Parallelization of deduction strategies: an analytical study

Maria Paola Bonacina [*]
Department of Computer Science
University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu

Jieh Hsiang [†]
Department of Computer Science
National Taiwan University
Taipei, Taiwan
hsiang@csie.ntu.edu.tw

Abstract

In this paper we present a general analysis of the parallelization of deduction strategies. We classify strategies as *subgoal-reduction strategies*, *expansion-oriented strategies* and *contraction-based strategies*. For each class we analyze how and what types of parallelism can be utilized. Since the operational semantics of deduction-based programming languages can be construed as subgoal-reduction strategies, our analysis encompasses, at the abstract level, both strategies for deduction-based programming and those for theorem proving. We distinguish different types of parallel deduction based on the granularity of parallelism. These two criteria – the classification of strategies and of types of parallelism – provide us with a framework to treat problems and with a grid to classify approaches to parallel deduction. Within this framework, we analyze many issues, including the dynamicity and size of the data base of clauses during the derivation, the possibility of conflicts between parallel inferences, and duplication versus sharing of clauses. We also suggest the type of architectures that may be suitable for each class of strategies. We substantiate our analysis by describing existing methods, emphasizing parallel expansion-oriented strategies and parallel contraction-based strategies for theorem proving.

The most interesting and least explored by existing approaches are the contraction-based strategies. The presence of contraction rules – rules that delete clauses – and especially the application of *backward contraction*, emerges as a key issue for parallelization of these strategies. Backward contraction is the main reason for the impressive experimental success of contraction-based strategies. Our analysis shows that backward contraction makes efficient parallelization much more difficult. In our analysis, coarse-grain parallelism appears to be the best choice for parallelizing contraction-based reasoning. Accordingly, we propose a notion of *parallelism at the search level* as coarse grain parallelism for deduction.

Keywords: Parallel theorem proving, distributed deduction, contraction.

[*]Supported in part by the GE Foundation Faculty Fellowship to the University of Iowa and by the National Science Foundation with grant CCR-94-08667.

[†]Supported in part by grant NSC 83-0408-E-002-012T of the National Science Council of the Republic of China.

1 Introduction

The emergence and sustained growth of parallel architectures has inspired researchers to consider the possibility of parallel computation in virtually all applications of computing. A fairly large amount of work has been and is being devoted to the design of parallel high level programming languages, including the logic, functional, equational, object-oriented paradigms and combinations thereof. A common idea at the foundations of these paradigms is that computation is a form of deduction. Thus, in this context, parallel computation is parallel deduction. However, the study of how to parallelize other applications of deduction, such as automated theorem proving, has not been started until recently. For instance, historically speaking, resolution-based theorem proving preceded logic programming, but parallel logic programming was attempted much earlier than parallel resolution-based theorem proving.

One conceivable reason is that for many years automated theorem proving was not regarded as truly feasible, because obtaining fully automated proofs of difficult theorems seemed beyond the power of the available methods. This perception has changed significantly in recent years, largely due to the success of new techniques such as *Prolog technology based theorem proving*, *contraction-based theorem proving* and special-purpose methods (most notably [13] and [52]), which led to a number of powerful and successful provers (e.g., [1, 3, 13, 20, 30, 40, 45, 47]).

In this paper we present an analysis of the problems in parallelizing deduction strategies and a survey of existing methods. We start by giving a classification of deduction strategies, as *subgoal-reduction* strategies, *expansion-oriented* strategies and *contraction-based* strategies. Deduction strategies for high-level programming languages and some theorem proving strategies, such as Prolog technology theorem proving, belong to the first class. Theorem-proving strategies based on resolution and paramodulation belong to the second or third class depending on the role played by the contraction inference rules. Next, we define three types of parallelism: *parallelism at the term level* (fine granularity), *parallelism at the clause level* (medium granularity), and *parallelism at the search level* (coarse granularity). We relate the granularity of parallelism to the choice of memory configuration (i.e., shared versus distributed memory) and to the problem of *conflicts* between parallel inference steps accessing common premises.

These two criteria, class of strategy and type of parallelism, provide us with a framework to treat the problems in parallel deduction and with criteria to classify proposed approaches. We consider according to the above classifications parallel term rewriting, (e.g., [24, 25, 31, 32]), parallel Prolog technology theorem proving, (e.g., METEOR [3], PARTHENON [12, 21], PARTHEO [42]), parallel expansion-oriented methods, (e.g., DARES [22], PARROT [29]), parallel connection graph procedures, (e.g., [37, 19]), parallel implementations of the Buchberger algorithm, (e.g., [17, 26, 44, 49]), parallel implementations of Knuth-Bendix completion, (e.g., [54]) and parallel theorem provers with contraction, (e.g., ROO [39], the Team-Work method [4, 23] and Clause-Diffusion [10, 9, 11]).

Our analysis shows that as we move from subgoal-reduction strategies to expansion-oriented strategies and finally to contraction-based strategies, the database becomes more and more *dynamic* and the amount of *inter-dependence* between inferences grows. Therefore, while subgoal-reduction strategies are amenable to all three types of parallelism and expansion-oriented strategies may still exploit both clause level and search level parallelism, only the latter is cost-effective for contraction-based strategies.

In the last part, we focus on contraction-based strategies, and we describe a notion of *parallelism at the search level.* We analyze the advantages and disadvantages of *distributed memory* versus those of *shared memory* to implement this type of parallelism. The outcome is in favor of either a purely distributed environment or a hybrid one, that is, a distributed environment with a shared memory component.

In the last technical section of the paper we briefly describe our view of what we think as the appropriate architectures for each category of the strategies.

This paper contains the following contributions:

- Our classification of strategies according to the behaviour of the presentation during the derivation, and the proceeding analysis of strategies and granularities of parallelism seem to be new.
- We differentiate between *forward* and *backward* contraction, and we point out the key role of this distinction in understanding the problems in the parallelization of contraction.
- We identify the source of difficulty in parallelizing contraction-based strategies, namely backward contraction. Ironically, backward contraction is also the raison d'être of contraction-based strategies.
- We propose a notion of *parallelism at the search level*, which entails partitioning the search space among cooperating agents rather than merely duplicating the problem under different strategies.
- We survey several methods: the approaches that we selected here, sometimes considerably far apart, did not appear together elsewhere. Our intention is not to provide a complete survey, but to offer specimens for our concepts and illustrate the material we used for our analysis.

Acknowledgements

The bulk of this work was done while both authors were with the Department of Computer Science of the State University of New York at Stony Brook. In addition to the support from the Department and the University, we acknowledge support from the National Science Foundation with grant CCR-8901322 given to SUNY Stony Brook. The first author was also supported by a fellowship of Università degli Studi di Milano, Italy.

2 The classification of strategies

A *theorem proving problem* consists in deciding, given a set of clauses S and a clause φ, whether φ is a theorem of S. We call S the *presentation* and φ the *target* or *goal.* A theorem proving strategy $\mathcal{C}$ is specified by a set of *inference rules* and a *search plan.* *Expansion* inference rules, such as resolution and paramodulation, derive new clauses from the existing ones and add them to the data base. *Contraction* inference rules, for instance, subsumption and simplification by means of equations, delete clauses or replace them by logically equivalent clauses that are smaller according to some pre-defined well-founded ordering. The search plan chooses the inference rule and the premises for each step, thereby constructing a derivation

$(S_0; \varphi_0) \vdash_{\mathcal{C}} (S_1; \varphi_1) \vdash_{\mathcal{C}} \ldots (S_i; \varphi_i) \vdash_{\mathcal{C}} \ldots$.

Depending on the strategy, one may refine the notion of derivation by adding more or deleting components in the tuple. For instance, in refutational, resolution-based strategies, the negation of the input target may be added to the presentation and the target component may not be singled out at the successive stages of the derivation.

Each step in the derivation may be an expansion or a contraction step and modify the presentation and/or the target. At the operational level, we distinguish further between *forward* contraction steps and *backward* contraction steps. A contraction step is a *forward contraction step* if it reduces a newly generated clause. If it reduces an existing clause, then it is a *backward contraction step.*

Deduction strategies can be classified into three categories, according to the behaviour of the presentation during a derivation.

- **Subgoal-reduction strategies**:

  The main characteristic of this class of strategies is that the presentation remains *static* during the derivation:
  $$(S; \varphi_0) \underset{\mathcal{C}}{\vdash} (S; \varphi_1) \underset{\mathcal{C}}{\vdash} \ldots (S; \varphi_i) \underset{\mathcal{C}}{\vdash} \ldots.$$
  Strategies for functional and logic programming and some theorem-proving strategies, such as Prolog technology-based methods [45], belong to this class. In a Prolog-style logic program, for instance, the presentation, represented by the program, is not modified during the computation. The elements in the presentation are applied to reduce the target to subgoals, until a solution is reached. A Prolog derivation can be further refined into
  $$(S; \varphi_0; A_0) \underset{\mathcal{C}}{\vdash} (S; \varphi_1; A_1) \underset{\mathcal{C}}{\vdash} \ldots (S; \varphi_i; A_i) \underset{\mathcal{C}}{\vdash} \ldots,$$
  where the target φ_i is the current goal and A_i is the set of its ancestors, which are saved for the purpose of backtracking. To summarize, in a subgoal-reduction strategy, the database is *static.*

- **Expansion-oriented strategies**:

  Earlier studies of general-purpose theorem proving mostly emphasized expansion inference rules like resolution, which add new deductive consequences to the presentation whose members are usually not removed. We categorize them as *expansion-oriented strategies*, which include

  – theorem-proving strategies that feature only expansion rules and

– theorem-proving strategies that feature also contraction rules, but apply them only for *forward contraction*, that is the contraction of newly generated clauses before their insertion in the data base.

A derivation has the form

$$(S_0; \varphi_0; N_0) \underset{\mathcal{C}}{\vdash} (S_1; \varphi_1; N_1) \underset{\mathcal{C}}{\vdash} \ldots (S_i; \varphi_i; N_i) \underset{\mathcal{C}}{\vdash} \ldots,$$

where $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$. We use the component N_i to represent the set of newly generated clauses, which we call *raw clauses*, because they have not been contracted yet. Whenever a raw clause ψ is generated from elements in S_i, it is added to N_i. In a strategy with forward contraction (such as forward subsumption), ψ may first be reduced by using S_i. If ψ is reduced to a nontrivial clause ψ', ψ' is then added to S_{i+1}. Since no contraction is applied to any clause in S_i, the database is *monotonically increasing.*

- **Contraction-based strategies**:

  This class includes deduction strategies with both expansion and contraction rules. The latter are often applied eagerly, for both forward and backward contraction. We call this the *eager contraction* scheme. A derivation has the form

$$(S_0; \varphi_0) \underset{\mathcal{C}}{\vdash} (S_1; \varphi_1) \underset{\mathcal{C}}{\vdash} \ldots (S_i; \varphi_i) \underset{\mathcal{C}}{\vdash} \ldots.$$

  In an eager contraction scenario, whenever a raw clause ψ is generated from two elements in S_i, it is added to S_i to form S_{i+1}. Then ψ is reduced to its normal form ψ' with respect to S_i (i.e., ψ is *forward contracted*). Then if ψ' has not been deleted, it is used to try to contract all elements in S_i. If a clause in the database is reduced, it is also used to further contract other clauses. This entire process is what we call backward contraction: a clause already existing in the database and in use for inferences may be reduced or deleted by a clause generated afterwards. Since contraction is applied to clauses in S_i, the database is *dynamic.* The important monotonicity property here is $R(S_0) \subseteq R(S_1) \subseteq \ldots \subseteq R(S_i) \subseteq \ldots$, where $R(S)$ denotes the set of redundant[1] clauses in S [7]. In other words, a clause that becomes redundant during a derivation remains redundant for the rest of the derivation.

