Experiments with subdivision of search in distributed theorem proving

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

Abstract

We introduce the distributed theorem prover *Peers-mcd* for networks of workstations. Peers-mcd is the parallelization of the Argonne prover EQP, according to our Clause-Diffusion methodology for distributed deduction. The new features of Peers-mcd include the AGO (Ancestor-Graph Oriented) heuristic criteria for subdividing the search space among parallel processes. We report the performance of Peers-mcd on several experiments, including problems which require days of sequential computation. In these experiments Peers-mcd achieves considerable, sometime super-linear, speed-up over EQP. We analyze these results by examining several statistics produced by the provers. The analysis shows that the AGO criteria partitions the search space effectively, enabling Peers-mcd to achieve super-linear speed-up by parallel search.

1 Introduction

Distributed deduction is concerned with the problem of proving difficult theorems by distributing the work among networked computers. The motivation is to study new forms of distributed reasoning, and to improve the performance, and therefore the applicability, of automated reasoning systems.

In previous work, we defined a methodology for distributed deduction called *deduction by Clause-Diffusion* [6, 9]. It launches concurrent, asynchronous, deductive processes to search in parallel the space of the problem. Each process executes a theorem-proving strategy, develops its own derivation, and builds its own database of clauses. The search space is subdivided among the processes, which cooperate to find a proof, and communicate by broadcasting the clauses they generate. As soon as one of them succeeds, all processes halt. A key issue for this type of approach is how to subdivide the search space, in such a way that the parallel search succeeds faster than the sequential search.

In this paper we present a new approach to this problem. The main difficulty is that the search space is infinite, and therefore we can have only partial knowledge. We have a finite amount of data, e.g. the clauses generated so far, and we are trying to induce a successful subdivision of the search space by subdividing the clauses that generate it. In these conditions, it is impossible to partition the search space into disjoint parts, as it would be desirable ideally to maximize the advantage of parallel search. Therefore, the concurrent searches will *overlap* to some extent. The goal is to limit this overlap.

At each step of the derivation we consider the generated clauses as a *search graph*, the visited portion of the entire search graph [11]. Within this search graph, we can trace for each clause φ its *ancestor-graph*. The ancestor-graph contains information on how φ is connected to other clauses, and how the different processes contributed to generate φ. We reason that we can use this information to assign clauses to processes, in such a way to limit the overlap in an intuitive sense. Thus, we define *Ancestor-Graph Oriented* (AGO) criteria for the allocation of clauses. Since our knowledge of the graph is only partial, these criteria are heuristics.

The problem of dividing the search space in theorem proving is largely new. The methods which choose fine-grain parallelism are not concerned with parallel search and decomposition of the search space. Their objective is the parallelization of the inner operations of a strategy. Recent developments in this direction include the system PaReDuX [13, 15], and the fine-grained concurrent completion of [26]. Surveys of previous approaches can be found in [8, 38]. An approach closer to Clause-Diffusion is the Team-Work method of [1, 2, 17]. The system of [14] aims at integrating features of PaReDuX with features of Team-Work. Similar to Clause-Diffusion, Team-Work prescribes that concurrent, deductive processes search the space of the problem, generating independent derivations. The processes execute different search plans, and periodically merge their databases. Team-Work features a variety of criteria to retain "good" data, discard "bad" data, and form a new database, which is given to all processes for the next phase of independent deduction. Clause-Diffusion also allows the processes to apply different search plans, but the emphasis is on the subdivision of the space. Team-Work does not subdivide the space and emphasizes the use of parallelism to combine the strengths of different search plans.

In this paper we present the following contributions:

- A new Clause-Diffusion theorem prover called *Peers-mcd*.
- The Ancestor-Graph Oriented criteria.

*Supported in part by the National Science Foundation with grant CCR-94-08667.

In E. Kaltofen and M. Hitz (eds.), *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, Wailea, Maui, Hawaii, July 1997. ACM Press, 88–100, 1997.

- The experiments with Peers-mcd in Robbins algebra and their analysis.

Peers-mcd is the successor of *Peers* [12]. It is new in the following components:

- Improved parallelization methodology: Peers-mcd implements Modified Clause-Diffusion [6], which refines in many ways the first Clause-Diffusion methodology [9] implemented in Peers.
- New underlying sequential prover: Peers incorporated an early ancestor of EQP of November 1992, while Peers-mcd parallelizes EQP0.9 of June 1996 [30], which solved the Robbins challenge [31].
- New criteria for the partition of the search: Peers-mcd features the AGO criteria that the previous Clause-Diffusion prototypes did not have.
- New communication software: Peers used p4 [16], while Peers-mcd uses the MPI standard for message passing [20], which is a successor of p4.

