subdivision of work and no deduction, while each slave generates a derivation according to a sequential search plan. If the control of deduction and the control of parallelism are *combined*, the search plan needs to control *both*, and therefore *a different notion of search plan is necessary.* This happens in approaches where the processes are *peers.*

6.1. Parallel search with master and slaves

In the *master-slave* organization, all communication happens between the master and each slave. A typical way of applying it to distributed search is to let the master process subdivide the work, and assign work batches to the slaves. This was the case for instance in PSATO [104], a distributed-search master-slave implementation of the Davis-Putnam algorithm [38,37] for propositional satisfiability. For this algorithm, the search space is the static and finite tree of its recursive calls, so that the distribution consisted in assigning subtrees to slaves. If a slave reports “satisfiable”, the master halts all slaves, whereas only when all slaves have reported “unsatisfiable”, can the master declare the set unsatisfiable.

All the following methods are for theorem proving, and therefore it is sufficient that a deductive process finds a proof to terminate the parallel search.

If multi-search and master-slave hierarchy are combined, the slaves try different search plans on the whole problem, and the master process is responsible for merging their results. An example of this approach was the *Team-Work method*, originally conceived for ordering-based equational strategies [39,4,5,46,44,42], and later extended to a framework for applications in distributed artificial intelligence [45]. The master process, called *supervisor*, assigns to every slave the theorem-proving problem, a time period, and a different search plan. The slaves, called *experts*, develop their derivations. When the time period expires, each slave interrupts its derivation, evaluates it based on heuristics, and sends the result of the evaluation to the master. The master determines which slave had the best result, and broadcasts this information, so that this slave becomes the new master, and the other slaves send it their "best" equations based on heuristics. The master adds them to its data base and then broadcasts it to the slaves, together with a time period and a search plan for the new round of deductions. Team-Work seeks to achieve a speed-up by *interleaving* search plans (e.g., slave p_i applies search plan Σ_i to a data base produced in the previous round by a search plan Σ_j), and also combining their results (because selected equations from all slaves are added to the new common data base). In Team-Work the master-slave hierarchy is not rigid, since the role of master floats, but the method belongs to this class because it has a central control. The PaReDux system also evolved from the fine-grained approach of [31,29] to a master-slave concept à la Team-Work [30].

For subgoal-reduction strategies, the "nagging" technique of [91] combined AND-parallelism, master-slave organization and multi-search in the context of PTTP. The well-known drawback of AND-parallelism at the clause level is the dependencies induced by variables shared among the conjuncts. The "nagging" technique assumes that the master is a standard PTTP prover, which, however, may fork slave processes to try different permutations of the current goal, and if a permutation fails the slave sends a message to the master to force it to backtrack. Thus, AND-parallelism is employed only for early failure detection. We interpret "nagging" as multi-search, because trying different permutations can be understood as assigning different AND-rules (i.e., different ξ) to the processes.

In heterogeneous systems with a master-slaves organization, the slaves try different sets of inference rules, and the master combines their results. For instance, the HPDS system of [92] had three deductive processes, one executing Guided Linear Deduction (GLD), which is similar to model elimination, one executing hyperresolution (HR), and one executing unit-resulting resolution (UR). All three processes featured forward and backward subsumption, and a depth-first search plan with iterative deepening. Every process sends the clauses it generates, including subsumed clauses tagged as such, to a *deduction controller* (the master), which 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. Clauses generated by GLD may be forwarded to the HR process, but not vice versa: clauses derived by the HR or UR processes play the role of lemmas for the GLD process, and by sending only unit clauses to the latter, the deduction controller restricts lemmatization to unit lemmas. Furthermore, the deduction controller gives every process information on clauses subsumed by the other processes.

Another approach with a master-slave organization and heterogeneous inference systems was the Distributed Larch Prover (DLP) [95], a coarse-grain parallelization of the Larch Prover (LP) [61]. Since LP is *interactive*, a main motivation for the design of DLP was to enhance the user productivity by enabling the experimenter to launch and monitor multiple proof attempts in parallel. For this purpose, the master operates as a *coordinator*, which on one hand provides the user with a global interface to control the parallel proofs, and on the other hand is responsible for assigning to the slaves, called *workers*, the proof tasks specified by the user.

In summary, the master-slave philosophy is to *separate the control of parallelism and the control of deduction*: in most methods, each slave executes a sequential search plan to generate its derivation, and all other control issues (e.g., subdivision, communication, selection of "good" data, user interface) are dealt with in a centralized way by the master, which does not perform deductions. Thus, a *parallel search plan* for such strategies can be seen as *a collection of sequential search plans*, one per slave. If $\mathcal{C} = \langle I, \Sigma \rangle$ is a sequential strategy, and $\mathcal{C}' = \langle I, \Sigma' \rangle$ is a parallel-search master-slaves parallelization of $\mathcal{C}$, we have $\Sigma' = \langle \Sigma_1, \ldots, \Sigma_n \rangle$, if there are n slaves. In the case of distributed search with no multi-search, it is $\Sigma_1 = \cdots = \Sigma_n$, and the activities of the slaves are differentiated only by subdivision decided by the master.

6.2. Parallel search with peer processes

The master-slaves hierarchy is attractive, precisely because of the simplicity gained by separating the control of parallelism from the control of deduction. It worked well in PSATO [104], because there was no need for communication among slaves and the partition of the search tree was well-understood. In ordering-based theorem proving, the subdivision is much more difficult, because the search space is generally an infinite, highly redundant and *dynamic* graph (because of pruning by contraction [26]). Slaves need to be aware of the contractions made by other slaves, which means the master may become a *communication bottleneck*. If the master alone is responsible for contraction, it may become a *backward-contraction bottleneck*. Also in multi-search the master-slaves hierarchy induces some overhead (e.g., the periodical interruption of the slaves, and reconstruction of a common data base at the master in Team-Work).

For these reasons, more recent approaches to parallel search adopt peer processes. In multi-search, the search plan executed by each peer needs to control *communication and deduction*, and a *multi-search plan* is a collection of n such plans. In distributed search, the search plan executed by each peer needs to control also the *subdivision* of work.

6.2.1. Multi-search plan

A theorem proving strategy with parallel search features, in addition to the inference system I, a set M of *communication operators*, including at least *send* and *receive*. We define the communication operators in such a way that they are as similar as possibile to inference rules. First, assuming that all processes execute sound inferences, there is no difference, from a logical point of view, between clauses received from another process and clauses generated locally. Second, from an operational point of view, if process p_k sends a clause φ to process p_j and p_j receives it, the effect is that φ is *added* to the database of p_j, which is not so different from adding φ because it was deduced. Third, since we view communication as a responsibility of the search plan, we want to include communication steps in the derivation, and for this purpose we would like their effect to be as similar as possible to that of inferences.

Based on this motivation, we define *send* and *receive* as functions that return a set of clauses to be added and a set of clauses to be deleted (like the inference rules), where addition and deletion refer to the data base of the process executing the communication step. For the domain, the argument of *send* is the set of data being sent, so that the type is $send\colon \mathcal{P}(\mathcal{L}_\Theta) \to \mathcal{P}(\mathcal{L}_\Theta) \times \mathcal{P}(\mathcal{L}_\Theta)$. Since *send* does not modify the data base of the sender, we have that for all X, $send(X) = (\emptyset, \emptyset)$. If p_k executes a *send* at step i, defining $S^k_{i+1} = S^k_i \cup \pi_1(send(X)) - \pi_2(send(X))$ gives $S^k_{i+1} = S^k_i$ as desired.