This classification is not meant to capture all properties of the strategies. It emphasizes those aspects that will be relevant to parallelization, such as the static or dynamic nature of the database.

3 Granularity of parallelism in inference systems

We now classify the types of parallelism in terms of *granularity.* We regard *parallelism at the term level* as fine-grain parallelism, *parallelism at the clause level* as medium-grain parallelism, and *parallelism at the search level* as coarse-grain parallelism. For each of these types of parallelism

[1]A clause is *redundant* in S if it can be deleted without modifying the essential structure of proofs in S: not only the set of theorems of S is unaffected, but also the minimal (i.e., intuitively essential) proofs in S are unaffected if redundant clauses are deleted.

we identify the granularity of data and the granularity of operations. The granularity of data tells which type of datum represents a *grain* of memory with its own access rights. The granularity of operations tells which type of operation is executed by each concurrent process:

	granularity of data	granularity of operations
parallelism at the term level	term	subtask of inference step
parallelism at the clause level	clause	inference step
parallelism at the search level	set of clauses	many inference steps

In **parallelism at the term level** every term is a *grain* of memory. Two processes may access concurrently two subterms of a same term, provided certain conditions, such as disjointness, are met. Thus, the data of the problem may be partitioned among concurrent processes by giving to concurrent processes disjoint subterms, and the concurrent processes may cooperate to achieve a single inference step. Parallel matching, parallel unification and parallel rewriting (e.g. [24, 25, 31, 32]) are examples of parallelism at the term level. At the **clause level**, every clause is a grain of memory: the data are partitioned by giving to concurrent processes distinct clauses and typically each process performs one inference step or relatively few inference steps with a common premise. Examples of methods that work at the clause level are [29, 39, 44, 49, 54]. In **parallelism at the search level**, concurrent, asynchronous processes search in parallel for a solution. As soon as one of them succeeds, the whole process succeeds. The search space is partitioned among the processes by distributing the clauses. Unlike fine- and medium-grain parallelism, each process is given a large portion of the data, (e.g., a fairly large set of clauses), and the data sets of the processes may be physically separated. Each process develops its own derivation, while communicating with the other processes. Different types of parallelism at the search level are presented in [4, 22, 26].

3.1 Shared memory versus distributed memory

The granularity of parallelism is related to the choice between *shared memory* and *distributed memory*. The *local memory* of a process p_i is the memory which is accessible only to p_i itself. We use *distributed memory* for a configuration where the entire memory of the system is represented by the union of the local memories. If a process p_j needs to access data belonging to another process p_i, process p_j has to send a *message* to p_i. All nonlocal references are handled through messages. In *shared memory*, on the other hand, an address in memory can be accessed directly by all the processes.

Common data are shared in shared memory, while they may be duplicated in distributed memory. Sharing requires some sort of protection to prevent concurrent-write, for example, the *critical regions* applied in [49] or the *locks* used in [39, 54]. Protection may slow down write-access to the shared data, imposing synchronization delays. A distributed approach does not require synchronization, and therefore it may allow in principle a higher degree of parallelism. The trade-off is represented by a larger amount of memory and the communication delay of message passing. Parallelism at the term level and parallelism at the clause level lead in general to adopting a shared memory: for instance, it would not be practical to scatter the terms of a clause over a distributed memory. Parallelism at the search level can be implemented in principle in either shared or distributed memory, provided that the data sets of the processes are physically distinct.

3.2 Conflicts

The granularity of parallelism is also related to the problem of conflicts between concurrent processes. We say that two concurrent processes are in **conflict** if they need to access the same grain of data. First we describe conflicts at the clause level, and then we discuss the relation between granularity and conflicts. Two inference steps whose sets of premises are disjoint are trivially not in conflict. Since expansion steps do not modify their premises, expansion steps simply need to be granted *read-access* to their premises. We assume that concurrent read can be safely admitted and therefore expansion steps do not cause conflicts. In the presence of contraction steps, three types of conflicts may emerge:

1. *write-write conflict between contraction steps*: two concurrent contraction steps try to re-write the same clause φ,

2. *read-write conflict between contraction steps*: a contraction step attempts to re-write a clause φ, which is being read by another contraction step (which would use φ to contract another clause ψ) and

3. *read-write conflict between contraction steps and expansion steps*: a contraction step tries to re-write a clause φ, which is being read by an expansion step (which would use φ as parent).

Under the assumption that raw clauses are not involved in any inference before undergoing forward contraction, both types of read-write conflicts are caused by backward contraction. In [10], we showed that read-write conflicts between contraction steps do not represent real conflicts: if the step deleting the common premise φ prevents the application of φ to contract ψ, the latter can still be reduced. Therefore in the following we shall mention only the two remaining types, 1 and 3, of conflicts.

A write-write conflict between contraction steps is a real conflict, since one step prevents the other. However, even if only one of the two steps commits, the common premise φ is reduced to a smaller form.

Read-write conflicts between contraction steps and expansion steps are the most critical ones. If the expansion step proceeds, it may generate a clause that is very likely to be redundant, because one of its parents φ is not fully reduced. The other choice is to suspend the expansion step and let the contraction step commit and reduce φ. More generally, in order to prevent the generation of redundant clauses, one would like to suspend the expansion step until it is guaranteed that its premises are fully reduced by contraction. However, in a parallel setting it may not be possible to satisfy this requirement without introducing possibly long synchronization delays, since the clauses which may reduce φ may belong to other processes.

Employing different granularities implies choosing different approaches to resolve the problem of conflicts. Given the conflicts at the clause level, one approach is to eliminate some of them by adopting a finer granularity, that is, moving from the clause level to the term level. If access is at the term level, steps accessing disjoint positions within a clause are not in conflict. The cost of this choice is the overhead of handling access at the term level. Another approach moves in the opposite direction, that is, from the clause level to the search level. In parallelism at the search level in distributed memory, *there are no conflicts*, because the concurrent processes access separate sets of data. Two concurrent processes may rewrite concurrently the same clause, because they are rewriting two distinct copies of the same clause. In other words, conflicts are eliminated by duplicating premises, at the cost of introducing additional redundancy with respect to a sequential execution.

4 Subgoal-reduction strategies

In this section we consider two classes of subgoal-reduction strategies: strategies for Prolog technology-based theorem proving and strategies for rewrite-based programming languages.

4.1 Prolog technology parallel theorem provers

Prolog technology theorem proving (PTTP) appeared first in [45]. A Prolog technology prover is basically a resolution-based theorem prover realized on top of a WAM (Warren Abstract Machine) [50], the virtual machine designed for the fast implementation of Prolog [47]. The inference system of a PTTP is an extension of Prolog's inference mechanism to first order logic, based on the *model elimination* principle [38]. The search plan is *depth-first iterative deepening* [34, 46], a modification of the depth-first search procedure of Prolog. Depth-first iterative deepening is depth-first search with a limit k on the depth of the descent. If, after k steps of depth-first search along a path, no solution has been found, the procedure backtracks and selects another path. If all paths have been pursued down to depth k and yet no solution has been found, the limit k is raised to $k+n$, for some $n \geq 0$, and the search restarts.

Well-known parallel Prolog technology theorem provers include PARTHENON [12, 21], PARTHEO [42], and METEOR [3]. In this Prolog-based setting, parallelism is *or-parallelism*: each concurrent process tries to apply a clause, or "rule" in Prolog, to one of the current goals. All processes have access to the input set of clauses. At first, each process selects an input clause, possibly a different one for each process, and tries to resolve it with the input goal (i.e., the negation of the target theorem) generating new subgoals. A goal is regarded as a *task*, namely the task of trying to derive the empty clause from that goal. The distribution of tasks among the processes is done through *task stealing*: if a process runs out of active tasks, either because no rule applies to its tasks or because of the limit k in the descent, it gets more tasks from the queues of other processes. When parallelizing iterative deepening, one has to choose whether the same limit k is set for all processes or each process has its own limit. In the first case, the derivation reaches the limit when the processes fall in a deadlock, all of them asking each other for tasks. The deadlock may be solved by resetting the limit to a higher value. In the second case, the scheduling procedure becomes more complicated as the queues of the processes may contain tasks pertaining to different values of k. In order to preserve completeness, tasks originated under lower values of the limit need be given higher priority.

These are just the basic elements of a parallel Prolog technology theorem prover. PARTHENON implements them for shared memory machines, whereas PARTHEO has been implemented on a network of transputers and on an Intel hypercube. The METEOR system is available for sequential machines, parallel machines with shared memory, and networks of workstations. We refer to [3, 12, 21, 42] for specific descriptions of how the principles outlined above are realized in these systems. One interesting aspect of PARTHEO is the mechanism for task stealing. In PARTHEO, task stealing is realized by message-passing. A message representing a subgoal φ is an encoding of the WAM operations that are necessary to derive φ from the input clauses. Upon reception of such a message, a processor derives φ by executing the steps encoded in the message. Thus, it happens routinely that the processors repeat steps that other processors have already performed. The rationale for this choice lies in the general philosophy of Prolog technology theorem proving. PTTP makes a limited use of contraction inference rules. For instance, in PARTHEO, tests for subsumption, tautology elimination and purity reduction [18] are applied to the input set only in a pre-processing phase. Prolog technology theorem proving is a representative of the choice of not investing significant resources in contraction and concentrating all efforts in speeding the expansion process. One hopes to make expansion steps so fast that it becomes irrelevant, performance-wise, whether they are redundant or not. Accordingly, the authors of PARTHEO assume that the WAM operations are so fast, that the redundancy represented by repeating them upon reception of messages does not harm the performances.

4.2 Parallel term rewriting for equational languages

Simplification by equations, also known operationally as term rewriting, is a fundamental inference rule in contraction-based strategies. Thus, an analysis of the parallelization of deduction strategies cannot ignore parallel term rewriting. However, a large part of the existing research on parallel simplification has been done with the purpose of designing parallel interpreters for equational languages and parallel architectures to support them, (e.g., [24, 25, 31, 32]). A survey of such works is beyond the scope of this paper. Also, since we focus on the application of automated deduction to theorem proving, rather than to interpreting programs, we do not describe any such method in detail. We limit ourselves to discuss why these methods are subgoal-reduction strategies and which special hypotheses they assume. The effect of such hypotheses on parallelization will be analyzed in Section 4.3.

An equational program is a set of equations E and the input to the program is a term t. E is the *presentation* and t is the *goal*. In *functional programming*, an execution consists in reducing t by the equations in the program E until it is E-irreducible, that is, no equation in E applies. The irreducible term is the output. Operationally, each inference employs *matching*, and it may apply to any position of the goal t. The equations in the program are required to satisfy certain conditions (e.g., *non-overlapping* and *left linear* [27]) which imply the uniqueness of the irreducible form. Therefore backtracking is not necessary, because the same irreducible form, if one exists, will be found regardless of the path chosen in the reduction.