In the experiments, Peers-mcd exhibits impressive speed-up's over EQP. The set of problems includes hard lemmas that are part of the solution of the Robbins challenge, and take several days of CPU time on a single processor. Peers-mcd is the second prover after EQP to prove them, and its performance is the fastest so far.

We analyze the experiments in detail in terms of many statistics. We point out the factors that contribute to explain the speed-up. The data on clauses generated show that the AGO criteria are effective in subdividing the search in these experiments. A discussion of future work closes the paper.

The speed-up of Peers-mcd in these experiments is sometime *super-linear*. This may be disconcerting, because a parallel algorithm may achieve at most linear speed-up with respect to the best sequential algorithm. We emphasize that achieving super-linear speed-up by Clause-Diffusion does not violate this law. Inference systems for theorem proving are non-deterministic. In order to obtain a deterministic procedure, an inference system I is coupled with a search plan Σ, that controls the application of the inference rules, to give a theorem-proving strategy $\mathcal{C} = \langle I; \Sigma \rangle$. If one designs a parallel implementation of $\mathcal{C}$, we expect the speed-up to be at most linear with respect to the best sequential implementation of $\mathcal{C}$. Assume, on the other hand, that we design a new parallel strategy $\mathcal{C}' = \langle I; \Sigma' \rangle$, which has the same inference system, and a new parallel search plan Σ'. Then, the speed-up with respect to the best sequential implementation of $\mathcal{C}$ can be super-linear, because the search plan is different.

The Clause-Diffusion methodology is precisely an approach to generating a parallel strategy $\mathcal{C}' = \langle I; \Sigma' \rangle$ from a sequential strategy $\mathcal{C} = \langle I; \Sigma \rangle$. Assume that Σ has a criterion ζ to select inference rules, and a criterion ξ to select clauses as premises [10]. The domain of ξ is the entire database of existing clauses. In the Clause-Diffusion strategy $\mathcal{C}'$, ξ is replaced by a ξ', such that the domain of choice for every parallel process p_i is restricted to the subset of existing clauses assigned to p_i. Therefore, the generated search and often the generated proof are different, and a super-linear speed-up may happen.

2 The distributed theorem prover Peers-mcd

Peers-mcd implements the Modified Clause-Diffusion method of [6]. The name "peers" emphasizes that all deductive processes are peers, with no master-slave relation. In this section we summarize the operations of the prover. We recall that *forward contraction* is the contraction of a newly generated clause, while *backward contraction* is the contraction of a clause that was already kept, given an identifier, and used as premise of inferences.

Peers-mcd is written in C and uses MPI [20] for message passing. When invoked on a problem, it starts N processes, if N is the number of processes specified on the command line. Each process runs on a different workstation, and all processes execute the same code. The workstations involved are specified by the user in a "procgroup" file [20]. Process p_0 reads the problem from standard input and broadcasts to the other processes the input clauses and the input parameters. Similar to Otter [29], the prover has a default strategy that the user can modify by assigning values to parameters in the input file. Thus, we have N processes $p_0, \ldots, p_{N-1}$ executing the strategy on the given problem.

The component of a Clause-Diffusion strategy that determines the subdivision of the search space is the *allocation algorithm*. Whenever a process p_i generates a clause, by either expansion or backward-contraction, the clause is subject to forward contraction. If it is not deleted, p_i executes the allocation algorithm for this clause, say φ. If the algorithm returns value j (possibly $j = i$), φ is assigned to p_j. Its identifier is $< j, i, k >$, if it is the k-th clause generated by p_i and assigned to p_j. Then p_i broadcasts φ to all processes as an *inference message*, and also keeps it.

The assignment of clauses to processes determines the subdivision of the search space as follows. For expansion, take the paramodulation inference rule. A process executes a paramodulation step only if it owns the clause paramodulated into. For backward-contraction, we distinguish between deletion, such as in subsumption, and replacement, such as in simplification. The former is unrestricted. For the latter, Modified Clause-Diffusion establishes that if ψ can be simplified, only the owner of ψ generates its normal form. The other processes merely delete ψ.