Since the set of data being received is not known until after it has been received, we define $receive\colon \mathbb{N} \to \mathcal{P}(\mathcal{L}_\Theta) \times \mathcal{P}(\mathcal{L}_\Theta)$, where the argument is a natural number that may represent the identifier of a channel (similar to the identifiers of streams used to implement input/output in functional programming languages such as ML), or the address of a receiving buffer (similar to one of the arguments of the functions for send and receive in MPI [63]). If p_k executes a *receive* at step i, n is an identifier of a channel or buffer for p_k, and X is the set of data pending to be received through n, $receive(n) = (X, \emptyset)$, and defining $S^k_{i+1} = S^k_i \cup \pi_1(receive(X)) - \pi_2(receive(X))$ gives $S^k_{i+1} = S^k_i \cup X$ as desired.

Since the search plan is in charge of communication, the rule-selecting function ζ may select not only inference rules, but also communication operators, yielding $\zeta\colon States^* \to I \cup M$, or $\zeta\colon States^* \times \mathbb{N} \times \mathbb{N} \to I \cup M$, where the second and third arguments are the identifier of the process which is doing the selection and the number of processes, respectively.

Thus, a *search plan with communication* for a set I of inference rules and a set M of communication operators has the form $\Sigma = \langle \zeta, \xi, \omega \rangle$, where

- ζ: $States^* \times \mathbb{N} \times \mathbb{N} \to I \cup M$ selects an inference rule or a communication operator for the next step;
- ξ: $States^* \times \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\mathcal{L}_\Theta)$ selects a set of premises from the current state (e.g., $\xi((S_0, \ldots, S_i), n, k) \subseteq S_i$); and
- ω: $States \to Bool$ returns *true* if and only if the given state is successful.

A *multi-search plan*, or *multi-plan* for short, is a collection of search plans with communication, one per peer process: $\Sigma = \langle \Sigma_0, \ldots, \Sigma_{n-1} \rangle$.

6.2.2. Multi-search strategies with peer processes

Concrete instances of the above notions are offered by the successors of Team-Work (e.g., [40,55,41]), where the periodical reconstruction of a common data base was replaced by a communication scheme where every process may send clauses to another process during the derivation. A multi-plan for these strategies may combine premise-selecting functions ξ_i's that employ different goal-oriented heuristics (e.g., [43]), or different heuristics to decide which clauses deserve to be sent (e.g., TECHS [41] or CPTHEO [57,100]). Although mentioned in some papers, heuristics to decide which clauses to receive do not appear practical, because they require the receiver to use the received data to determine whether it is worth keeping it (e.g., a heuristic that rates an equation as "good" if it contracts many equations).

The method of [55], called *requirement-based cooperative theorem proving*, provides for two peer processes, one executing the SPASS prover [96], which implements ordering-based strategies for first-order logic with equality, and one executing DISCOUNT, the sequential base of the Team-Work prover for equational logic [5]. The two processes communicate by *expansion requests* (e.g., process p_0 sends to p_1 clause φ and p_1 replies by sending all resolvents between φ and the clauses in its `usable` list[9]), and *contraction requests* (e.g., p_0 sends to p_1 clause φ and p_1 replies by sending all its clauses that contract φ), which fit in our formal description of processes sending clauses. This system also features some distributed search in the form of problem decomposition by splitting clauses.

The TECHS system [40,41] aims at combining heterogeneous provers while minimizing changes to the provers themselves, to the extent that provers communicate by writing and reading files. TECHS combines SPASS and DISCOUNT with the model-elimination tableau-based prover SETHEO [76,82]. SETHEO and SPASS exchange subgoals (from SETHEO to SPASS) and lemmas (from SPASS to SETHEO), while SPASS and DISCOUNT exchange equations. Heuristics to select clauses to send include favoring unit lemmas and equations (from SPASS to SETHEO, because ordering-based provers handle equality better than tableau-based provers), and favoring short, general equations that one hopes may induce much contraction and little expansion (between SPASS and DISCOUNT). Ac-

[9] These provers adopted the terminology of Otter, with `sos` renamed `to-be-selected`.

cording to [40], SETHEO profites the most from the cooperation thanks to the lemmatization effect. TECHS is used within the ILF environment that integrates automated and interactive provers, model checkers, and tools for proof presentation [36].

A movement from fine or medium-grain to coarse-grain parallelism, and from master-slaves to peer processes, occurred also in subgoal-reduction strategies. The research begun with PARTHEO [88], which was based on OR-parallelism at the clause level[10], continued with SPTHEO [93], P-SETHEO [99] and CPTHEO [57,100], towards heterogeneous, multi-search systems. CPTHEO launches SETHEO and the resolution-based prover Delta, introduced as a preprocessor for SETHEO in [87], in parallel. Since clauses generated by Delta are used by SETHEO as *lemmas*, CPTHEO is also a descendant of HPDS [92], with the master-slaves structure of HPDS replaced by peer processes in CPTHEO. Assume that p_0 is a SETHEO process and p_1 a Delta process; p_0 sends to p_1 subgoals that it could not solve in the given resource (e.g., depth) limit of iterative deepening; p_1 replies by sending lemmas that unify with those subgoals; and p_0 restarts with its next round of iterative deepening. In a different scheme, p_0 sends to p_1 all open leaves in its current tableau, and p_1 replies by sending lemmas that appear "similar" to those leaves according to the criteria of [56]. In either case p_1 ranks and selects lemmas based on size (small term complexity is preferred), size of proof (lemmas whose proof was large are preferred, because they are like products with more added value, since more work was done by Delta to produce them), and the similarity-based criteria of [56].

As another example, the "nagging" technique[11] of [91] turned to emphasize peer processes and multi-search over time, making the slaves first-class deductive processes, which may complete the proof, rather than being used only for early detection of failures.

6.2.3. Distributed-search plan

A distributed-search plan needs an additional component to handle the subdivision. Since the search space of a theorem-proving problem is infinite and unknown, at each stage S_i of a derivation the search plan subdivides the inferences that can be done in S_i. Thus, the subdivision is built *dynamically* during the derivation. We reason that from the point of view of each process p_k, an inference is either *allowed* (it is assigned to p_k in the subdivision), or *forbidden* (it is assigned to others). Therefore, a distributed-search plan features a *subdivision function* $\alpha\colon \mathbb{N} \times I \times \mathcal{P}(\mathcal{L}_\Theta) \rightarrow Bool$, where $\alpha(k, f, X) = true/false$ means that p_k is allowed/forbidden to apply rule f to premises X. However, the subdivision function may keep the partial history of the derivation and the number of processes into account, yielding $\alpha\colon States^* \times \mathbb{N} \times \mathbb{N} \times I \times \mathcal{P}(\mathcal{L}_\Theta) \rightarrow Bool$. We require that α

[10] See Section 5.
[11] See Section 6.1.

is *total on generated clauses* (i.e., $\alpha((S_0,\ldots,S_i),n,k,f,X) \neq \perp$ if $X \subseteq \bigcup_{j=0}^{i} S_j$), and *monotonic*, in the sense of not changing the status of a step after it has been decided (i.e., $\alpha((S_0,\ldots,S_i),n,k,f,X) \sqsubseteq \alpha((S_0,\ldots,S_{i+1}),n,k,f,X)$ where $\sqsubset$ is the partial ordering $\perp \sqsubset false$ and $\perp \sqsubset true$).