In *logic programming*, the goal t contains variables and an execution consists in computing an answer substitution for those variables. At each step an equation in the program is overlapped (a step similar to resolution in Prolog) with the goal to generate a new subgoal. Operationally, each inference employs *unification*, to instantiate the variables in the goal and generate the answer substitution, and it applies at the top of an atom in the goal. Backtracking is used to search for an answer.

Beyond the operational differences, equational strategies for functional or logic programming are subgoal-reduction strategies, because

- there is a goal,
- the presentation (i.e., the program) is *static* during the derivation, and
- each step in the derivation applies to the goal and consists in replacing the current goal by subgoal(s).

4.3 Analysis of the parallelization of subgoal-reduction strategies

The following properties of subgoal-reduction strategies can be observed in both equational programs and Prolog technology theorem proving (PTTP):

- **Static database**:

  The input presentation, for example, the equations of an equational program or the "rules" in the sense of Prolog, remains unchanged throughout the execution. In PTTP, the subgoals are not subject to contraction after having been generated. This has three consequences:

  - **Preprocessing**:

    It is possible to *preprocess* the input clauses for fast execution. For instance, the equations of an equational program can be preprocessed for the purpose of parallel matching (e.g., [32]) or compiled in a lower level language with explicit parallelism [24, 25].
  - **Read-only data**:

    The input clauses are read-only data, which may be stored conveniently in a shared memory.
  - **Specialized data structures**:

Since there is a definite distinction between the goal and the "rules", these two different types of data may be represented by different, specialized data structures. For instance, in PTTP, it is possible to represent a subgoal by an encoding of the WAM operations that derive it (e.g., [42]). In equational programming, specialized data structures can be designed to prevent conflicts among concurrent rewriting steps [32].

- **No conflicts**:

  In equational programming, the only conflicts that may arise are write-write conflicts between contraction steps. Most definitions of equational programs require that the equations are *left-linear* and *non-overlapping* (e.g., [27]). If two nonoverlapping equations $l_1 \simeq r_1$ and $l_2 \simeq r_2$ can simplify a given term t, the nonoverlapping property implies that l_1 and l_2 match two subterms of t at disjoint positions. Intuitively, two simplification steps that reduce disjoint subterms need write-access to different portions of the term and therefore can be applied in parallel. If all the equations in a program are nonoverlapping, this is true for *any* two steps in any execution of such program. In other words, there are *no conflicts*.

  Furthermore, by exploiting preprocessing of equations and specialized data structures, it is possible to weaken significantly the requirements that the equations be left-linear and nonoverlapping, while maintaining the *no conflicts* property [32].

  In PTTP the basic inference mechanism features only expansion steps, that is, the resolution-type expansion inferences that derive from a goal and a "rule" a new set of subgoals. Therefore, there are *no conflicts* between inference steps. For instance, two processes that read the same "rule" and apply it to different subgoals can be executed in parallel.

Because of these properties (i.e., *static data base* and *no conflicts*), subgoal-reduction strategies are amenable to all three types of parallelism: parallelism at the term level (e.g., parallel rewriting), parallelism at the clause level, (e.g., or-parallelism [12, 21, 42]), and parallelism at the search level, (e.g., the distribution of subgoals to be solved independently by concurrent processes).

5 Expansion-oriented strategies

Examples of approaches to parallel expansion-oriented theorem proving are the PARROT prover [29], the parallel connection graph procedures (e.g., [19, 37]), the DARES system [22], and the parallel implementations of the Buchberger algorithm in [16, 17, 26, 44, 49]. We shall explain later why we treat the Buchberger algorithm as expansion-oriented for the purpose of parallelization.

The PARROT prover utilizes parallelism at the clause level in shared memory. We do not describe this system in detail, because most of the considerations it may raise will be covered by the study of parallelism at the clause level in shared memory for contraction-based strategies, and in particular by the analysis of ROO [39] (see Section 6.2).

In the following subsections we consider the parallel connection graph procedures, the DARES project, and the parallel implementations of the Buchberger algorithm.

5.1 Parallel connection graph procedures

In connection graph procedures [35], a set of clauses is represented as a graph, where a node corresponds to a clause and an edge links any two literals which are unifiable and have opposite sign. The basic inference rule is *link resolution*: two literals connected by an edge are resolved upon and the resolvent clause is added as a new node of the graph. The resolvent inherits the edges of its parent nodes, while the edge resolved upon is deleted. We refer to [35] for a more complete account of the method and its refinements.

A natural way to parallelize connection graph strategies is to resolve several links in parallel. A method based on this principle is given in [37] for a shared memory machine. The main problem with this type of approach is that unrestricted parallel link resolution is inconsistent. Informally, one reason of inconsistency is that parallel link resolution steps may be in *conflict*. This is possible because in the connection graph representation resolution steps are not purely expansion steps. Indeed, each link resolution step *deletes* the edge resolved upon, thereby modifying the representation of its premises. The inconsistency of unrestricted parallel link resolution is well-known. It was observed first in [36] and we refer to [36, 37] for a more complete account of these issues. The answer to the inconsistency problem has been to impose strong restrictions on which links may fire concurrently, such as in [37]. Such restrictions, however, limit severely the amount of parallelism that can be exploited. Because of the nature of the connection graph representation, parallel link resolution yields parallelism at the literal level that is a sort of fine-grain parallelism. The high incidence of conflicts is evidence that parallelism at the literal level is too fine for expansion-oriented strategies.

Other parallel connection graph methods (e.g., [19]) aim at utilizing a coarser type of parallelism, by partitioning the connection graph into subgraphs and assigning each subgraph to a deductive process. This approach has also been designed for shared memory. The conflicts and the consequent restrictions do not disappear completely: they are limited to the borders of the subgraphs. Locks are employed to prevent the conflicts. In spite of the slightly coarser granularity, it seems that the incidence of conflicts in shared memory and the consequent restrictions still prevented these methods from being effective.

5.2 The DARES system

DARES (Distributed Automated REasoning System) [22] is an application of technologies for distributed problem solving to theorem proving. It assumes a resolution-based strategy with a *level-saturation* search plan. The inference mechanism features resolution and forward subsumption, but no backward contraction. A number of processes, called *agents*, work concurrently. In turn, each agent comprises many sub-processes: several deductive processes and one *mail process*, which is dedicated to inter-agent communication. An agent is a sequential process: the processes inside an agent are scheduled for execution according to a round-robin scheduling. All deductive processes of all agents execute the given strategy. In general, different deductive processes work on different problems. It is not clear in [22] whether this means that more than one problem may be given in the input or whether the input problem is reduced to sub-problems during the computation. In the second hypothesis, it is not specified how the given problem is reduced to sub-problems to be given to different processes. The only constraint is that no two processes of the same agent work on the same problem.

The authors observe that theorem proving is a computationally intensive application and therefore the agents should be *loosely coupled*, in the sense of devoting more time to computation than to communication. Indeed, each agent has several deductive processes and just one mail process. Each agent has "incomplete knowledge". This means that it has access only to a subset of the clauses involved. It follows that the agents need to "import knowledge" from other agents, in order to be able to solve the problems. This communication is organized as follows. Whenever an agent A cannot derive new resolvents from its clauses, it sends to other agents a request for more data. Upon receiving such a request, any other agent B will reply by sending to A a subset of its clauses. Agent A will include the received clauses in its data base and proceed with the inferences.

This scheme of communication uses three heuristic criteria to decide, respectively, when to issue a request, what to put in the request, and what to put in the reply. In addition to sending a request when it cannot generate new resolvents, an agent applies a heuristic criterion to establish whether it is making progress toward the solution or not. If the decision is negative, it emits a request for more data. One such criterion is described in [22]. It is based on measuring the length of the clauses and the number of distinct predicates in successive sets of new resolvents. Inference steps which reduce these measures, together with steps that generate or use unit clauses, are regarded as "proof-advancing". If the derivation is not proof-advancing for two consecutive stages, the agent sends a request-message to import more clauses. The decision is based only on the two most recent stages of the derivation. A request-message consists of a set of literals. Another heuristic is employed to select the literals to be put in the message. Subsumption is applied to make sure that a request-message does not contain instances of other literals in the message. The receiver of a request-message tests the literals in its clauses for unifiability with the literals in the message. If at least a literal in a clause unifies with at least a literal in the message, the clause is selected. Then, all selected clauses are sent together as a reply-message to the agent which issued the request-message. Subsumption is used also to delete instances in a reply-message. This application of subsumption is actually the only backward contraction in DARES. Clearly, this does not change the expansion-oriented nature of the method, because it is applied only within the messages, not directly to the data bases of the processes.

DARES has been implemented on a simulator of distributed systems, which runs on a network of Lisp machines. Because of the presence of the simulator, the absolute run times of such a system are probably not very significant, and indeed they are not given in [22]. Experimental measures of how the run time varies with the number of agents and the amount of redundancy in the system are reported. However, this study refers the results for just one theorem proving problem. In this context, redundancy simply means duplication: the *redundancy factor* for a clause is 0 if there is just one copy of that clause in the entire system; it is 1 if all agents have a copy of that clause. This analysis is more concerned with redundancy as a property of the given distribution of clauses than with redundancy generated by the system itself. Indeed, redundancy is treated as an independent variable. Not surprisingly, as redundancy grows, each agent's behaviour becomes closer to that of a sequential theorem prover, and the performance of DARES gets worse.

The effectiveness and efficiency of a system like DARES rest on the heuristics used in the communication part. We list here a number of issues that may be of interest for further study. There is no treatment of the *fairness* of such heuristics, that is, whether they ensure that whenever there is a proof, it will be found. The selection of clauses for the reply-messages in [22] seems rather expensive, since it consists in actually trying resolution between the clauses in the data base and the literals in the request-message. These resolution steps will be performed again when the reply-message reaches its destination. Finally, the heuristics described may not be sufficiently effective in partitioning the search space. For instance, if the literals in the request-message are sufficiently general, such as literals whose arguments are all variables, almost all the clauses may be selected, so that the data bases at the sender and at the receiver of the request-message will be almost identical. Furthermore, no distinction is made between generated clauses and received clauses: if a clause φ is generated by agent A and then sent to agent B, both A and B will perform all the inference steps they can with φ. As more and more clauses are exchanged, this will contribute to increase the overlap among the data bases and thus the portions of search space allotted to different agents.

5.3 Parallel implementations of the Buchberger algorithm

Problems related to those of parallel theorem proving have been addressed by the study of parallel and distributed implementations of the *Buchberger algorithm* [16, 17, 26, 44, 49]. The Buchberger algorithm works on polynomials, equated to 0 and treated as oriented equations. It takes as input a set of polynomials and gives as output a set of polynomials, which is a *Gröbner basis* for the ideal generated by the input polynomials. A Gröbner basis has the property that it reduces to 0 all and only the polynomials belonging to the ideal [14].

The Buchberger algorithm features an expansion inference rule similar to superposition [33] and a contraction rule similar to simplification. Indeed, the similarity between the Buchberger algorithm and Knuth-Bendix completion was explored in [15]. We present these parallel implementations of the Buchberger algorithm in the section on parallel expansion-oriented strategies for two reasons.

First, most of the parallel Buchberger algorithms we survey choose not to implement backward contraction in full. In other words, they do not maintain the current basis fully reduced during the derivation.