These elements are sufficient for the following presentation of the AGO criteria and the experiments. We refer to [6] for the complete description of Modified Clause-Diffusion, and to [9] for a wider treatment of various strategies within the Clause-Diffusion methodology.

3 The AGO criteria

A key feature of Peers-mcd are the *ancestor-graph oriented criteria* (AGO criteria) for the allocation of clauses. We define the ancestor-graph and then some of these heuristics.

For a given set S of input clauses, and inference system I of the strategy, the search space contains all the clauses that can be derived from S by I in any number of steps. In [11], we proposed a model of search where the search space is represented as a *search graph*, with vertices labelled by the clauses, and arcs representing the inferences. More precisely, the search graph is a hypergraph, because inference steps generally have multiple premises. If an inference step with premises $\varphi_1, \ldots, \varphi_n, \varphi_{n+1}$ generates φ_{n+2} and deletes φ_{n+1}, the search graph contains a hyperarc

$$(\varphi_1, \ldots, \varphi_n; \varphi_{n+1}; \varphi_{n+2}).$$

As an example, consider a normalization step where φ_{n+1} is replaced by its normal form φ_{n+2} with respect to the simplifiers $\varphi_1, \ldots, \varphi_n$. For an expansion inference the hyperarc has the form $(\varphi_1, \ldots, \varphi_n; \varphi_{n+1})$, where φ_{n+1} is generated

from the premises. A complete definition of search graph can be found in [11].

Given a clause φ, its *ancestor-graph* is obtained by proceeding backward from φ in the search graph. If φ is an input clause, its ancestor-graph is φ itself (one vertex labelled by φ). If there is a hyperarc $e = (\varphi_1, \ldots, \varphi_n; \varphi_{n+1}; \varphi)$, and $t_1, \ldots, t_n, t_{n+1}$ are ancestor-graphs of $\varphi_1, \ldots, \varphi_n, \varphi_{n+1}$, the graph with root φ connected by e to the subgraphs $t_1, \ldots, t_n, t_{n+1}$ is an ancestor-graph of φ.

Intuitively, if the parallel processes search the same part of the space, parallel search may not help. Thus, a goal of an allocation criterion is *to limit the overlap of the parallel searches*. The ancestor-graphs of generated clauses represent the available information on the search space during the derivation. Therefore, we use this information to try to prevent the processes from overlapping too much.

An ancestor-graph represents a way to generate a clause. Since a clause can be generated in many ways, the ancestor-graph of a non-input clause in a search graph is not unique. However, for a specific variant of a clause generated in a specific derivation, the ancestor-graph is unique. Furthermore, the data structures in EQP already contain information on the ancestors of clauses, for the purpose of extracting the proof from the derivation. (Note that the generated proof is an ancestor-graph of the empty clause.) It follows that the AGO criteria use information which is kept by the prover anyway, with no additional storage requirement. In the following, we consider two AGO criteria, named *parents* and *majority*.

3.1 The AGO criterion parents

The *parents* criterion is based on the intuition that clauses which have the same parents should be assigned to the same process. The idea is that clauses which have the same parents are spatially close in the search graph. If we assign such clauses to different processes, we may increase their overlap.

Given a clause φ, the *parents* criterion takes the identifiers of the parents of φ. These identifiers are given as input to a function f, which returns the number of the process φ should be assigned to. Since f is a function (unique image), all clauses with same parents receive the same process. Input clauses do not have parents, and therefore are assigned to process p_0.

The criterion is parametric with respect to f and the notion of parents. For the former, the current implementation of Peers-mcd uses addition modulo N, where N is the number of processes. For the latter, we observe that a clause can be generated essentially in two ways: by expansion (e.g., paramodulation), or by backward contraction (e.g., normalization). Therefore, Peers-mcd implements two variants of the parents criterion:

1. The *para-parents* criterion considers only paramodulation parents. If φ is generated from ψ_1 and ψ_2 by paramodulation, φ is assigned to the process given by $id(\psi_1) + id(\psi_2)$ *mod* N. Otherwise, φ is assigned to process p_0.

2. The *all-parents* criterion considers both paramodulation parents and backward-contraction parents. If φ is generated by paramodulation from ψ_1 and ψ_2, it is assigned to $id(\psi_1) + id(\psi_2)$ *mod* N. If φ is generated by backward contraction of ψ, φ is assigned to $id(\psi)$ *mod* N. Otherwise, φ is assigned to process p_0.