Thus, a *distributed-search plan* for a set I of inference rules and a set M of communication operators has the form $\Sigma = \langle \zeta, \xi, \alpha, \omega \rangle$, where $\langle \zeta, \xi, \omega \rangle$ is a search plan with communication, and α is a subdivision function. This notion of distributed-search plan covers, for instance, the requirement-based theorem proving of [55], where inferences may be allowed/forbidden based on tags attached to the premises to signify that the clause belongs to a certain subproblem generated by splitting, and the Clause-Diffusion strategies of the next section.

6.2.4. Distributed-search strategies with peer processes

DARES [33] can be considered an early approach with distributed-search and peer processes. The main idea was to subdivide the problem among the processes, and let each process ask the others for more clauses, if it runs out of applicable resolution inferences, or is not making sufficient progress towards the proof according to heuristic measures. However, the method was described very informally, and especially the subdivision part was unclear. The issue of fairness of communication and subdivision, hence completeness of the distributed strategy, was not treated. Furthermore, DARES was designed for resolution with at most forward subsumption, at a time when it was already understood that resolution with both forward and backward subsumption is generally preferable, so that DARES avoided the difficulty of backward contraction, and parallelized a sub-optimal sequential strategy, against the principle of parallelizing the best existing sequential procedure.

The most long-lasting heritage of DARES has been the idea of demand-driven communication, or that a process should receive clauses only if it needs them. The critical point of this notion is what it means that a process “needs clauses”, and whether the notion of “need” and the ensuing communication are strong enough to ensure completeness in the presence of a subdivision of the data. Not surprisingly, this notion of demand-driven communication has had more fortune with multi-search strategies, such as those in Section 6.2.2, where completeness can be preserved easily by choosing fair sequential search plans.

The *Clause-Diffusion methodology* of [15,22,23] pioneered distributed search for contraction-based strategies. In addition to concrete strategies, it offered a framework where distributed strategies, distributed derivations and their properties, such as distributed fairness, were defined and studied formally. Clause-Diffusion was conceived as a methodology to transform a given sequential strategy $\mathcal{C} = \langle I, \Sigma \rangle$ into a distributed strategy $\mathcal{C}' = \langle I, \Sigma' \rangle$, in such a way that if $\mathcal{C}$ is complete, $\mathcal{C}'$ is also complete. A Clause-Diffusion strategy subdivides the search space by assigning generated clauses to processes and partitioning the inferences among the processes based on the ownership of clauses. For example, a process is allowed to do a paramodulation step only if it owns the clause paramodulated into. Unlike in a master-slave organization, where the master distributes the data to the slaves, the allocation of clauses to processes is achieved by having each process executing the same *allocation algorithm* for each clause it generates. The resulting partition of clauses is *logical*, not physical as in DARES, in the sense that every clause belongs to only one process, but may appear in the data base of many. To emphasize this point, the term *allocation algorithm* was replaced later by *subdivision criterion* [14].

In order to preserve completeness, the processes communicate clauses by messages, called *inference messages*, because receiving a clause triggers the allowed inferences. Sufficient conditions for fairness, hence completeness, include broadcasting all persistent non-redundant clauses[12]. As soon as a process finds a proof, it broadcasts a halting message so that all processes terminate.

While it applies to ordering-based strategies in general, the Clause-Diffusion methodology was designed having contraction-based strategies for logics with equality in mind. This brought the problem of the parallelization of contraction to the forefront. It was in this context that the *backward-contraction bottleneck* was identified, contributing to the choice (and the definition) of parallelism at the search level. This led in turn to defining the problem of *distributed global contraction*, or how to keep a distributed data base inter-reduced. Several *distributed global contraction schemes* were proposed in [15,22]. The scheme eventually implemented in both *Aquarius* [23] and *Peers* [27] was the simplest: every process retains the received inference messages so that its local data base mirrors the global one, and keeps its data base inter-reduced[13]. While the main idea was distributed search, Clause-Diffusion also allowed processes to apply different search plans (e.g., [23]).

Clause-Diffusion evolved into *Modified Clause-Diffusion* [16], which improved subdivision of inferences, communication and proof reconstruction. Clause-Diffusion subdivided only expansion inferences, whereas Modified Clause-Diffusion subdivides also backward-simplification inferences, by establishing that any process can delete a reducible clause φ, but only the owner of φ can generate its reduced form φ'. If φ' is persistent, it will be broadcast and the other processes will receive it. In this way the subdivision of inferences is finer, without preventing or delaying the deletion of redundant clauses. For communication, Clause-Diffusion featured, in addition to inference messages, messages to bring clauses to their owners. In Modified Clause-Diffusion also this purpose is achieved by inference messages, reducing both types and number of messages. This is possible because clauses are broadcast as inference messages not when selected for

[12] This is for *distributed uniform fairness* [22,16]; if the fairness condition were weakened, the communication requirement would be weakened also; see [24] for a study of non-uniform fairness.

[13] This corresponds to the "localized image sets" with "direct contraction" scheme of [15,22].

inferences, as in Clause-Diffusion, but right after generation and forward contraction. This, together with better *naming schemes* (i.e., the rules to assign identifiers to clauses across the distributed data base) contributed to achieving the property of *distributed proof reconstruction*: the successful process can reconstruct the proof consulting only its data base.

Modified Clause-Diffusion was implemented in *Peers-mcd* [17], with the Argonne prover EQP [80] as sequential base. Peers-mcd implements the *ancestor-graph oriented* (AGO) criteria to assign clauses to processes [14,18]. An issue with the subdivision of the search space is to prevent the search spaces explored by the different processes from *overlapping* too much. Some overlap is unavoidable in asynchronous distributed search: even if every inference is assigned to only one process, there may be duplicate inferences, because the processes can generate in different ways distinct variants of the same clauses, producing distinct variants of the same logical inference, before the variants are eliminated by distributed global contraction. Thus, the AGO criteria are heuristics that aim at limiting the overlap, and are among the features that allowed Peers-mcd to generate the first distributed proof of the Robbins theorem, as shown in the next section.

7. First distributed proof of the Robbins theorem

The problem of proving that Robbins algebras are Boolean dates back to 1933, when E.V. Huntington demonstrated [68,69] that the equation

$$n(n(x)+y)+n(n(x)+n(y))=x \quad (H)$$

is sufficient to present Boolean algebra, together with associativity and commutativity of +. Herbert Robbins conjectured that the equation

$$n(n(x+y)+n(x+n(y)))=x \quad (R)$$

is also sufficient. Since a proof of the conjecture was not found, an algebra defined by equation (R) with associativity and commutativity of + was called a *Robbins algebra*. Equations (H) and (R) were called *Huntington axiom* and *Robbins axiom*, respectively. The problem of determining whether a Robbins algebra is Boolean remained open, and eventually became known in the theorem proving community as the *Robbins problem* [101].