Second, backward contraction is not as crucial for the Buchberger algorithm as it is for theorem proving. The former is guaranteed to succeed, namely, to generate a Gröbner basis, regardless of the amount of contraction performed. If the output basis is not fully reduced, it is possible to follow the execution of the parallel algorithm by a final phase, where a sequential normalization algorithm is applied to all the elements of the basis. This is done for instance in [44, 49]. The logical domain of theorem proving, on the other hand, is usually undecidable. The absence of backward contraction in a theorem proving strategy may cause the prover to saturate the available memory with redundant clauses and run out of memory before finding a proof.

Another difference between the Buchberger algorithm and theorem proving strategies is that the expansion inferences in the Buchberger algorithm do not use unification, since there are no variables, as the "variables" in the polynomials are constants logically. It follows that expansion steps are much less expensive than in theorem proving. Therefore, expansion-oriented implementations of the Buchberger algorithm are less expensive and thus more feasible than expansion-oriented approaches in theorem proving.

The algorithms presented in [16, 17, 26, 44, 49] are conceived for three different models of parallel computation: a *shared-memory multiprocessor* in [49], a *data-flow machine* in [44] and a *distributed-memory multiprocessor* in [16, 17, 26].

A parallel Buchberger algorithm in shared memory

In [49], the data base of equations is stored in shared memory with **Concurrent-Read-Exclusive-Write (CREW)** access. The algorithm is a parallel version of the Buchberger algorithm with *critical regions* to enforce exclusive-write. The algorithm is simple, and a formal proof of correctness is given in [49]. However, it seems to contain an unnecessarily high degree of sequentiality. The reason is that the granularity of memory protection is very large: entire sets of equations, e.g. the current version of the basis and the set of raw critical pairs, are accessed in exclusive-write mode. Only one expansion process at any given time can access the set of raw critical pairs to add a new pair. It follows that the expansion part of the algorithm is basically sequential. No *backward contraction* is included.

A data-flow parallel Buchberger algorithm

The method in [44] is inspired by an analogy, due to Buchberger himself, between the Buchberger algorithm and the **Eratosthenes's sieve** algorithm for generating all the prime numbers smaller than or equal to a given n. A data-flow version of Eratosthene's sieve works as a *pipe* with stages $s_1, \dots s_i$ to store the prime numbers $m_1, \dots m_i$ discovered so far. The numbers from 1 to n are input in increasing order at one extreme of the pipe and travel through its stages. At each stage s_j, any candidate for primality k is tested for divisibility by m_j. If m_j divides k, k is eliminated. If no m_j's divides k, k is saved as m_{i+1} at a new stage s_{i+1}. When the input stream is exhausted, all the numbers remained in the pipe are the prime numbers.

According to the analogy between the Buchberger algorithm and Eratosthene's sieve, simplification corresponds to the divisibility test and the Gröbner basis of equations corresponds to the set of prime numbers. Any raw critical pair $l \simeq r$ travels through the pipe, and the equations already stored in the stages of the pipe are applied to reduce $l \simeq r$. If $l \simeq r$ is not deleted in the process, that is, it is reduced to a nontrivial equation $l' \simeq r'$, the latter is added as a new stage at the end of the pipe. Then, a copy of $l' \simeq r'$ travels backward, and at each stage it tries to generate raw critical pairs with the equation stored at that stage. Raw critical pairs are collected at the input extreme of the pipe and fed back in the pipe for simplification. To summarize, forward contraction is performed as new equations traverse the pipe in forward direction. Expansion is performed as the equations traverse the pipe in backward direction. When the algorithm halts, the contents of the pipe is a Gröbner basis.

Backward contraction does not fit very naturally in this approach. The analogy with Eratosthene's sieve does not help for backward contraction. If a number k is not divided by any prime number smaller than k, then k is prime. It follows that no stage of the pipe ever needs to be deleted. On the other hand, for an equation $l \simeq r$ to be part of a fully reduced version of the basis, it is not sufficient that $l \simeq r$ is fully reduced with respect to the equations generated before $l \simeq r$. It is necessary to keep trying to reduce $l \simeq r$ by equations generated afterwards. According to [44], some backward contraction is performed when the equations travel in backward direction. However, it is not guaranteed that the output basis is fully reduced. Indeed, the method features a final phase, where a sequential normalization algorithm is applied to all the elements of the basis output by the pipe.

A distributed Buchberger algorithm for constraints solving

The algorithm in [26] is designed to be applied to solve polynomial constraints during the interpretation of a program in a logic programming language with constraints. Thus the input to the Buchberger algorithm is not a given set of polynomials, but a stream of polynomials progressively generated by an interpreter. Each input equation is sent through a shared bus to all the processors. A processor p_i maintains two sets of equations: a set CP_i of raw critical pairs, just received from the input or just generated at the node, and a current version of the basis B_i. The distribution of the work among the processors is regulated by a *work-load function* w, which assigns equations to processors. Each processor p_i is responsible for normalizing with respect to B_i those equations in CP_i that w assigns to p_i. Whenever the smallest equation $l \simeq r$ in CP_i is B_i-irreducible and belongs to p_i, node p_i broadcasts $l \simeq r$ to all the other nodes, generates all the critical pairs between $l \simeq r$ and elements in B_i, stores them in CP_i and moves $l \simeq r$ from CP_i to B_i. Any other processor p_j, upon receiving $l \simeq r$, performs the same steps, that is, it generates all the critical pairs from $l \simeq r$ and B_j, stores them in CP_j and moves $l \simeq r$ from CP_j to B_j.

All nodes do all the expansion steps independently. The choice of having each node doing all expansion steps is clearly related to the low cost of the expansion steps in the Buchberger algorithm. Forward contraction is distributed among the processors according to the work-load function. The latter is originally a partition of the input and is updated dynamically during the computation. Once again, backward contraction receives the least attention. As a background task, not included in the main algorithm, each processor p_i is supposed to normalize with respect to B_i the equations of B_i which belongs to p_i. However, if an equation in B_i is simplified, the reduced form is not moved to CP_i and the B_j's at the other nodes are not modified accordingly.

A transition-based distributed Buchberger algorithm

The distributed Buchberger algorithm of [16, 17] is obtained by applying to Buchberger algorithm a general approach for parallel programming, proposed in [53]. Given a sequential algorithm, a parallel algorithm is obtained in two phases: first, derive from the sequential algorithm a set of *non-deterministic transitions*, second, design a scheduler to execute the transitions. Each transition defines a *task* and the parallel algorithm is basically composed of the tasks and the scheduler. Given a pool of processes, each one of them will execute concurrently the scheduler and thus perform instances of tasks (i.e., apply transitions rules to data). In shared memory, each process schedules a transition and then gets the data to which apply the transition from shared data structures, with locks to ensure exclusive access. In distributed memory, each process gets the data from its own local memory, and some sort of communication is in place to ensure that the processes cooperate. We refer to [53] for a complete treatment. This technique applies rather naturally to algorithms for symbolic computation, where a specification of the algorithm as a set of nondeterministic transition rules often is given in the first place, for example, the inference rules of an inference system.

In [17], two distributed versions of Buchberger algorithm are described and proved correct, one with backward contraction and one without. For both algorithms, G is the current basis and *gpq* (*global pair queue*) is a queue of pairs of polynomials to be used for generating other polynomials. The basis is *replicated*: each process p_i has its own version G_i of the current basis in its local memory. The global pair queue is *distributed*: each process p_i has a subset gpq_i of gpq. The basic transitions that p_i may perform are as follows:

1. Select a pair from gpq_i and generate a new polynomial q (expansion); q belongs to p_i.
2. Reduce q to its irreducible form q' with respect to G_i (forward contraction).
3. Add q' to G_i and the pairs (q', r) for all r in G_i to gpq.

The work is distributed among the processes by partitioning the global pair queue gpq of the abstract algorithm into the gpq_i's and by establishing that a polynomial may be reduced only by the process that owns it. If the pair queue at one process (e.g., gpq_i) is depleted, while the pair queue at another process (e.g., gpq_j) keeps growing, "load-balancing messages" are used to move pairs from gpq_j to gpq_i.

Each process p_i also has a *shadow set* G'_i of G_i. The purpose of the shadow sets is communication: whenever a process p_i adds a polynomial to its G_i, it also adds it to all the G'_j's, i.e. the shadow sets of the other processes. Adding an element to G'_j *"invalidates"* G_j. In order to *"validate"* G_j, process p_j needs to move to G_j all the polynomials that are in G'_j but not in G_j. The key property is that each process keeps working, even if its G_i has been invalidated. Addition of a polynomial to all the remote shadow sets is done by broadcasting a message. The sender waits to receive acknowledgements from all the other processes. In order to save memory and keep the size of the messages small, shadow sets and messages contain *global identifiers* of polynomials, not the polynomials themselves. When p_i validates G_i, it sends requests for all the polynomials whose global identifiers are in G'_i but not in G_i, and waits until it has received all the polynomials it has requested.

If backward contraction is added, the polynomials are subject to contraction after they have been inserted in the basis. Any polynomial in the basis may be applied to reduce any other polynomial in the basis, with the purpose of keeping the basis *inter-reduced*. In the distributed algorithm in [17], backward contraction is performed in the shadow sets G'_i's. For this purpose, the shadow sets contain the polynomials, not just their identifiers. More precisely, the shadow sets contain *version sequences*, namely, lists of all the forms of a polynomial. The topmost element of a sequence is the current form of a polynomial. When it is reduced (backward contracted), the new form is appended on top of the list. No polynomial is really deleted. A local basis G_i is then a selection of versions, not necessarily the most updated ones, from a subset of version sequences in G'_i. A backward contraction step in a G'_i invalidates all the G_j's. Validation of a G_i consists of copying to G_i all the polynomials whose most recent form in G'_i is not in G_i.

In [16], an implementation on a CM-5 with 128 nodes is described. The reported experiments display near linear, linear, or even superlinear speedups and the algorithm is said to outperform previous implementations in shared memory (e.g. [49]). A comparison with [26] is not available. The basic choices in the implementation as described in [16], however, refer only to the algorithm without backward contraction, so that the reader does not know how the presence of backward contraction influences the implementation.

Discussion

Among these four implementations of Buchberger algorithm, the first two, [49] and [44], realize parallelism at the clause level. The two distributed algorithms, [26] and [16, 17], feature some parallelism at the search level: each process has its own version of the basis, polynomials are distributed among the processes and forward contraction tasks are subdivided accordingly.

The transition-based approach of [16, 17] is the most advanced in terms of backward contraction. The choice of a coarser granularity of parallelism appears crucial in implementing backward contraction. The idea is to allow backward contraction without imposing a global synchronization among the processes whenever a basis is modified locally by backward contraction. This is achieved by letting the processes work *independently*: each process may work while its basis is invalidated and the successive validation is used to communicate to the process the effects of backward contraction at the other nodes. Other aspects of the method, however, lean toward medium-grain parallelism. The processes are not entirely asynchronous: the validations also play the role of synchronization points, since a process performing validation actually *waits* till all the requested polynomials arrive. The suggested presence of a *coordinator* process, which arbitrates invalidations, may indicate that the processes are not allowed to broadcast their messages, invalidating others' basis, in a totally independent fashion. The messages for load balancing may also reduce the independence of the processes by introducing more synchronization points. The algorithm seems to generate many small-sized messages for load balancing. This use of message-passing reflects the choice of the CM-5 as target machine: this type of message-passing is feasible in medium-grain parallelism on a multiprocessor rather than in coarse-grain parallelism on a network.