These two variants differ in the treatment of clauses generated by backward-contraction. The para-parents criterion assigns them to one process, while the all-parents criterion uses the backward-contraction parent to distribute them.

Simplifiers applied during forward or backward contraction are not considered as parents. Assume that φ is generated by paramodulation from ψ_1 and ψ_2, and normalized to φ' by simplifiers $\alpha_1, \ldots, \alpha_n$ in the forward-contraction phase. Then, φ' is assigned to $id(\psi_1) + id(\psi_2)$ *mod* N, without involving $\alpha_1, \ldots, \alpha_n$. Similarly, if ψ is backward-contracted to φ by simplifiers $\beta_1, \ldots, \beta_n$, φ is assigned to $id(\psi)$ *mod* N. Applied simplifiers are excluded because their inclusion would defeat the purpose of the criterion. Two clauses generated by the same paramodulation parents may be normalized by different sets of simplifiers. A criterion which keeps the latter into account would allocate them to different processes.

3.2 The AGO criterion majority

The *majority* criterion is also based on an intuitive notion of proximity in the search space. Rather than considering proximity between clauses (e.g., clauses that have the same parents are close), it considers proximity between clauses and processes.

From the point of view of the processes, assume that a process p_i is searching a certain area of the search space. Then, the clauses whose ancestor-graphs lay mostly in that area should be assigned to p_i, so that p_i continue doing that area, while the other processes do the others. If we assign such clauses to another process p_j, we may cause p_i and p_j to overlap too much.

From the point of view of the clauses, a clause φ should be assigned to a process which is active near φ. Otherwise, we would augment the overlap of the processes. Since what is known of the search space around φ is its ancestor-graph, we look for a process whose search overlaps the most with the ancestor-graph of φ.

The majority criterion implements this idea by assigning φ to a process which owns a majority of its ancestors. More precisely, the algorithm retrieves the ancestor-graph t of φ. It assigns to each process p_i a number of votes equal to the number of clauses in t owned by p_i. A process which gets a majority of the votes wins and gets φ. Ties are broken arbitrarily.

A rationale for counting owned ancestors is that a process performs the inferences whose premises it owns. Thus, a process which owns a majority of t is likely to have executed most of the arcs in t, so that indeed that process has traversed this part of the search graph.

This mechanism does not apply to input clauses, because input clauses have no ancestors. The majority criterion cannot deal with input clauses by assigning them to p_0 like the parents criterion does. The reason is that if all input clauses are assigned to a process, all clauses will be assigned to that process, since it will have a majority of ancestors for all clauses. (This does not happen with the parents criterion, because it uses the identifier of the clauses, not only their owner). Therefore, this algorithm applies some other criterion to input clauses. For instance, it uses the un-informed criterion *rotate*, which picks process $p+1$ *mod* N, if there are N processes and p was chosen last. More generally, Peers-mcd has a parameter *switch-owner-strat* for this purpose. If *switch-owner-strat* has value n, each process uses the rotate criterion n times, and then switches to the majority criterion.

Since the ancestor-graph of a clause contains both expansion and contraction ancestors, applied simplifiers are

included in the ancestors count. For instance, if φ is generated from ψ_1 and ψ_2 by paramodulation, and then forward-contracted to φ' by simplifiers $\alpha_1, \ldots, \alpha_n$, the clauses ψ_1, ψ_2, φ, $\alpha_1, \ldots, \alpha_n$ and all their ancestors are ancestors of φ'.

4 The theorem prover EQP

Each process in Peers-mcd executes the code of the prover EQP of W. McCune [30]. This system implements contraction-based strategies for equational reasoning, with associativity and commutativity (AC) built-in.

The strategies use a *complete simplification ordering* $\succ$ on terms [21]. In EQP, this ordering is a *recursive path ordering* [18], built from a total precedence on the function symbols. The user gives the precedence in the input file, and the prover completes it if it is not total. Function symbols have lexicographic status by default, and multiset status if specified in the input file. If the theory declares AC operators, a weaker simplification ordering replaces the complete simplification ordering: $s \succ t$ if $|s| > |t|$ and no variable has more occurrences in t than in s.

An input or generated equation $s \simeq t$ is oriented into $s \to t$, if $s \succ t$. If $s \# t$ (neither $s \succ t$ nor $t \succ s$), EQP "flips" the equation: it stores it as two pairs $s \approx t$ and $t \approx s$. (We follow the convention of [19] of using $\simeq$ for an equation viewed as an unordered pair, and $\approx$ for an equation viewed as an ordered pair.)