In the early 90's, Steve Winker proved by hand that each of the following two conditions:

$$\exists x \exists y\ x+y=x \quad (FWC)$$

$$\exists x \exists y\ n(x+y)=n(x) \quad (SWC)$$

termed First Winker Condition and Second Winker Condition, respectively, in [80], is sufficient to make a Robbins algebra Boolean [97,98], but such lemmas remained beyond the possibilities of automated theorem provers.

In 1996 the automated prover EQP of William McCune proved that Robbins algebras are Boolean, as reported in [81]. The proof was obtained by showing that:

- The First Winker Condition implies the Huntington axiom ($FWC \Rightarrow H$).
- The Second Winker Condition implies the First Winker Condition ($SWC \Rightarrow FWC$).
- The Robbins axiom implies the Second Winker Condition ($R \Rightarrow SWC$).

This mechanical proof has been explained in mathematical terms in [35].

In the following we compare EQP (version 0.9d) and Peers-mcd (a version of April 1999 based on EQP0.9d) on the three parts of the proof, on workstations HP series C360 with 1 G of memory. In Table 3, speed-up is the ratio t/t' if t is the CPU time of EQP and t' the CPU time of the peer that finds the proof. Efficiency is the ratio $t/(t' \times n)$, where n is the number of machines (i.e., workstations), hence peer processes, one per machine, used by Peers-mcd.

Lemma	*EQP*	*Peers-mcd*	*Speed-up*	*Efficiency*
$FWC \Rightarrow H$	294 (4 min 54 s)	73 (1 min 13 s)	4	2
$SWC \Rightarrow FWC$	85,890 (23 h 51 min 30 s)	23,713 (6 h 35 min 13 s)	3.62	1.81
$R \Rightarrow SWC$	64,210 (17 h 50 min 10 s)	34,792 (9 h 39 min 52 s)	1.85	0.92

Table 3
CPU times (in seconds) of EQP and Peers-mcd with 2 processes.

Since Peers-mcd used 2 processes, the speed-up was super-linear in the first two lemmas and almost linear in the third. Peers-mcd can prove $SWC \Rightarrow FWC$ in 16,928 s with 4 processes (speed-up = 5, efficiency = 1.25) and in only 3,720 s with 6 processes (speed-up = 23, efficiency = 3.83). Using more processes did not improve the performance for the other two lemmas.

We emphasize that the proofs found by EQP were *not* used as a guidance for Peers-mcd: the two provers were given the same input equations on each lemma. In terms of the applied strategies, the inference system included AC-paramodulation, AC-simplification, subsumption, and deletion by weight, with parameter *max-weight* equal to 30, 34, and 50, in the three proofs, respectively, for both provers. EQP used basic paramodulation [7], for $FWC \Rightarrow H$ and $R \Rightarrow SWC$, and plain paramodulation for $SWC \Rightarrow FWC$. This is the best choice for EQP. McCune also used basic paramodulation for $FWC \Rightarrow H$ [80], plain paramodulation for $SWC \Rightarrow FWC$ [80], and basic paramodulation for

$R \Rightarrow SWC$ [81]. Peers-mcd used plain paramodulation for $FWC \Rightarrow H$ and $SWC \Rightarrow FWC$, and basic paramodulation for $R \Rightarrow SWC$. Thus, the only difference in the choice of inference system for the results in Table 3 is that EQP used basic paramodulation, and Peers-mcd did not, on $FWC \Rightarrow H$, the easiest of the three lemmas.

We also ran Peers-mcd on $FWC \Rightarrow H$ with basic paramodulation: with 2 processes, it obtained a proof in 200 s (speed-up = 294/200 = 1.47, efficiency = 0.73). Symmetrically, we ran EQP on $FWC \Rightarrow H$ with plain paramodulation, which took 614 s, so that the speed-up of Peers-mcd was actually 614/73=8.41, with an efficiency of 4.2.

The search plan was the *pair algorithm* [81] for all the experiments by both provers. Pairs of equations were selected by *best-first search* with the length of the pair as heuristic evaluation function, but in the proof of $R \Rightarrow SWC$ both provers assigned value 4 to the parameter *pick-given-ratio*, which means that the search plan selected the oldest candidate pair every 4 choices. The proof of $R \Rightarrow SWC$ was obtained by both provers by assigning value 0 to the parameter *AC-superset-limit*, which implies the highest degree of pruning of AC-unifiers.

Peers-mcd used the AGO criteria *majority*, in the proofs of $FWC \Rightarrow H$ and $SWC \Rightarrow FWC$, and *all-parents* in the proof of $R \Rightarrow SWC$. The majority criterion assigns an equation φ to the process that owns a majority of its ancestors, with ties broken arbitrarily. The rationale of this criterion is that such process is already most active in the area of the search space where φ appears, and assigning φ to others could augment the overlap. The all-parents criterion assigns φ to the process numbered $id(\psi_1) + id(\psi_2) \ mod \ n$, if φ was generated by paramodulation from ψ_1 and ψ_2, to the process numbered $id(\psi) \ mod \ n$, if φ was generated by backward contraction of ψ, where n is the number of processes and $id(\psi)$ is the identifier of ψ. The intuition behind this criterion is that clauses which have the same parents are spatially close in the search graph, and therefore should be assigned to the same process to limit the overlap. More details on these criteria can be found in [14].

8. Discussion

While parallel automated theorem proving is still a young field, several approaches to parallelization have been tried for various classes of strategies: Figure 3 summarizes the state of the art by combining Figures 1 and 2, with dotted lines linking types of parallelism and classes of strategies they have been applied to.

Reading the dotted lines in Figure 3 from top to bottom and from left to right, we see that parallelism at the term level was applied to parallel term rewriting (e.g., [62,48,73,71,1,2]), and to contraction-based strategies (e.g., [29,31,72,86]). AND-parallelism and OR-parallelism were applied to subgoal-