If this invalidation-validation technique were to be applied to theorem proving, a very important question would be when to schedule validation: how often a process should suspend inferences for updating its database with respect to the others. The more frequently validation is done, the more frequently a process suspends the "creative" part of its work (e.g., generating new clauses) to do "book-keeping". The clause generation rate may become too low. In [17], validation is said to be performed "lazily" and "on demand", but it is not clear whether this means that a process should do validation only when it is otherwise idle. Lazy validation is feasible for the Buchberger algorithm, because backward contraction plays a smaller role in the Buchberger algorithm than in theorem proving, as we pointed out at the beginning of this section. In theorem proving, an exceedingly lazy validation scheme may not be appropriate. For instance, a process may generate a very good simplifier and add it to all the shadow sets. If validation is lazy, the other processes may not consider that simplifier till much later, perhaps after having generated so many redundant clauses that their performances is already irremediably compromised. In particular, a scheduler such that validation is done only when a process is otherwise idle, may postpone indefinitely a validation phase, and thus the diffusion of the effects of backward contraction, as the process is constantly busy with deduction.

5.4 Analysis of the parallelization of expansion-oriented strategies

Expansion-oriented strategies do not have most of the properties of subgoal-reduction strategies. The picture changes as follows:

- **Monotonically increasing database**:

  The database grows during the derivation because of expansion steps. Therefore, it is no longer possible to preprocess all the clauses at "compile time", before the derivation. Also, while a single clause in S_i can still be considered as read-only data, thanks to the absence of backward contraction, the whole component S_i is no longer read-only since expansion processes need write-access to add new clauses.

- **Conflicts**:

  - Read-write conflicts do not arise, because there is no backward contraction.
  - Write-write conflicts may appear as conflicts between forward contraction steps on a raw clause. In connection graph procedures, link resolution steps may cause write-write conflicts as they modify concurrently the structure of the graph.

    For conflicts between forward contraction steps, the hypotheses assumed by some subgoal-reduction strategies, such as non-overlapping equations, are no longer realistic. To circumvent the requirements such as non-overlapping, some interpreters of equational programming languages use sophisticated data structures and techniques to perform parallel rewriting (e.g., [32]). Once again, these techniques require some pre-processing of the simplifiers at compile time.

    In theorem proving, since the number of objects that need to be forward contracted is large (all raw clauses), and the database is ever growing, one cannot effectively perform any preprocessing. Consequently, it is difficult to take advantage of elaborate data structures to conduct parallelism in forward contraction.

    Thus, it may be more cost-effective to perform forward contraction sequentially. Indeed, in a theorem proving derivation, normalization is usually conceived as a single inference step.

While parallelism at the clause level and parallelism at the search level are still feasible, parallelism at the term or literal level becomes much less appealing, when moving from subgoal-reduction strategies to expansion-oriented strategies. The main reason is *scale*. In equational programming, the entire computation is a normalization of a single term. In theorem proving, a normalization is a small task with respect to the amount of work represented by the whole derivation. Then, the overhead of parallelism at the term level is likely to be excessive, defeating the advantages of parallelization.

6 Contraction-based strategies

In this section we overview a parallel transition-based implementation of Kunth-Bendix completion [53, 54], the parallel theorem prover ROO [39], and the *teamwork method* [23, 4].

6.1 Parallel transition-based Knuth-Bendix completion

The project described in [54] applies to Knuth-Bendix completion the **transition-based approach to parallel programming** of [53], whose application to the Buchberger algorithm was described in Section 5.3. In the application to completion, the transitions are basically the inference rules of the completion procedure. In fact, the list of transitions defined in [54] follows closely the presentation of Kunth-Bendix completion as a set of inference rules first given in [5]. Examples of such transitions are "normalizing an equation", "finding whether an equation is trivial", or "applying a rule to try to simplify all left-hand sides of rules". Each process executes the transitions according to the same scheduler. The basic scheduler described in [54] implements a simplification-first search plan, since it prescribes to select first the transitions performing simplification and to apply them until no longer applicable.

The database of equations and rewrite rules is kept in a shared memory, organized as first-in first-out queues of pointers to equations. For instance, the transition "generate all the critical pairs between two selected rules" add raw critical pairs to the $NewEqs$ queue. The transition "normalize an equation" selects an equation from the queue $NewEqs$ and puts its normalized form in the queue $NormEqs$. The usage of pointers allow to save some memory space by having equations occurring in more than one queue without being duplicated. Conflicts in the access to queues are avoided by using locks on the pointers to the front and the rear of each queue. Since the processes execute the same scheduler, the activities of different processes differ in the selection of data. For instance, let p_1 and p_2 be two processes that are both scheduled to execute a transition of type "find whether an equation is trivial". Process p_1 accesses the queue $NormEqs$ and extracts an equation. As soon as process p_1 has released the lock at the pointer to the front of $NormEqs$, process p_2 may get it and extract another equation. After the test, if the equations are not trivial, they are appended at the rear of the queue $NontrivEqs$.

This method has been implemented on a Firefly with 6 CVAX and on a Sequent. The data in [54] refer only to the first machine. In the theoretical description of the application of the transition-based approach to completion, each transition corresponds to an inference rule. The transitions actually implemented either apply an inference rule at most once to at most all the equations, or apply all inference rules that apply, as many times as possible, to a single selected equation. Thus parallelism is at the clause level.

The strong points of this system appear to be its simplification-first scheduler and a skillful implementation of the access to the shared queues. In particular, there are few critical regions and they are few instructions long. Also, equations are rewritten without locking. This is possible because the hardware of the shared memory used in the implementation guarantees that if a location is read and written at the same time, the read instruction observes either the value before the write instruction or the value after the write instruction, but not an intermediate, unfinished value. If the read instruction is issued by a transition performing an expansion inference and the write instruction by a transition performing simplification, it may happen that the expansion inference applies to the equation before the simplification. Similarly, two write instructions may try to access the same location at the same time and in such a case one write instruction may be lost.

According to the authors, these phenomena – the expansion of non-simplified premises and the loss of some rewrite steps – would not influence significantly the performances. The speedup reported in [54] is $\frac{4}{5}n$ for n processors on the dedicated Firefly. We should note, however, that the authors have considered only the basic version of Kunth-Bendix completion, rather than the Unfailing Knuth-Bendix procedure [6, 28], so that whenever an unoriented equation is generated, the strategy fails. Therefore the examples they tried are not significant problems in terms of theorem proving.

6.2 The ROO parallel theorem prover

The **ROO theorem prover** [39] is a parallel, shared memory version of the theorem prover Otter [40, 41] for first-order logic with equality. We need to first describe Otter.

Otter is a refutational, resolution-based theorem prover. The database of clauses is divided into two main components, the *Set of Support* (Sos) and the set of $Usable$ clauses. According to the Set of Support Strategy [51], each expansion inference step uses at least one parent from the Sos. Otter also features a $Demodulators$ list, which contains equations to be used as simplifiers, and a $Passive$ list, whose clauses are used for forward subsumption and unit-conflict only. (A unit-conflict step is a binary resolution step that generates the empty clause.)

We remark that Otter can simulate a wealth of strategies. For example, if $Usable$ contains the presentation and Sos contains the negation of the target, Otter generates a derivation which resembles backward reasoning: all the generated clauses are descendants of the negated target. If $Usable$ contains the negation of the target and Sos contains the presentation, the derivation resembles forward reasoning: it generates clauses from the presentation until it obtains one that resolves with the negated target, or one of its descendants, to give the empty clause. Intermediate variations may be obtained by different partitions of the input between Sos and $Usable$. In some problems, forward reasoning can also be realized by putting part of the presentation in $Usable$, part of the presentation in Sos, and the negation of the target in $Passive$. The resulting derivation is even more strongly oriented toward forward reasoning than one where the negated target is in Usable, because no clause except the empty clause may be generated from the input target.

Otter works by executing a basic loop. At each iteration, Otter selects a clause, termed *given clause*, from the Sos. Let $i_1, \ldots, i_n$ be the set of expansion inference rules in Otter. For each rule

i_k, $1 \leq k \leq n$, Otter executes two phases:

1. First, it generates all the clauses, $\psi_1 \ldots \psi_n$, that can be derived by rule i_k from the given clause and any clause in the $Usable$ list. Each newly generated clause is *preprocessed*, (forward contracted), right after having been generated. For instance, Otter preprocesses ψ_1 before generating ψ_2. If ψ_1 is not deleted during preprocessing, ψ_1, or possibly a reduced form of ψ_1, is appended to Sos.

2. Second, the clauses that have been just appended to Sos are applied to contract pre-existing clauses. This stage is called *postprocessing*, and it corresponds to backward contraction in our terminology.

After these two phases have been performed for all expansion rules in $i_1, \ldots, i_n$, Otter appends the given clause to $Usable$ and proceeds to the next iteration of the loop body.

This order of operations implements an almost simplification-first search plan. The reason why Otter is not strictly simplification-first is that only preprocessing, but not postprocessing, is performed after the generation of every single new clause. Postprocessing is performed only after the generation of the batch of new clauses, derivable from the given clause and the clauses in $Usable$, by each expansion rule.

The loop described above is the very basic mechanism of Otter. The prover features a wealth of options that allow the user to choose selections of inference rules, criteria to sort clauses in the Sos, criteria to retain or discard clauses, just to mention a few. The advantage of giving to the user so much leeway is that she/he can experiment with a variety of strategies. For instance, since not all combinations of options yield complete strategies, the user has the opportunity to play with incomplete strategies, which may be very interesting, as they may turn out to be especially efficient on specific problems.

The basic idea of the parallelization of Otter realized in ROO is to have several given clauses active in parallel, thereby realizing another instance of parallelism at the clause level. The authors call *Task A* the task of performing the body of the basic loop of Otter for a given clause. Thus in ROO there are many concurrent processes executing Task A on different given clauses. This parallelization, though, causes three main problems:

- Concurrent append to Sos: it happens whenever two processes executing Task A try to append to Sos their newly generated clauses. Two such processes compete for write-access to Sos in shared memory.

- Addition of redundant clauses to Sos: redundancy is introduced because as process p_1 appends its batch of new clauses S_1 to Sos and process p_2 appends its batch S_2, the clauses in S_1 are not reduced with respect to the clauses in S_2 and vice versa. In fact S_1 and S_2 may even contain identical clauses, and all these redundant clauses would be appended to Sos. This is an instance of conflict between parallel expansion and contraction.

- Concurrent deletion of clauses in Sos or $Usable$: process p_1, postprocessing its new clause ψ_1, and process p_2, postprocessing its new clause ψ_2, compete for write-access to Sos or

Usable with the purpose of deleting clauses contracted by ψ_1 and ψ_2, respectively. This is clearly a problem with parallel backward contraction.

The authors of ROO have tried to overcome these obstacles by delaying additions of clauses to *Sos* and deletions from *Sos* and *Usable*. A process executing Task A is not allowed to append its new clauses to Sos. It appends them to another list, termed *K-list*. Similarly, a process executing Task A is not allowed to delete clauses from *Sos* or *Usable* during postprocessing. Rather, it appends an identifier of the clause to be deleted to another list, called *To-be-deleted*. A different task, *Task B*, consists in selecting a clause from the K-list, repeating preprocessing and postprocessing for that clause, and finally appending it to *Sos*. If processes p_1 p_2 have generated two clauses ψ_1 and ψ_2, such that ψ_1 can contract ψ_2, both ψ_1 and ψ_2 end up in the K-list. If ψ_1 is extracted first, ψ_1 is applied to contract ψ_2 as part of the postprocessing of ψ_1. If ψ_2 is selected first, ψ_2 is contracted by ψ_1 as part of the preprocessing of ψ_2. In either case the redundant clause ψ_2 does not make it to the *Sos*. Also, the process executing Task B has the right to carry out deletions on *Sos* and *Usable* according to the contents of To-be-deleted. All processes obey the same scheduler, which prescribes to execute Task B, if the K-list is not empty and no other process is doing it, Task A otherwise. Thus, only one process is allowed to be executing Task B at any given time.