The expansion inference rule is *paramodulation* [34], with the AC unification algorithm of [22, 36] if there are AC operators. EQP applies paramodulation to the left side of stored rewrite rules or equations. Thus, if $s \# t$, EQP considers for paramodulation s in $s \approx t$ and t in $t \approx s$.

The contraction inference rules are *simplification*, *subsumption*, and *functional subsumption* [21]. Simplification applies as simplifiers only the rewrite rules, not the equations.

EQP implements three refinements of paramodulation. The first one is the *superposition of unoriented equations* of the Unfailing Knuth-Bendix completion of [4, 21]. It is called "ordered paramodulation" in EQP, and it is controlled by the option *ordered-paramod*. If this rule is selected, paramodulation applies to s in $s \approx t$ only if $s\sigma \not\preceq t\sigma$, where σ is the most general unifier (mgu) of the paramodulation step. This inference rule cannot be used with AC operators in EQP, because the prover does not implement a complete simplification ordering modulo AC.

The second restriction is *blocking* [3, 25, 27, 35] (option *prime-paramod*), which prevents a paramodulation step whose mgu contains reducible terms.

The third special rule is *basic paramodulation* [5, 32] (option *basic-paramod*), which excludes terms introduced by substitutions. Such terms are *not basic*. The completeness of basic paramodulation requires that also simplification is restricted to basic terms. Since simplification is unrestricted in EQP, the strategies with basic paramodulation are incomplete.

The combination of basic paramodulation and simplification is not the only source of incompleteness in EQP. The *max-weight* parameter inherited from Otter [29] allows to discard clauses based on size. The *AC-superset-limit* parameter limits the number of generated AC-unifiers. If either one of these parameters is used, the strategy is incomplete.

EQP features two search plans, the *given clause algorithm* and the *pair algorithm*. The given-clause algorithm is the same of Otter. It selects a given clause φ, makes all inferences between φ and the previously given clauses, and repeats. The pair algorithm selects a pair of clauses, performs all inferences between them, and repeats. By default, clauses/pairs to be given are sorted by increasing length, so that the shortest one is picked next. This amounts to *best-first search* with the length of the clause as heuristic evaluation function. The *pick-given-ratio* parameter, also inherited from Otter, allows one to add some *breadth-first search*. If the value of *pick-given-ratio* is k, the search plan selects the oldest candidate every k choices.

We adopt the conventions of [30] for the names of the strategies. The name of a strategy is composed of the name of the inference system, the value of the *pick-given-ratio* (n if it is not set), and the name of the search plan (given or pair). The "starting" inference system includes (AC)-paramodulation, (AC)-simplification, and subsumption.

Other systems are defined similarly, e.g., "basic" is the same as "starting" with basic paramodulation in place of paramodulation. The "super0" system is the same as the starting system, with value 0 assigned to the parameter *AC-superset-limit*. For this parameter, the lower is the value, the more AC-unifiers are excluded. Thus, value 0 implies the highest degree of pruning of AC-unifiers. Examples of names of strategies are *start-n-pair*, *basic-4-giv*, and *super0-n-pair*.

5 Robbins algebra

Robbins algebra is known in theorem proving for the challenge problem of showing that every Robbins algebra is Boolean. The problem dates back to 1933, when E. V. Huntington demonstrated [23, 24] that the equation

$$n(n(x) + y) + n(n(x) + n(y)) = x \tag{1}$$

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

$$n(n(x + y) + n(x + n(y))) = x \tag{2}$$

is also sufficient. A proof of the conjecture was not found. Since it was not known whether such an algebra is Boolean, an algebra defined by equation 2 with associativity and commutativity of $+$ was called a *Robbins algebra*. Equations 1 and 2 were called *Huntington axiom* and *Robbins axiom*, respectively. The problem of determining whether a Robbins algebra is Boolean remained open, and eventually became known in the theorem proving community as the *Robbins problem* [41].

The Argonne approach to this problem has been to decompose it by finding intermediate conditions: show that the condition implies the Huntington axiom, and the Robbins axiom implies the condition. Two such conditions are

$$\exists x \exists y \; x + y = x, \tag{3}$$

$$\exists x \exists y \; n(x + y) = n(x), \tag{4}$$

called First Winker Condition and Second Winker Condition, respectively [30, 39, 40]. S. Winker proved by hand that each of the two conditions is sufficient to make a Robbins algebra Boolean [39, 40]. However, these lemmas remained beyond the possibilities of theorem provers.