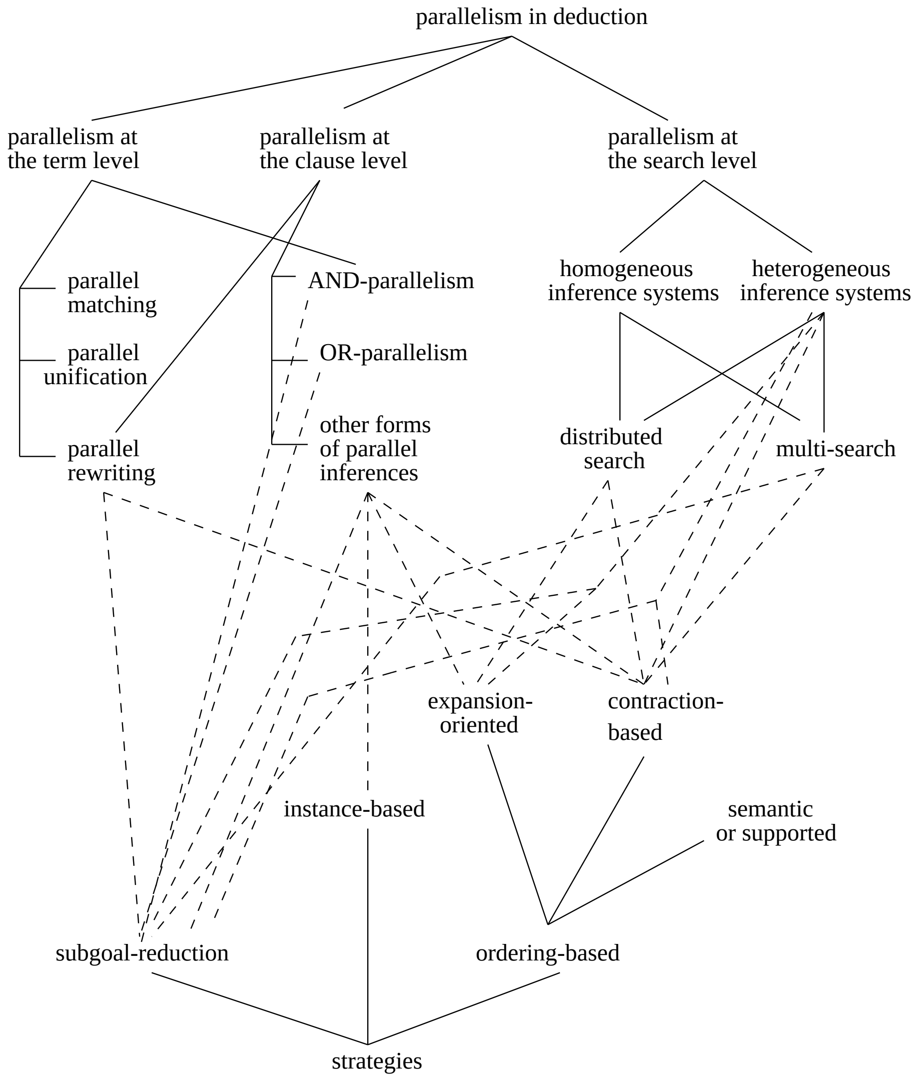

Figure 3. Matching parallelization principles and classes of strategies.

reduction strategies, in the context of PTTP (e.g., [28,3] for OR-parallelism, [91] for AND-parallelism) and tableau-based strategies (e.g., [88]). Other forms of parallelism at the clause level were applied to instance-based strategies (e.g., [102]), expansion-oriented strategies (e.g., [70]), contraction-based strategies (e.g., [77,78,103]), and other subgoal-reduction-style strategies, such as the higher-order analytic tableaux of [74] and the Gentzen proofs of [65]. Distributed search with homogeneous inference systems was applied to expansion-oriented strategies (e.g., [33,104]), and contraction-based strategies (e.g., [22,23,27,16]). Multi-search with homogeneous inference systems was applied to subgoal-reduction strategies (e.g., [91]) and contraction-based strategies (e.g., [4,5,46,44,42]). Heterogeneous systems, possibly with multi-search, were obtained by combining subgoal-reduction strategies with expansion-oriented strategies (e.g., [92,57,100]), subgoal-reduction strategies with contraction-based strategies (e.g., [40,41]), and different contraction-based strategies (e.g., [95,55]).

This study is dedicated to parallel fully-automated theorem proving, and does not cover other subfields of automated deduction, such as interactive proof assistants and model generation systems. It is important, however, that parallelism has been applied also in those domains, e.g., [95,83,65] for the former and [58,64,104] for the latter. We also refer to [67,66] for the neighbour field of parallel symbolic computation.

8.1. Summary

In this paper, we have analyzed the impact of parallelization on the *control of search*. We observed that since parallelism at the term level is *below* the level where the search plan makes decisions, approaches with parallelism at the term level tend to replace the search plan by low-level data-driven forms of concurrency (e.g., [72,86,52]), or produce strategy-compliant parallelizations (e.g., [31,29]). The potential problem is a loss of control for the former, and an excess of control for the latter. Data-driven concurrency may be appropriate for ground computations that are guaranteed to converge (e.g., computing a congruence closure for ground completion), but may represent a counter-productive loss of control in general theorem proving, where the essence, from a practical point of view, is not saturation (i.e., do all the steps, with the order being a secondary issue), but effective search (i.e., find a good order to do the steps in order to avoid doing them all). Strategy-compliant parallelizations, on the other hand, may be too conservative: they avoid the risk of mixing search with parallelism, but they renounce using parallelism to try to generate better searches.

For parallelism at the clause level, we noted that since it is precisely at the level of the inferences, it turns the search plan into a *scheduler* of parallel inferences, which assign inferences – viewed as *tasks* – to a pool of parallel processes. One reason why this type of approach was appealing is that it allowed one to apply to theorem proving scheduling techniques (e.g., *task stealing*) defined for generic scheduling problems. However, such general techniques may not take specific theorem-proving knowledge into account (e.g., the differences between expansion tasks and contraction tasks). More importantly, the problem is granularity, that is, whether processing a given-clause (e.g., [77]) or a subgoal (e.g., [28,88,3]) is a sufficiently large task. Such tasks are likely to be too small with respect to the amount of work required by the theorem-proving problem, so that too much time is spent in task scheduling and not in inference making. Furthermore, this problem becomes worse as the difficulty of the theorem-proving problem grows, against the expectation that parallel theorem proving makes a difference precisely on the hardest searches (e.g., the more clauses or subgoals the problem requires us to handle, the smaller is the task of handling one of them).

This difficulty seems less severe in parallel hyperlinking [102], because a derivation by this method is broken into hyperlinking rounds interleaved with propositional unsatisfiability tests, and the task of taking care of a clause or literal may not be too small with respect to a single hyperlinking round. However, another problem is that the process in charge of scheduling and updating the shared data base may become a bottleneck.

For parallelism at the search level, we note from Figure 3 that while both distributed-search and multi-search ordering-based strategies exist, distributed search does not seem to have been applied to subgoal-reduction strategies. A possible explanation lies in the differences between the kinds of control most commonly adopted for the two classes of strategies, as discussed in Section 2. Ordering-based strategies work with a set of objects, and build many proof attempts implicitly; therefore, it is quite natural to think of subdividing the set of objects, or, better, the inferences they permit, in order to subdivide the search space, hence the proof attempts. On the other hand, subgoal-reduction strategies work on one goal object at a time, building a proof attempt at a time, so that there is no room for a coarse-grain subdivision that subdivides the proof attempts. If one applies the idea of subdivision within the single proof attempt, it may fall back on OR-parallelism or other forms of parallelism at the clause level, whose granularity is too small for theorem proving.

The application of distributed search to subgoal-reduction strategies would require us to design distributed-search plans for such strategies. A distributed-search subgoal-reduction strategy should feature parallel processes, each of whom develops a derivation, hence a sequence of proof attempts (e.g., tableaux), and whose activities are differentiated by a distributed-search plan with a subdivision function that subdivides the inferences, hence the tableaux, among the processes. Communication in such a method could consist of exchanging lemmas: envision two processes p_i and p_k such that a sub-tableau $\mathcal{X}$ with root labelled by literal A is forbidden for p_i and allowed for p_k, because the inference that extends A is assigned to p_i by the subdivision function. If $\mathcal{X}$ fails to close, it is safe for p_i to ignore it; if p_k succeeds in closing $\mathcal{X}$, p_k sends to p_i the corresponding lemma[14] $\neg A$. If it comes to communication of lemmas, however, the question remains of whether having multiple subgoal-reduction processes exchanging lemmas may pay off, or whether it may be better to resort to heterogeneous systems, having ordering-based components generating lemmas for the subgoal-reduction components (e.g., [57,40,41]).