Being implemented on top of Otter, ROO inherits the efficiency in the basic operations and the high portability that are among the outstanding features of Otter. The experimental results are remarkable, as ROO achieves near-linear speedup on many problems, including difficult ones [39]. On the other hand, the performances of ROO show some irregularities: first, the results are unstable, that is, two executions of ROO on the same input may report different timings, as a result of the nondeterminism of the parallel computation. Second, the prover obtains superlinear speedup on certain problems, but is very disappointing on others. The latter are certain equational problems, where high amounts of backward simplification and backward subsumption are required. The problem is that in these examples Task B turns out to be a bottleneck. The K-list grows very long, and the single process performing Task B falls behind in moving clauses from the K-list to *Sos*. It follows that other processes scheduled to execute Task A starve for work, in the absence of clauses to be picked as given clause. This happens for instance if a very good simplifier ψ, which reduces most of the clauses in the database, is generated. During the postprocessing of ψ, almost all the clauses will be moved to the K-list, so that the process executing Task B is overwhelmed and the other processes are idle.

Another reason for concern is memory usage. The amount of memory required by ROO grows with the number of active processes. Although this is somewhat expected, poor utilization of the allocated memory has also been observed [39]. Otter maintains a list of available records for each data type, such as “clause” or “symbol table entry”. Whenever a clause is deleted, its record is put in the list of available records. Whenever a new clause is created, a record is taken from the list of available records, if it is not empty. Otherwise, memory is allocated from the operating system. Since most deletions are performed by Task B, the processes executing Task A may keep asking for allocation of new records, while the list of available records at the process executing Task B grows and remains unused. This problem is being addressed by having available records released by Task B made available to other processes.

6.3 The teamwork method

The **teamwork method** [23, 4] is another application to theorem proving of technologies developed for artificial intelligence systems. It distinguishes four types of processes: “experts”, “referees”, “specialized experts”, and a “supervisor”. The experts are in charge of performing the inferences. Each expert receives a copy of the input set of equations, and it proceeds by using the inference rules of the strategy to develop its own derivation independently from the other experts. Thus the teamwork method implements a form of parallelism at the search level. The derivations produced by different experts may be different, because different experts use different search plans. Periodically, all the experts halt, and the second family of processes, the referees, start. The referees evaluate the derivations according to several measures. Some measures apply to the latest state of a derivation, in other words, the database of equations the expert has generated so far. One very simple such measure is the number of equations in the database. More refined measures, akin to those in [2], involve heuristics to estimate how useful the equations in the database are, with respect to the purpose of reducing the target theorem. Other measures refer to the history of the derivation (e.g., the number of simplification steps performed so far). These criteria are used to rank the derivations and to extract “good” equations from the data bases the experts have produced. All these data are passed by the referees to the supervisor. The task of this process is to select the best derivation, the “winner”, based on the informations provided by the referees. The latest state of the best derivation is given as new database to all the experts. Also, “good” equations from other derivations may be added. Then all the experts restart from this common state. In addition, the experts may invoke specialized experts, processes devoted to a specific task such as normalization or constraint solving [2].

The teamwork method has some similarity with the DARES system, since both methods adopt coarse-grain parallelism in a distributed environment. On the other hand, they address different classes of strategies, that is, expansion-oriented for DARES and contraction-based for the team-work method. In DARES the input clauses are distributed among the agents, whereas all processes have all the input clauses in the teamwork method. The latter allows different processes to execute different search plans, whereas in DARES all agents utilize level-saturation. The control of communication is also very different. In DARES, an agent decides to communicate to others based solely on *local* information, that is, the state of its own derivation. In the teamwork method, the evaluation is done by the referees using *global* information, that is, the states of all the derivations. The overall computation proceeds by alternating phases where the experts work independently to phases where the referees and the supervisor reconstruct a common state.

The teamwork method has been implemented in the *DISCOUNT* theorem prover for equational logic, which implements strategies based on Unfailing Knuth-Bendix completion. The experimental results reported in [4, 23] are impressive, although limited to five problems on only two nodes. The basic idea in this approach is that the periodical evaluation of the derivations and the consequent construction of a best database gives a special edge. It allows to mix the results of different search plans or to *interleave* them, by selecting at one stage the results of a search plan and those of another one at the next. The authors claim that good speedups may be

[2]The description in terms of distinct processes is for clarity only; in practice, each deductive process behaves alternately as expert and as referee.

obtained by allowing some processes to follow unfair search plans, which may be very efficient for some derivations. A delicate issue with such a combination of search plans is the global fairness of the distributed derivation. In [4], it is explained that unfair processes can be tolerated, provided at least one is fair and the fair one is selected infinitely often as the "winner" by the referees. In practice, this means that after a certain number of rounds, a fair search plan need always be selected. On the negative side, when the experts replace their own database by the one indicated by the supervisor, a great amount of potentially useful work may be wasted. The quality of the measures employed by the referees is critical to contain such waste. Another cost is represented by the synchronization delay imposed on the experts to let the referees and the supervisor intervene. Such synchronization limits the capability of the method to exploit parallelism at the search level.

6.4 Analysis of the parallelization of contraction-based strategies

For contraction-based strategies, the dynamic character of the database and the incidence of conflicts increase even further, with respect to expansion-oriented strategies:

- **Highly dynamic database**:

  Because of **backward contraction**, the database S_i is no longer monotonic during the derivation: additions by expansion are intertwined with deletions by contraction. All clauses are repeatedly subject to contraction; hence all the items in the database need be accessible not only for reading but also for writing. It follows that there is **no read-only data**. Since each contraction step requires both read-access and write-access, the ratio of read-accesses versus write-accesses may be expected to be lower than in the expansion-oriented strategies. This is a factor against shared memory, where write-access is more expensive than read-access.

- **Conflicts**:

  In addition to write-write conflicts between contraction steps, read-write conflicts between expansion and contraction steps also appear, because of backward contraction.

The following table summarizes the basic elements of the analysis conducted so far for the three classes of strategies:

	Subgoal-reduction strategies	*Expansion-oriented strategies*	*Contraction-based strategies*
Size of the database	*small*	*very large*	*very large*
Dynamicity of the database	*static*	*monotonic*	*dynamic*
Preprocessing	*yes*	*no*	*no*
Read-only data	*yes*	*yes*	*no*
Specialized data structures	*yes*	*no*	*no*
Conflicts	*no*	*no*	*yes*

All the considerations against parallelism at the term level in expansion-oriented theorem proving apply even more strongly to the case of contraction-based strategies. Furthermore, the presence of backward contraction challenges also the applicability of parallelism at the clause level. A clause that is reduced by a backward contraction step should be tested for further contraction with respect to all the other clauses. Thus, each backward contraction step may induce many. If one works with parallelism at the clause level in shared memory, this avalanche growth of contraction steps may cause a write-bottleneck, since all the backward contraction processes ask for write-access to the database in shared memory. We call this phenomenon the **backward contraction bottleneck**. Not all the backward contraction processes may be served and an otherwise unnecessary sequentialization is imposed. The clauses that are supposed to be subject to backward contraction may not be made available for other tasks (e.g., expansion steps), so that these are delayed in turn.

This phenomenon can be observed for instance in the transition-based parallel Knuth-Bendix procedure of [54] and in ROO [39], which both realize parallelism at the clause level in shared memory. In ROO, the concurrent deduction processes are in conflict, if they try to access the list *Sos* (Set of Support) in shared memory to perform backward contraction. This is an instance of the backward contraction bottleneck. In order to avoid this bottleneck, it is established that the concurrent processes do not do backward contraction. The clauses that need to be contracted are stored in a separate list, called *K-list*. This list is accessed by a single process in charge of executing backward contraction. But the backward contraction bottleneck reappears, since the K-list grows too long and the single backward contraction process becomes the bottleneck.

The source of the problem is the choice of a granularity (i.e., the clause level), which is too fine for contraction-based theorem proving. Therefore, a solution is to adopt a coarser granularity and realize parallelism at the search level. Among the methods we surveyed, the distributed Buchberger algorithm of [16, 17] and the teamwork method are examples in this direction.

6.5 Parallelism at the search level

Parallelism at the search level tries to realize an intuitive idea of *parallel search.* A sequential deductive process may search along one path only, whereas many concurrent processes may search in parallel along different paths. For this intuition to be productive, it seems desirable that the parallel processes *do not overlap.* If they do, it is likely that they are in fact exploring the same paths in the search space. If the concurrent processes search along the same paths, the advantage of having more processes rather than one is wasted. On the other hand, processes that do not overlap, and search different portions of the search space seem more likely to find a solution faster than by sequential search. Thus, the concurrent deductive processes should be *largely independent, asynchronous*, and cooperate *loosely.* Each process searches for a proof by developing its own derivation and as soon as one of them halts successfully, the whole process halts. The search space is partitioned among the processes by distributing the clauses and possibly subdividing the inferences. Because the search space is partitioned, the processes need to cooperate by exchanging data.

6.5.1 Agreement of data

Parallelism at the search level may be realized in principle in either shared or distributed memory. However, *distributed memory* is preferable, because it implements naturally the requirement that the data sets of the deductive processes are physically separated. The concurrent processes may access data for both reading and writing in a totally independent, asynchronous fashion. The costs are represented by the amount of memory and the communication delay of message-passing. For the latter, we observe that one important source of the need for message-passing in distributed systems is the maintainance of *agreement.* This property is also called *consistency* in the terminology of distributed databases and *coherence* in the terminology of parallel architectures. It means that if a datum is duplicated (e.g., one copy at p_i and one copy at p_j), the two copies need to *agree*: whenever the copy at p_i is modified, the copy at p_j should be updated accordingly as soon as possible. In many applications of distributed systems, agreement is necessary for the operations of the system to be correct. Automated deduction is *not* such an application. If we have two copies of a clause φ, one at p_i and one at p_j, and the copy at p_i is simplified to φ', the two clauses φ and φ' are *logically equivalent.* It follows that the completeness of a theorem proving strategy is not hindered if agreement is not strictly enforced. It is a matter of performance, and not of correctness, whether and how promptly the copy of φ at p_j should be updated. The fact that agreement is not necessary is a point in favor of distributed memory for theorem-proving applications, since enforcing agreement may be expensive.

Since most of the activities in contraction-based deduction are better implemented as intrinsically independent, asynchronous activities, and agreement of data is not critical, it is more suitable to parallelize contraction-based strategies in a distributed environment. However, there is an exception: forward contraction. The simplification steps in a normalization process are tightly cooperating processes, to such an extent that the whole normalization process may be conceived as a single inference step. Thus, it may be useful to have all the simplifiers stored in one place, e.g. in a shared memory component. Therefore, one interesting architecture for parallelizing contraction-based strategies is to use a distributed shared-memory machine.

As the last example of the paper, we present the *Clause-Diffusion* method, which can be implemented in a purely distributed configuration or a mixed configuration with predominantly distributed memory and an additional shared-memory component.