Thanks to its combination of contraction-based strategies, and fast algorithms for indexing of clauses [28] and AC-matching, EQP has been the first theorem prover to demonstrate these lemmas [30]. A proof is obtained by showing that the First Winker Condition implies the Huntington

axiom, and that the Second Winker Condition implies the First Winker Condition. In the next section we describe the experiments on these lemmas with Peers-mcd.

6 The experiments

We report the performances of Peers-mcd on some AC problems, including three formulations of the lemma "The First Winker Condition implies the Huntington axiom," and the lemma "The Second Winker Condition implies the First Winker Condition." The data show that:

- For all the reported experiments, there exists an AGO allocation criterion which leads Peers-mcd to speed-up with respect to EQP.
- For all the experiments with the strategy *start-n-pair*, there exists an AGO allocation criterion which enables some configuration of Peers-mcd to obtain *super-linear speed-up* with respect to EQP.
- To our knowledge, the proofs found by Peers-mcd for the lemmas in Robbins algebra are the fastest so far.

The strategies used in these experiments are *start-n-pair*, *basic-n-pair* and *super0-n-pair*. We selected the strategy *start-n-pair* for two reasons. First, it is the default strategy of EQP, except for using the pair algorithm in place of the given clause algorithm. Since the pair algorithm performs much better than the given clause algorithm in the sequential experiments in Robbins algebra (e.g., [30]), we can assume *start-n-pair* as the default strategy for these problems. Second, *start-n-pair* has a complete inference system, except for deletion by weight. It allows us to observe the effect of parallelization on a complete strategy, rather than the combination of the effect of parallelization and the effect of other enhancements that make the prover faster at the expense of completeness.

We selected *basic-n-pair* and *super0-n-pair*, because they give the best sequential performance on some of the problems. Unlike *start-n-pair*, however, *basic-n-pair* and *super0-n-pair* have incomplete inference systems. In addition to the incompleteness due to deletion by weight, *basic-n-pair* is incomplete because of the combination of basic paramodulation and unrestricted simplification, and *super0-n-pair* is incomplete because it ignores most AC-unifiers.

We use the allocation criteria rotate, para-parents, all-parents, and majority. For the latter, *switch-owner-strat* has value 20 for all the experiments, so that the first 20 allocation choices are made by rotate, and all the following ones by majority. This value is high enough to include all input clauses for many problems, including those considered here.

The experiments were conducted on a cluster of workstations HP 715, each with 64M of memory, connected by a local area network. The reported run times are *average wall-clock times*, expressed in seconds. The CPU time, or *user time* in Otter and EQP, is commonly used to compare sequential theorem provers. For a sequential theorem prover, the difference between the wall-clock time and the CPU time is negligible in most cases. This is because the single process terminates almost immediately after finding the proof. For a parallel program the wall-clock time may be higher than the CPU time, because it includes also the time spent in clearing pending messages and halting all the processes. Therefore, the wall-clock time is more appropriate for a comparison including parallel programs.

We use the following abbreviations in the tables. For the allocation criteria (abbreviated "crit."), R is rotate, P is para-parents, AP is all-parents, and M is majority. For the strategies (abbreviated "strat."), s-n-p is *start-n-pair*, s0-n-p is *super0-n-pair*, and b-n-p is *basic-n-pair*. For the provers, EQP is EQP0.9, N-P is N-Peers, that is, Peers-mcd with N processes. For lack of better names, we denote the problems by the names of the corresponding files in our directories. The times for Robbins3 with s-n-p and R, P, M, Robbins8 with s-n-p and R, P, M, Robbins9 with s-n-p, R, P, M, and limited to 4 nodes, appear also in the system description [7].

6.1 A classical problem in ring theory

We begin with the problem of showing that $x^3 = x$ implies commutativity in the theory of rings (file name **x3**). The first mechanical solution of this problem was one of the early successes of contraction-based strategies [37].