Furthermore, it seems that in order to be fair a distributed-search subgoal-reduction strategy would need to generate and accumulate tableaux, that is, it

[14] All lemmas are unit lemmas in Horn logic, in first-order logic closing $\mathcal{X}$ may correspond to proving a lemma $\neg A \vee C$, where C is a disjunction of literals.

should adopt a best-first, rather than depth-first, plan. A similar experience was made in [9] for contraction, where the idea of applying subsumption among tableaux assume that more than one tableau is generated and kept.

We recall that the analysis of [21] suggested that while subgoal-reduction strategies may be amenable to all three types of parallelism (term-level, clause-level and search-level), parallelism at the term level is too fine-grained for expansion-oriented strategies, and both parallelism at the term level and parallelism at the clause level are too fine-grained for contraction-based strategies. The study of how parallelization affects the search plan in this paper suggests that the granularity of parallelism at the term and clause level may be too small also for subgoal-reduction strategies. Moreover, the consideration of the nature of the respective searches in this paper leads us to conjecture that while both distributed search and multi-search are applicable to ordering-based strategies, only the latter may be suitable for subgoal-reduction strategies with a depth-first search plan that tries one proof attempt at a time.

The activity in the field since 1992, when the material in [21] was first written, shows a movement from fine or medium-grain parallelism towards coarse-grain parallelism, and from master-slaves hierarchies to peer processes (e.g., PaReDux, from fine-grain parallelism [29,31] to an approach à la Team-Work [30], the parallelization of SETHEO [76,82] from OR-parallelism at the clause level [88] to parallel search and cooperation of theorem provers [93,99,57,100]). Also research programs that were aiming at parallel search since their inception have evolved in this direction further (e.g., Team-Work, from a master-slave organization [4] to peer processes [55], heterogenous systems, from master-slaves [92] to peer processes [57,40,41], and the evolution from Clause-Diffusion [22] to Modified Clause-Diffusion [16]).

8.2. Directions for future work

While it appears more suitable for theorem proving than finer forms of parallelism, parallel search is not free of obstacles. Problems include the *cost of communication*, the *overlap* of the parallel searches, and the *scalability*.

In distributed search, communication is required for completeness, leading to search plans that eventually broadcast all persistent non-redundant clauses. This may be a high amount of communication, although not as high as in approaches that combine distributed memory with finer granularities and need message-passing also to achieve individual inferences. In multi-search and combinations of theorem provers, it is difficult to design heuristics that are effective in determining which data are "good", and therefore should be communicated. If the heuristics are relaxed, communication may become too intense; if they are strict, too little may be communicated, so that it does not make a difference and the processes ignore each other; if the heuristics are complex, they may introduce too much overhead. In combinations of theorem provers the problem is compounded by the usage of different inference systems and even different logics.

If the parallel processes end up exploring *overlapping* areas of the search space, their efforts are partly wasted. In distributed search, it is impossible to partition the search space into disjoint parts, and in practice it is hard even to *minimize the overlap*. In multi-search, the overlap may be an even more serious problem, because the processes visit the same search space, only in a different order. The search plans may not be sufficiently different, or seemingly different search plans may generate very similar searches on some inputs. Part of the problem is that most known fair search plans are exhaustive, and it may be rare to get significantly different searches from plans that are all exhaustive in nature.

Scalability is difficult in parallel search, because the addition of a new process may dramatically change the searches of the others. In distributed search, one would expect that if we add more processes the performance improves, because each process should get a smaller portion of space to search. However, this is not guaranteed to happen, because increasing the number of processes changes the subdivision not only quantitatively, but also qualitatively. The search space allowed to process p_k in a search with $2n$ processes may be radically different than the search space allowed to p_k in a search with n processes, and the performance is not guaranteed to improve, because the subdivision with $2n$ processes may be worse from the point of view of finding a proof (e.g., it may break the search space in a way that delays p_k in generating the proof found with n processes). In multi-search, one expects that a process p_i, executing plan Σ_i, takes advantage of receiving from a process p_j, executing plan Σ_j, data that Σ_i would only generate later. However, it may also happen that receiving data from p_j forces p_i to consider data that does not help to find a proof sooner than p_i alone would, so that the performance does not improve. In heterogeneous systems, enriching the pool with an additional inference system may not help, if the added rules are not useful for the problem at hand.

For parallel search to be beneficial, however, it may not be necessary to limit communication and overlap globally: for instance, in the experiments on the Robbins problem reported in Section 7, each parallel process generates far fewer clauses than the sequential process. This may indicate that the subdivision induced by the AGO criteria is effective, and enables the parallel process that finds the proof to ignore many clauses that may not be redundant with respect to all proofs, but are redundant with respect to the generated proof. In such experiments, the lack of scalability may be the price to pay for the super-linear speed-up.

Many directions for future work are open. In addition to continuing addressing the problems above, Figure 3 shows that not much work has been done in combining parallel search with *target-oriented* and *semantic* strategies. For instance, one may envision combining parallelism with semantic information, such as parallel processes reasoning under different interpretations.

A formal analysis weighting the disadvantages of *communication* and *overlap* with the advantages of the subdivision of search in *distributed-search contraction-based strategies* was begun in [19], applying the *bounded search spaces methodology* of [26]. That analysis tried to determine whether distributed search may make the bounded search space *smaller* by doing at least as much contraction as the sequential process and adding the effect of the subdivision. However, distributed search may take advantage of performing steps in different order, especially contraction steps, hence producing *different* search spaces and different proofs. Thus, a direction for further research is to analyze this *reordering of the search* in both distributed search and multi-search.

Acknowledgements

I wish to thank Bernhard Gramlich, Hélène Kirchner and Frank Pfenning, for their invitation to give a talk at the Third Workshop on Strategies in Automated Deduction, which gave me the opportunity to write this paper; the KI Group of the Fakultät Informatik of the Technische Universität Dresden, and the KI Group of the Fachbereich Informatik of the Universität Koblenz-Landau, where I worked on this paper; Peter Baumgartner and Ulrich Furbach for bringing to my attention proof confluence as a criterion of strategy classification; and the anonymous referees for their suggestions and remarks that allowed me to improve the paper significantly.

References

[1] Iliès Alouini. Concurrent garbage collector for concurrent rewriting. In Jieh Hsiang, editor, *Proceedings of the 6th RTA*, volume 914 of *LNCS*, pages 132–146. Springer Verlag, 1995.

[2] 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.

[3] 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*. Kluwer Academic, 1991.

[4] Jürgen Avenhaus and Jörg Denzinger. Distributing equational theorem proving. In Claude Kirchner, editor, *Proceedings of the 5th RTA*, volume 690 of *LNCS*, pages 62–76. Springer Verlag, 1993.

[5] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: a system for distributed equational deduction. In Jieh Hsiang, editor, *Proceedings of the 6th RTA*, volume 914 of *LNCS*, pages 397–402. Springer Verlag, 1995.

[6] Leo Bachmair and Harald Ganzinger. A theory of resolution. Technical Report MPI-I-97-2-005, Max Planck Institut für Informatik, 1997. To appear in J. Alan Robinson and Andrei Voronkov, eds., *Handbook of Automated Reasoning.*

[7] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[8] Peter Baumgartner. Hyper tableaux — the next generation. In Harrie de Swart, editor, *Proceedings of TABLEAUX-98*, volume 1397 of *LNAI*, pages 60–76. Springer, 1998.