6.6 The Clause-Diffusion methodology

The basic idea in this approach [9, 10] is to have a deductive process running at each node of a distributed environment and *to partition the search space* among such processes. The search space is determined by the input clauses and the inference rules. At the clauses level, the input and the generated clauses are distributed among the nodes. Thus, a fundamental component of a Clause-Diffusion strategy is an *allocation algorithm*, which decides where to allocate a clause. Once a clause ψ is assigned to process p_i, ψ becomes a **resident** of p_i. In this way each node p_i is allotted a subset S^i of all the clauses. The union of all the S^i's, which are not necessarily disjoint, forms the current *global data base.* Each process is responsible for applying the inference rules of the strategy to its residents, according to the search plan.

Since the global database is partitioned among the nodes, no node is guaranteed to find a proof using only its own residents. To assure that a solution will be found when one exists, the nodes need to communicate to each other their residents in form of messages. Partitioning the search space makes cooperation necessary. These two basic ideas, subdivision of the problem and cooperation, differentiate radically the Clause-Diffusion method from the teamwork method, where each deductive process has a copy of the entire problem, and thus processes compete, rather than cooperate. The messages carrying the clauses are called **inference messages**. Each process sends the inference messages for its own residents and uses the received inference messages to perform inferences with its residents. The local database at each node contains at any stage of the derivation both resident clauses and visiting clauses, which came in as inference messages. Thus, one should keep in mind that the physical subdivision of the data set (i.e., whether a clause is stored at a node) is different from the logical subdivision (i.e., whether a clause belongs to a process).

The communication of inference messages may be realized in different ways. In a purely distributed system, inference messages are implemented as messages, which may be routed or broadcast. Depending on the broadcasting algorithm, a node may forward copies of the inference message to different nodes, while still retaining a copy for its own inferences. Thus, there may be several inference messages, all carrying the same clause, active at different nodes. In a system with a shared-memory component, the exchange of inference messages may be realized through the shared memory, by implementing send/receive as write/read in shared memory. Through inference messages, each clause is progressively "diffused" to all the processes, hence the name Clause-Diffusion.

The assignment of clauses to processes is also used to partition the search space at the inference level. Using the paramodulation inference rule as an example of expansion step, one may establish that visiting clauses are paramodulated *into* the residents, but not vice versa. This restriction distributes the expansion steps among the nodes and also prevents a systematic duplication of steps: without such restriction, each paramodulation step between two residents ψ_1 of p_1 and ψ_2 of p_2 would be performed twice, once when ψ_1 visits p_2 and once when ψ_2 visits p_1. Other expansion inference rules fit naturally in this pattern [9, 10].

So far we have not considered contraction. Indeed the Clause-Diffusion methodology applies also to expansion-oriented strategies, according to the basic mechanisms described above. In a contraction-based strategy, an expansion step should be performed only if all the premises are fully reduced, at least with respect to the local database. To ensure this, the method requires that each processor keep its residents as reduced as possible. When an inference message is received at a node, it is also subject to (forward) contraction. If it has not been deleted, it is used to contract the residents (backward contraction). Only afterwards is an inference message allowed to perform expansion. In other words, while the expansion steps are subdivided based on the ownership of the clauses, each process uses all the clauses it has access to to perform as much contraction as possible. The motivation is that contraction serves the purpose of keeping the database always at the minimal, and thus it is not productive to restrict it by the ownership criterion.

To keep data as reduced as possible, contraction needs to be done with respect to the global data base. Thus, the Clause-Diffusion method features a number of *distributed global contraction schemes* to enable a node to perform contraction with respect to a distributed set of clauses. After a raw clause ψ is reduced to ψ' according to a global contraction scheme, the process that reduced ψ to ψ' executes the allocation algorithm for ψ'. It may retain it as its resident or send it as **new settler** to be a resident at another node.

A distributed global contraction scheme enables a process to contract a clause, regardless of whether the clause is a resident, a message or a raw clause, with respect to the global data base $\bigcup_{i=1}^{n} S^i$ (n is the number of processes). Since the global data base is distributed, no process has direct access to the global data base as a whole. The distributed global contraction schemes of [9, 10] provide each process with access to an approximated version SH of the global database, termed *image set*. Each process uses the elements in the image set as simplifiers to perform contraction. If a shared-memory component is available, the image set SH is held in the shared memory. Otherwise, each node in the distributed environment stores an image set: process p_i running at the i-th node accesses the image set SH^i. In both cases, no write-bottleneck occurs, because the clauses being rewritten by contraction are held in the local memories of the nodes.

An image set is an approximation in the sense that it is not guaranteed to be updated with respect to contraction. If a resident $\psi \in S^i$ is reduced to ψ' at p_i, the image set accessible by another process p_j may still contain ψ, the old "image" of that resident. Since the elements in the image sets are used as simplifiers, not as parents in expansion step, the delay in updating the image sets with respect to contraction does not contradict the contraction-first policy of contraction-based strategies. Several techniques to generate and update image sets, with or without shared memory, are described in [9, 10, 11]. The inference message that are used to allow expansion steps between remote parents are also employed in the generation and maintenance of image sets, so that distributed global contraction does not require additional messages.

A description of all the components of the Clause-Diffusion methodology (e.g., distributed global contraction, distributed allocation, and message passing), is beyond the scope of this paper. We refer to [9] for a complete coverage of Clause-Diffusion, to [11] for its implementation, to [8] for a formal treatment of distributed derivations and to [10] for a study comprising all these aspects.

7 Architectural consideration in the parallelization of strategies

In this section we investigate the types of architectures suitable for parallelizing different classes of strategies.

In general, the finer the granularity the more tightly coupled the processors should be. A similar principle can be envisioned for the employment of memory. If a strategy has significant amount of common data often read by processors, then a computer system with shared memory is better suited. On the other hand, if common data also require write-access, then it is probably better to simply keep copies in different memory location to avoid bottlenecks.

Since goal-oriented strategies can fully utilize elaborate preprocessing, shared-memory machines are suitable for implementing such strategies. Multiprocessors with a large number of nodes, each with small memory, and fast communication speed have also been considered for building the data structure necessary for preprocessing, for example [25, 32].

Shared-memory machines are also suitable for parallelizing expansion-oriented strategies. By keeping a common database accessible by all nodes, it is easy to read data and perform expansion inferences, as long as concurrent read is allowed. Such a shared database is also proper for performing forward contraction, since existing data are not replaced or deleted. We should mention, however, that the system DARES [22] implements expansion strategies in a distributed environment.

Contraction-based strategies provide the most interesting challenge in terms of the choice of architecture. In principle, distributed systems are better suited for such strategies since otherwise the write-bottleneck often occurs. The main complexities introduced by distributed memory are the communication latency and the duplication of data. Therefore, we envision the best architecture for such strategies to be a distributed-memory system with a shared memory component. By implementing message passing through the shared memory, one can reduce the cost of communication as well as providing a global data base of simplifiers for forward and backward contraction. Although the Clause-Diffusion method put the potential use of such an architecture into consideration in its design, due to the scarcity of machines featuring both distributed and shared memory, we have not seen any such implementation.

8 Discussion

8.1 Parallelization of deduction strategies

We presented an analysis of the parallelizability of deduction strategies, with a study of *backward contraction* and a notion of *parallelism at the search level*. In our analysis, deduction methods are classified in three categories: *subgoal-reduction strategies*, *expansion-oriented strategies* and *contraction-based strategies*. The main difference between expansion-oriented strategies and contraction-based strategies is that the latter feature **backward contraction**, while the former do not. Then, we defined three types of parallelism in logical inferences: *parallelism at the term level* (fine grain), *parallelism at the clause level* (medium grain), and *parallelism at the search level* (coarse grain). Most of the existing works in parallel deduction exploit parallelism at the term or clause level. At these levels, two concurrent inference steps may be in *conflict* to access common premises.

In derivations by subgoal-reduction strategies, the database containing the presentation, e.g. a logic program, is *small* and essentially *static*, because all steps in a derivation consist in reducing a goal to subgoals. Conflicts do not arise or may be prevented by preprocessing the clauses in the database at "compile-time", that is, before the derivation. Thanks to these conditions, subgoal-reduction strategies are amenable to all three types of parallelism.

Expansion-oriented strategies generate derivations where the database is *very large* and *monotonically increasing*, because of expansion. Thus, it is not possible to prevent conflicts by preprocessing all clauses at "compile-time" and a limited amount of conflicts (i.e., write-write conflicts in forward contraction) appears. Parallelism at the term level, such as parallel rewriting in forward contraction, becomes less cost-effective.

In derivations by contraction-based strategies, the database is *large*, because of expansion, *highly dynamic*, and *not monotonically increasing*, because of backward contraction. Furthermore, backward contraction applies to clauses that are already being used as parents of expansion steps, causing read-write conflicts between the backward contraction steps and the expansion steps. Under these conditions, parallelism at the clause level is also likely to cause too much overhead. Thus, parallelism at the search level seems the most suitable.

In summary, as we go from subgoal-reduction strategies through expansion-oriented strategies to contraction-based strategies, the database becomes more and more dynamic; the incidence of conflicts increases and, correspondingly, the appropriate granularity of parallelism grows. Some existing parallel theorem provers [39, 54] execute contraction-based strategies with parallelism at the clause level. According to our analysis, these approaches suffer from a negative phenomenon, which epitomizes the drawbacks of a too fine-grained parallelism in the presence of backward contraction. We termed this problem the **backward contraction bottleneck**. In shared memory it appears as a write bottleneck, due to the avalanche growth of write-access requests from backward-contraction processes.

Thus, in order to parallelize contraction-based strategies, one should employ **parallelism at the search level**. Some elements of parallelism at the search level appear in [4, 9, 17, 22, 26]. By parallelism at the search level, we mean concurrent, asynchronous processes searching in parallel for a solution. As soon as one of them succeeds, the whole process succeeds. Unlike in fine- and medium-grain parallelism, each process is given a large portion of the data (e.g., a fairly large set of clauses) and can develop its own derivation. The databases of the concurrent processes are physically separated (distributed memory). Therefore, no two concurrent inference steps are physically in conflict: two concurrent inference steps that use the same clause logically, access two distinct physical copies of the clause. The cost of preventing physical conflicts by using separated databases is the duplication of clauses. Our estimate is that it is better to let the processes proceed eagerly in parallel, generating additional redundant clauses and deleting them afterwards, rather than synchronize the processes, thus forcing them to wait, in order to avoid conflicts.

8.2 Comparison of parallel deduction methods

Within the above framework, we surveyed several approaches to the parallelization of deduction, including term rewriting, Prolog technology theorem proving, resolution-based theorem proving, connection graph procedures, the Buchberger algorithm, Knuth-Bendix completion, and contraction-based theorem proving. Our survey is not intended to be exhaustive, but rather to provide material for our analysis and to show that our framework applies to exisiting methods. A more conventional survey can be found in [43, 48].

The field of parallel theorem proving is so new, that most of the methods we surveyed have been implemented only as experimental prototypes. In this paper, we aimed at illustrating the procedures, more than the concrete systems implementing them. Accordingly, we chose to refer to the literature for more details on the systems and their performances, rather than collecting here such data. In fact, we feel that a mere comparison of performances, while interesting, is not sufficiently informative at this stage of the research in parallel deduction. The set of methods we described is so etherogeneous, that a straightforward comparison is hardly feasible. Some approaches have different applications, for example, theorem proving versus programming. Within theorem proving, some systems parallelize strategies of different nature, for example, subgoal-reduction strategies versus contraction-based strategies. These strategies represent different philosophies in theorem proving and pose different problems to the designer of a parallel prover. Because of this diversity, it is not possible to establish which method is best. Even within the same class of strategies, it seems premature to identify the best methods. We preferred to indicate strengths and weaknesses in the description of each method.