We use first the *super0-n-pair* strategy, and *max-weight* equal to 60. Peers-mcd improves over EQP with all allocation criteria. With four processes, and the AGO criteria of type parents, it solves the problem in only 5 sec (speed-up 8.2 and efficiency 2):

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*	*4-P*
s0-n-p	R	41	42	16	10
s0-n-p	P	41	42	15	**5**
s0-n-p	AP	41	42	14	**5**
s0-n-p	M	41	42	19	7

If we omit the *AC-superset-limit*, and therefore consider all AC-unifiers, the sequential time is 143 sec. Peers-mcd with four nodes and the AGO criterion para-parents can do it in 22 sec (speed-up 6.5 and efficiency 1.6):

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*	*4-P*
s-n-p	R	143	133	137	50
s-n-p	P	143	133	86	**22**
s-n-p	AP	143	133	84	65
s-n-p	M	143	133	113	48

While in the first table there is little difference among the four criteria, in the second table para-parents behaves best for this combination of problem and strategy.

6.2 A simple lemma in Robbins algebra

We consider next the lemmas in Robbins algebra, in order of increasing difficulty. The first lemma consists in showing that a Robbins algebra which satisfies $\exists x \exists y \exists z\ n(x+y) = x+z$, satisfies also the Second Winker Condition (file name **Robbins7**). The following table shows the performance of EQP and Peers-mcd with strategy *start-n-pair* and *max-weight* equal to 30. The AGO criterion majority is the winner, matched by all-parents only with six and seven nodes:

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*	*4-P*	*6-P*	*7-P*
s-n-p	R	72	77	61	19	33	22
s-n-p	P	72	77	74	33	73	14
s-n-p	AP	72	77	83	41	6	8
s-n-p	M	72	77	**8**	9	8	**5**

The shortest time is 5 sec, obtained by the majority criterion with seven nodes (speed-up 14.4 and efficiency 2). The most efficient result is to find a proof in 8 sec with only two nodes (speed-up 9 and efficiency 4.5).

6.3 The First Winker Condition implies $\exists x\ x + x = x$

A common formulation (e.g., [30]) of the problem of showing that the First Winker Condition implies the Huntington axiom is to prove that the First Winker Condition implies $\exists x\ x+x = x$ (file name **Robbins3**). The companion lemma shows that the Huntington axiom follows if $\exists x\ x + x = x$ is assumed (file name **Robbins2**, also called r2 in [12], and RBA-2 in [30]). Either Otter or EQP can prove Robbins2 in less than 20 sec.

The first proof of Robbins3 was generated by an early prototype of EQP in November 1992: it took 7801 sec on a Sparc2, with strategy *start-n-pair* and *max-weight* = 30 [30]. An early version of Peers-mcd, which still incorporated the sequential code of November 1992, found a distributed proof in 706 sec in the summer of 1996. It used seven HP 715, *start-n-pair*, value 30 for *max-weight*, and the AGO criteria para-parents. For comparison, one node on the same hardware took 6815 sec (speed-up 9.6 and efficiency 1.3). We keep the value 30 for *max-weight* in all the following experiments.

In [30], EQP0.9 finds a proof in 4400 sec with *start-n-pair*, and 1902 sec with *basic-n-pair*, running on an RS/6000. For EQP, *basic-n-pair* is the best strategy on this problem, while *start-n-pair* is the best strategy among those with the "starting" inference system. The following table reports the results of the current versions of the provers on the HP 715:

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*	*4-P*	*6-P*
s-n-p	R	3705	3953	1349	1340	1631
s-n-p	P	3705	3953	933	915	**522**
s-n-p	AP	3705	3953	1227	851	608
s-n-p	M	3705	3953	997	1043	1187
b-n-p	R	1661	1617	1566	1646	1563
b-n-p	P	1661	1617	2021	698	**551**
b-n-p	AP	1661	1617	1598	700	**551**
b-n-p	M	1661	1617	1548	3233	1820

For the sequential prover, *basic-n-pair* is approximately twice as good as *start-n-pair* on this problem. According to [30], this ratio is specific of this lemma, and does not generalize to other AC problems.

The distributed prover does not preserve the relationship between the strategies on this problem. With two processes, *start-n-pair* is better than *basic-n-pair* for all allocation criteria. With four processes, *basic-n-pair* is better than *start-n-pair* with the allocation criteria of type parents, but it is worse with rotate and majority. The combination of strategy *basic-n-pair*, allocation criterion majority, and four nodes behaves badly on this problem. The configuration with six processes gives the best results: 522 sec with *start-n-pair* and para-parents (speed-up 7 and efficiency 1.2), and 551 sec with *basic-n-pair* and either para-parents or all-parents (speed-up 3 and efficiency 0.5).