[9] Peter Baumgartner and Stefan Brüning. A disjunctive positive refinement of model elimination and its application to subsumption deletion. *Journal of Automated Reasoning*, 19:205–262, 1997.

[10] Peter Baumgartner, Norbert Eisinger, and Ulrich Furbach. A confluent connection calculus. In Harald Ganzinger, editor, *Proceedings of the 16th CADE*, volume 1632 of *LNAI*, pages 329–343. Springer, 1999.

[11] Wolfgang Bibel and Elmer Eder. Methods and calculi for deduction. Pages 68–183 in Vol. 1 of [59].

[12] Wolgang Bibel. *Automated Theorem Proving.* Friedr. Vieweg & Sohn, 2nd edition, 1987.

[13] Jean-Paul Billon. The disconnection method. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Proceedings of TABLEAUX-96*, volume 1071 of *LNAI*, pages 110–126. Springer, 1996.

[14] Maria Paola Bonacina. Experiments with subdivision of search in distributed theorem proving. Pages 88–100 in [66].

[15] Maria Paola Bonacina. *Distributed automated deduction.* PhD thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.

[16] Maria Paola Bonacina. On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method. *Journal of Symbolic Computation*, 21:507–522, 1996.

[17] Maria Paola Bonacina. The Clause-Diffusion theorem prover Peers-mcd. In William W. McCune, editor, *Proceedings of the 14th CADE*, volume 1249 of *LNAI*, pages 53–56. Springer, 1997.

[18] Maria Paola Bonacina. Mechanical proofs of the Levi commutator problem. In Peter Baumgartner et al., editor, *Notes of the CADE-15 Workshop on Problem Solving Methodologies with Automated Deduction*, pages 1–10, 1998.

[19] 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.

[20] Maria Paola Bonacina. A taxonomy of theorem-proving strategies. In Michael J. Wooldridge and Manuela Veloso, editors, *Artificial Intelligence Today*, volume 1600 of *LNAI*, pages 43–84. Springer, 1999.

[21] Maria Paola Bonacina and Jieh Hsiang. Parallelization of deduction strategies: an analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.

[22] Maria Paola Bonacina and Jieh Hsiang. The Clause-Diffusion methodology for distributed deduction. *Fundamenta Informaticae*, 24:177–207, 1995.

[23] 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.

[24] Maria Paola Bonacina and Jieh Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146:199–242, 1995.

[25] 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.

[26] 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.

[27] Maria Paola Bonacina and William W. McCune. Distributed theorem proving by Peers. In Alan Bundy, editor, *Proceedings of the 12th CADE*, volume 814 of *LNAI*, pages 841–845. Springer Verlag, 1994.

[28] 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.

[29] Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. Parallel ReDuX $\rightarrow$ PaReDuX.

In Jieh Hsiang, editor, *Proceedings of the 6th RTA*, volume 914 of *LNCS*, pages 408–413. Springer Verlag, 1995.

[30] Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. A master-slave approach to parallel term-rewriting on a hierarchical multiprocessor. In Jacques Calmet and Carla Limongelli, editors, *Proceedings of the 4th DISCO*, volume 1128 of *LNCS*, pages 184–194. Springer Verlag, 1996.

[31] Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin. Strategy-compliant multi-threaded term completion. *Journal of Symbolic Computation*, 21(4–6):475–506, 1996.

[32] Ricardo Caferra and N. Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13:613–641, 1992.

[33] Susan E. Conry, D. J. MacIntosh, and R. A. Meyer. DARES: a Distributed Automated REasoning System. In *Proceedings of the 11th AAAI*, pages 78–85, 1990.

[34] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, Eds. *Handbook of Tableau Methods*. Kluwer, 1998.

[35] Bernd Ingo Dahn. Robbins algebras are Boolean: a revision of McCune's computer-generated solution of Robbins problem. *Journal of Algebra*, 208:526–532, 1998.

[36] Bernd Ingo Dahn, J. Gehne, Th. Honigmann, and Andreas Wolf. Integration of automated and interactive theorem proving in ILF. In William W. McCune, editor, *Proceedings of the 14th CADE*, volume 1249 of *LNAI*, pages 57–60. Springer, 1997.

[37] Martin Davis, G. Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[38] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[39] Jörg Denzinger. *Teamwork: a method to design distributed knowledge based theorem provers*. PhD thesis, Universität Kaiserslautern, 1993.

[40] Jörg Denzinger and Bernd Ingo Dahn. Cooperating theorem provers. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume 2. Kluwer Academic, 1998.

[41] Jörg Denzinger and Dirk Fuchs. Cooperation of heterogeneous provers. In *Proceedings of IJCAI-99*, pages 10–15. Morgan Kaufmann, 1999.

[42] Jörg Denzinger, Marc Fuchs, and Matthias Fuchs. High performance ATP systems by combining several AI methods. In *Proceedings of IJCAI-97*, pages 102–107. Morgan Kaufmann, 1997.

[43] Jörg Denzinger and Matthias Fuchs. Goal-oriented equational theorem proving using Team-Work. In *Proceedings of the 18th KI*, volume 861 of *LNAI*, pages 343–354. Springer, 1994.

[44] Jörg Denzinger and Martin Kronenburg. Planning for distributed theorem proving: the teamwork approach. In Steffen Hölldobler, editor, *Proceedings of the 20th KI*, volume 1137 of *LNAI*, pages 43–56. Springer, 1996.

[45] Jörg Denzinger and Jürgen Lind. Twlib: a library for distributed search applications. In Chu-Sing Yang, editor, *Proceedings of ICS-96*, pages 101–108, 1996.

[46] Jörg Denzinger and Stephan Schulz. Recording and analyzing knowledge-based distributed deduction processes. *Journal of Symbolic Computation*, 21(4–6):523–541, 1996.

[47] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[48] Nachum Dershowitz and Naomi Lindenstrauss. An abstract concurrent machine for rewriting. In Hélène Kirchner and W. Wechler, editors, *Proceedings of the 2nd ALP*, volume 463 of *LNCS*, pages 318–331. Springer Verlag, 1990.

[49] Norbert Eisinger and Hans Jürgen Ohlbach. Deduction systems based on resolution. Pages 184–273 in Vol. 1 of [59].

[50] R. Engelmore and T. Morgan, Eds. *Blackboard Systems.* Addison Wesley, 1988.

[51] Robert L. Constable et al. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice Hall, 1986.

[52] 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, 1997.

[53] Melvin Fitting. *First-order Logic and Automated Theorem Proving.* Springer, 1990.

[54] Bertram Fronhöfer and Graham Wrightson, Eds. *Parallelization in Inference Systems.* Number 590 in LNAI. Springer-Verlag, 1990.

[55] Dirk Fuchs. Requirement-based cooperative theorem proving. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *Proceedings of the 6th JELIA*, volume 1489 of *LNAI*, pages 139–153. Springer, 1998.

[56] Marc Fuchs. Controlled use of clausal lemmas in connection tableau calculi. *Journal of Symbolic Computation*, 29(2):299–341, 2000.

[57] Marc Fuchs and Andreas Wolf. Cooperation in model elimination: CPTHEO. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th CADE*, volume 1421 of *LNAI*, pages 42–46. Springer, 1998.

[58] M. Fujita, Ryuzo Hasegawa, Miyuki Koshimura, and H. Fujita. Model generation theorem provers on a parallel inference machine. In *Proceedings of FGCS-92*, pages 357–375, 1992.