As a concluding remark, we should point out that we consider the parallelization of deduction strategies only at its infancy. Our goal in writing this paper is to stimulate interest in this area, rather than giving any authoritative conclusions. It is our hope that there will be many more exciting developments, far surpassing what we have described in this paper, in the near future.

References

[1] S.Anantharaman and J.Hsiang, Automated Proofs of the Moufang Identities in Alternative Rings, *Journal of Automated Reasoning*, Vol. 6, No. 1, 76–109, 1990.

[2] S.Anantharaman and N.Andrianarivelo, Heuristical Criteria in Refutational Theorem Proving, in A.Miola (ed.), *Proceedings of the Symposium on the Design and Implementation of Systems for Symbolic Computation*, Capri, Italy, April 1990, Springer Verlag, Lecture Notes in Computer Science 429, 184–193, 1990.

[3] O.L.Astrachan and D.W.Loveland, METEORs: High performance theorem provers using model elimination, in R.S.Boyer (ed.), *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Kluwer Academic Publisher, 1991.

[4] J.Avenhaus and J.Denzinger, Distributing Equational Theorem Proving, in C.Kirchner (ed.), *Proceedings of the Fifth Conference on Rewriting Techniques and Applications*,

Montréal, Canada, June 1993, Springer Verlag, Lecture Notes in Computer Science 690, 62–76, 1993.

[5] L.Bachmair, N.Dershowitz and J.Hsiang, Orderings for Equational Proofs, in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, 346–357, Cambridge, Massachussets, June 1986.

[6] L.Bachmair, N.Dershowitz and D.A.Plaisted, Completion without failure, in H.Aït-Kaci, M.Nivat (eds.), *Resolution of Equations in Algebraic Structures*, Vol. II: Rewriting Techniques, 1–30, Academic Press, New York, 1989.

[7] L.Bachmair and H.Ganzinger, Non-Clausal Resolution and Superposition with Selection and Redundancy Criteria, in *Proceedings of Logic Programming and Automated Reasoning*, Springer Verlag, Lecture Notes in Artificial Intelligence 624, 273–284, 1992.

[8] M.P.Bonacina and J.Hsiang, On fairness in distributed deduction, in P.Enjalbert, A.Finkel and K.W.Wagner (eds.), *Proceedings of the Tenth Symposium on Theoretical Aspects of Computer Science*, Würzburg, Germany, February 1993, Springer Verlag, Lecture Notes in Computer Science 665, 141–152, 1993.

[9] M.P.Bonacina and J.Hsiang, The Clause-Diffusion methodology for distributed deduction, to appear in D.A.Plaisted (ed.), *Fundamenta Informaticae*, Special Issue on Term Rewriting Systems.

[10] M.P.Bonacina, Distributed Automated Deduction, Ph.D. Thesis, Department of Computer Science, State University of New York at Stony Brook, December 1992.

[11] M.P.Bonacina and J.Hsiang, Distributed Deduction by Clause-Diffusion: the Aquarius Prover, in A.Miola (ed.), *Proceedings of the Third International Symposium on Design and Implementation of Symbolic Computation Systems*, Gmunden, Austria, September 1993, Springer Verlag, Lecture Notes in Computer Science 722, 272–287, 1993.

[12] S.Bose, E.M.Clarke, D.E.Long and S.Michaylov, Parthenon: A parallel theorem prover for non-Horn clauses, *Journal of Automated Reasoning*, Vol. 8, N. 2, 153–182, April 1992.

[13] R.Boyer and S.Moore, *A Computational Logic*, Academic Press, New York, 1979.

[14] B.Buchberger, An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimensional Polynomial Ideal, (in German), Ph.D. thesis, Department of Mathematics, University of Innsbruck, Austria, 1965.

[15] B.Buchberger, History and Basic Features of the Critical-pair/Completion Procedure, *Journal of Symbolic Computation*, Vol. 3, 3–38, 1987.

[16] S.Chakrabarti and K.A.Yelick, Implementing an Irregular Application on a Distributed Memory Multiprocessor, in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993.

[17] S.Chakrabarti and K.A.Yelick, On the Correctness of a Distributed Memory Gröbner Basis Algorithm, in C.Kirchner (ed.), *Proceedings of the Fifth Conference on Rewriting Techniques and Applications*, Montréal, Canada, June 1993, Springer Verlag, Lecture Notes in Computer Science 690, 77–91, 1993.

[18] C.L.Chang and R.C.Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[19] P.D.Cheng and J.Y.Juang, A Parallel Resolution Procedure Based on Connection Graph, in *Proceedings of the 6th Conference of the American Association for Artificial Intelligence*, 13–17, 1987.

[20] S.C.Chou, Proving and discovering theorems in elementary geometries using Wu's method, Ph.D. Thesis, Department of Mathematics, University of Texas at Austin, 1985.

[21] E.M.Clarke, D.E.Long, S.Michaylov, S.A.Schwab, J.-P.Vidal and S.Kimura, Parallel Symbolic Computation Algorithms, Technical Report CMU-CS-90-182, School of Computer Science, Carnegie Mellon University, October 1990.

[22] S.E.Conry, D.J.MacIntosh and R.A.Meyer, DARES: A Distributed Automated REasoning System, in *Proceedings of the 11th Conference of the American Association for Artificial Intelligence*, 78–85, 1990.

[23] J.Denzinger, Distributed knowledge-based deduction using the team work method, Technical Report, Department of Computer Science, University of Kaiserslautern, 1991.

[24] N.Dershowitz and N.Lindenstrauss, An Abstract Concurrent Machine for Rewriting, in H.Kirchner, W.Wechler (eds.) *Proceedings of the Second Conference on Algebraic and Logic Programming*, Nancy, France, October 1990, Springer Verlag, Lecture Notes in Computer Science 463, 318–331, 1990.

[25] J.A.Goguen, S.Leinwand, J.Meseguer and T.Winkler, The Rewrite Rule Machine, 1988, Technical Monograph PRG-76, Oxford University Computing Laboratory, August 1989.

[26] D.J.Hawley, A Buchberger Algorithm for Distributed Memory Multi-Processors, in *Proceedings of the International Conference of the Austrian Center for Parallel Computation*, Linz, Austria, October 1991, Springer Verlag, Lecture Notes in Computer Science.

[27] C.M.Hoffmann and M.J.O'Donnell, Programming with Equations, *ACM Transactions on Programming Languages and Systems*, Vol. 4, N. 1, 83–112, January 1982.

[28] J.Hsiang and M.Rusinowitch, On word problems in equational theories, in Th.Ottman (ed.), *Proceedings of the Fourteenth International Conference on Automata, Languages and Programming*, Karlsruhe, Germany, July 1987, Springer Verlag, Lecture Notes in Computer Science 267, 54–71, 1987.

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

[30] D.Kapur and H.Zhang, RRL: a Rewrite Rule Laboratory, in E.Lusk, R.Overbeek (eds.), *Proceedings of the Ninth International Conference on Automated Deduction*, Argonne, Illinois, May 1988, Springer Verlag, Lecture Notes in Computer Science 310, 768–770, 1988.

[31] O.Kaser, S.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 Conference on LISP and Functional Programming*, 335–344, 1992.

[32] C.Kirchner and P.Viry, Implementing Parallel Rewriting, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 123–138, 1992.

[33] D.Knuth and P.Bendix, Simple word problems in universal algebra, in J.Leech (ed.), *Computational Problems in Abstract Algebra*, 263–297, Pergamon Press, 1970.

[34] R.E.Korf, Depth-first iterative deepening: an optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 97–109, September 1985.

[35] R.Kowalski, A proof procedure using connection graphs, *Journal of the ACM*, Vol. 22, No. 4, 572–595, 1975.

[36] R.Loganantharaj, Theoretical and implementational aspects of parallel link resolution in connection graphs, Ph.D. Thesis, Department of Computer Science, Colorado State University, 1985.

[37] R.Loganantharaj and R.A.Müller, Parallel Theorem Proving with Connection graphs, in J.Siekmann (ed.), *Proceedings of the Eighth Conference on Automated Deduction*, Oxford, England, July 1986, Springer Verlag, Lecture Notes in Computer Science 230, 337–352, 1986.

[38] D.W.Loveland, A simplified format for the model elimination procedure, *Journal of the ACM*, Vol. 16, N. 3, 349–363, July 1969.

[39] E.L.Lusk and W.W.McCune, Experiments with ROO: a Parallel Automated Deduction System, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 139–162, 1992.

[40] W.W.McCune, OTTER 2.0 Users Guide, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, March 1990.

[41] W.W.McCune, What's New in OTTER 2.2, Technical Memorandum ANL/MCS-TM-153, Argonne National Laboratory, Argonne, Illinois, July 1991.

[42] J.Schumann and R.Letz, PARTHEO: A High-Performance Parallel Theorem Prover, in M.E.Stickel (ed.), *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, Germany, July 1990, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 28–39, 1990.

[43] J.Schumann, Parallel theorem provers – An overview, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 26–50, 1992.

[44] K.Siegl, Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages, Master thesis and Technical Report N. 90-54.0, RISC-LINZ, November 1990.

[45] M.E.Stickel, A Prolog technology theorem prover, *New Generation Computing*, Vol. 2, N. 4, 371–383, 1984.

[46] M.E.Stickel and W.M.Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, August 1985, 1073–1075.

[47] M.E.Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, *Journal of Automated Reasoning*, Vol. 4, 353-380, 1988.

[48] C.B.Suttner and J.Schumann, Parallel Automated Theorem Proving, in L.Kanal, V.Kumar, H.Kitano and C.B.Suttner (eds.), *Parallel Processing for Artificial Intelligence*, Elsevier, 1994.

[49] J.-P.Vidal, The Computation of Gröbner Bases on A Shared Memory Multiprocessor, in A.Miola (ed.), *Proceedings of International Symposium on the Design and Implementation of Symbolic Computation Systems*, Capri, Italy, April 1990, Springer Verlag, Lecture Notes in Computer Science 429, 81–90, 1990. Full version available as Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, August 1990.

[50] D.H.D.Warren, An abstract Prolog instruction set, Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.

[51] L.Wos, D.Carson and G.Robinson, Efficiency and completeness of the set of support strategy in theorem proving, *Journal of the ACM*, Vol. 12, 536–541, 1965.

[52] W.T.Wu, Mechanical theorem proving in elementary geometry and differential geometry, in the *Proceedings of the 1980 Beijing Symposium on Differential Geometry and Differential Equations*, Vol. 2, 125–138, 1982.

[53] K.A.Yelick, Using abstraction in explicitly parallel programs, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, available as Technical Report MIT/LCS/TR-507, July 1991.

[54] K.A.Yelick and S.J.Garland, A Parallel Completion Procedure for Term Rewriting Systems, in D.Kapur (ed.), *Proceedings of the Eleventh International Conference on Automated Deduction*, Saratoga Springs, New York, June 1992, Springer Verlag, Lecture Notes in Artificial Intelligence 607, 109–123, 1992.