[59] Dov M. Gabbay, Christopher J. Hogger, and J. Alan Robinson, Eds. *Handbook of Logic in Artificial Intelligence and Logic Programming (Vol. 1 & 2).* Oxford University Press, 1993.

[60] Jean Gallier, Paliath Narendran, David A. Plaisted, Stan Raatz, and Wayne Snyder. Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. *Journal of the ACM*, 40(1):1–16, 1993.

[61] Stephen J. Garland and John V. Guttag. An overview of LP. In Nachum Dershowitz, editor, *Proceedings of the 3rd RTA*, volume 355 of *LNCS*, pages 137–151. Springer Verlag, 1989.

[62] 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, March 1989.

[63] William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report 96/6, MCS Division, Argonne National Laboratory, 1996.

[64] Ryuzo Hasegawa and Miyuki Koshimura. An AND parallelization method for MGTP and its evaluation. Pages 194–203 in [67].

[65] Jason Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Proceedings of the 16th CADE*, volume 1632 of *LNAI*. Springer, 1999.

[66] Markus Hitz and Erich Kaltofen, Eds. *Proceedings of the 2nd PASCO.* ACM Press, 1997.

[67] Hoon Hong, Ed. *Proceedings of the 1st PASCO*, volume 5 of *Lecture Notes Series in Computing.* World Scientific, 1994.

[68] E. V. Huntington. Boolean algebra: A correction. *Transactions of the AMS*, 35:557–558, 1933.

[69] E. V. Huntington. New sets of independent postulates for the algebra of logic. *Transactions of the AMS*, 35:274–304, 1933.

[70] A. Jindal, Ross Overbeek, and W. Kabat. Exploitation of parallel processing for implementing high-performance deduction systems. *Journal of Automated Reasoning*, 8:23–38, 1992.

[71] Owen Kaser, Shaunak Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar.

Fast parallel implementations of lazy languages – the EQUALS experience. In *Proceedings of the ACM Conf. on LISP and Functional Programming*, pages 335–344, 1992.

[72] Claude Kirchner, Christopher Lynch, and Christelle Scharff. Fine-grained concurrent completion. In Harald Ganzinger, editor, *Proceedings of the 7th RTA*, volume 1103 of *LNCS*, pages 3–17. Springer Verlag, 1996.

[73] Claude Kirchner and Patrick Viry. Implementing parallel rewriting. Pages 123–138 in [54].

[74] Karsten Konrad. HOT: a concurrent automated theorem prover based on higher-order tableaux. In J. Grundy and M. Newey, editors, *Proceedings of TPHOLs*, volume 1479 of *LNCS*, pages 245–262. Springer Verlag, 1998. Also: SEKI Report SR-98-03, Fachbereich Informatik, Universität des Saarlandes.

[75] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyperlinking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.

[76] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. SETHEO: a high performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[77] Ewing L. Lusk and William W. McCune. Experiments with ROO: a parallel automated deduction system. Pages 139–162 in [54].

[78] Ewing L. Lusk, William W. McCune, and John Slaney. ROO: a parallel theorem prover. In Deepak Kapur, editor, *Proceedings of the 11th CADE*, volume 607 of *LNAI*, pages 731–734. Springer Verlag, 1992.

[79] William W. McCune. Otter 3.0 reference manual and guide. Technical Report 94/6, MCS Division, Argonne National Laboratory, 1994.

[80] 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, 1997.

[81] William W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[82] M. Moser, O. Ibens, Reinhold Letz, J. Steinbach, C. Goller, Johann Schumann, and K. Mayr. The model elimination provers SETHEO and E-SETHEO. *Journal of Automated Reasoning*, 18(2), 1997.

[83] Roderick Moten. Exploiting parallelism in interactive theorem provers. In J. Grundy and M. Newey, editors, *Proceedings of TPHOLs*, volume 1479 of *LNCS*, pages 315–330. Springer Verlag, 1998.

[84] David A. Plaisted. Equational reasoning and term rewriting systems. Pages 273–364 in Vol. 1 of [59].

[85] David A. Plaisted. Mechanical theorem proving. In Ranan B. Banerji, editor, *Formal Techniques in Artificial Intelligence*. Elsevier, 1990.

[86] Christelle Scharff. *Deduction avec contraintes et simplification dans les theories equationnelles.* PhD thesis, Université Henri Poincaré Nancy 1, September 1999.

[87] Johann Schumann. Delta: a bottom-up pre-processor for top-down theorem provers. In Alan Bundy, editor, *Proceedings of the 12th CADE*, volume 814 of *LNAI*, pages 774–777. Springer Verlag, 1994.

[88] Johann Schumann and Reinhold Letz. PARTHEO: a high-performance parallel theorem prover. In Mark E. Stickel, editor, *Proceedings of the 10th CADE*, volume 449 of *LNAI*, pages 28–39. Springer Verlag, 1990.

[89] Raymond M. Smullyan. *First-Order Logic.* Dover, 1995. (Republication of the work first published as "Band 43" Series *Ergebnisse der Mathematik und ihrer Grenzgebiete*, Springer Verlag, 1968).

[90] Rolf Socher-Ambrosius and Patricia Johann. *Deduction systems.* Springer, 1997.

[91] 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.

[92] 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.

[93] Christian B. Suttner. SPTHEO: a parallel theorem prover. *Journal of Automated Reasoning*, 18:253–258, 1997.

[94] Christian B. Suttner and Johann Schumann. Parallel automated theorem proving. In L. Kanal et al., editor, *Parallel Processing for Artificial Intelligence.* Elsevier, 1994.

[95] Mark T. Vandevoorde and Deepak Kapur. Distributed Larch prover (DLP): an experiment in parallelizing a rewrite-rule based prover. In Harald Ganzinger, editor, *Proceedings of the 7th RTA*, volume 1103 of *LNCS*. Springer, 1996.

[96] Christoph Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER. In Michael McRobbie and John Slaney, editors, *Proceedings of the 13th CADE*, volume 1104 of *LNAI*, pages 141–145. Springer, 1996.

[97] Steve Winker. Robbins algebra: conditions that make a near-Boolean algebra Boolean. *Journal of Automated Reasoning*, 6(4):465–489, 1990.

[98] Steve Winker. Absorption and idempotency criteria for a problem in near-Boolean algebras. *Journal of Algebra*, 153(2):414–423, 1992.

[99] Andreas Wolf. P-SETHEO: strategy parallelism in automated theorem proving. In Harrie de Swart, editor, *Proceedings of TABLEAUX-98*, volume 1397 of *LNCS*, pages 320–324. Springer, 1998.

[100] Andreas Wolf and Reinhold Letz. Strategy parallelism in automated theorem proving. In *Proceedings of FLAIRS-98*, 1998.

[101] Larry Wos. Searching for open questions. *Newsletter of the AAR*, 15, May 1990.

[102] Chih-Hung Wu and Shie-Jue Lee. Parallelization of a hyper-linking based theorem prover. *Journal of Automated Reasoning*, 26(1):67–106, 2001.

[103] Katherine A. Yelick and Stephen J. Garland. A parallel completion procedure for term rewriting systems. In Deepak Kapur, editor, *Proceedings of the 11th CADE*, volume 607 of *LNAI*, pages 109–123. Springer Verlag, 1992.

[104] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.