6.4 The First Winker Condition implies $\exists y \forall x\ x + y = x$

A more general formulation of the lemma (file name **Robbins8**) assumes the First Winker Condition and proves the target theorem $\exists y \forall x\ x + y = x$, which implies $\exists x\ x + x = x$ trivially. With this input, it is sufficient to employ only two nodes to succeed in 485 sec (speed-up 7.5 and efficiency 3.7):

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*
s-n-p	R	3649	3809	2220
s-n-p	P	3649	3809	1591
s-n-p	AP	3649	3809	1086
s-n-p	M	3649	3809	**485**

This result represents a super-linear speed-up also with respect to the time of EQP0.9 with *basic-n-pair*, which is 1605 sec.

6.5 Direct proof that the First Winker Condition implies the Huntington axiom

We gave to the theorem provers also the original formulation of the lemma, where the target theorem is the Huntington axiom itself (file name **Robbins9**). This makes the problem harder:

Crit.	*EQP*	*1-P*	*2-P*	*4-P*	*6-P*	*8-P*
R	4857	4904	3557	1177	3766	2675
P	4857	4904	1437	2580	3934	2158
AP	4857	4904	1534	2588	1819	**519**
M	4857	4904	872	**709**	707	1809

The strategy is *start-n-pair* (omitted in the table for reasons of space). The fastest configuration of Peers-mcd on this input takes 519 sec (speed-up 9.3 and efficiency 1.1), with eight nodes and the AGO criterion all-parents. The AGO criterion majority is more efficient, because it solves the problem in 709 sec with 4 processes (speed-up 6.8 and efficiency 1.7). EQP0.9 with *basic-n-pair* takes 2067 sec, so that the above times by Peers-mcd are the fastest, although the speed-up with respect to the time of 2067 sec is only sub-linear.

6.6 The Second Winker Condition implies the First Winker Condition

The problem is to show that a Robbins algebra which satisfies $\exists x \exists y\ n(x+y) = n(x)$ satisfies also $\exists x \exists y\ x+y = x$ (file name **Robbins10**).

We summarize first the results from [30]. EQP found the first proof of this lemma in February 1996 after running for 1,081,324.76 sec, or approximately 12.5 days on a 486DX2/66. The prover used a strategy similar to *start-n-pair* and value 34 for *max-weight*. The experiment was repeated later on an RS/6000, with *start-n-pair* and the same *max-weight*. The CPU time was 608,688.75 sec, or approximately 7.04 days, and the wall-clock time was 621,494 sec, or almost 7.2 days. Another attempt running the strategy *basic-n-pair* was terminated after seven days of computation and no proof. This suggests that *basic-n-pair* is not better than *start-n-pair* on this problem, which is much more difficult than Robbins3.

The following table reports the results of our experiments, also with strategy *start-n-pair* and *max-weight* 34:

Strat.	*Crit.*	*EQP*	*1-P*	*2-P*
s-n-p	R	518,393	520,336	265,145
s-n-p	P	518,393	520,336	**10,162**
s-n-p	AP	518,393	520,336	**10,023**
s-n-p	M	518,393	520,336	161,779

518,393 sec = 5 days, 23 hr, 59 min, and 53 sec
520,336 sec = 6 days, 32 min, and 16 sec
265,145 sec = 3 days, 1 hr, 39 min, and 5 sec
161,779 sec = 1 day, 20 hr, 56 min, and 19 sec
10,162 sec = 2 hr, 49 min, and 22 sec
10,023 sec = 2 hr, 47 min, and 3 sec

All AGO criteria show super-linear speed-up. The performances of para-parents and all-parents are spectacular: with all-parents the speed-up is almost 52 (51.72), and the efficiency almost 26 (25.86). The numbers for para-parents are very close, with speed-up 51 and efficiency 25.5. Majority achieves super-linear speed-up 3.2, with efficiency 1.6. The un-informed criterion rotate yields speed-up 1.9 (almost linear) and efficiency 0.97.

The results with para-parents and all-parents are an impressive example of the potential of parallel search on large problems. The results with rotate and majority show the stability of our distributed theorem prover, which can run smoothly for days of CPU time.

In the following sections we analyze the experiments by examining the statistics generated by the theorem provers.

7 Statistics from the experiments

The EQP theorem prover produces several statistics for each run. In Peers-mcd, each process produces its own set of statistics. We choose two statistics, the number of generated clauses (*clauses generated*), and the number of kept clauses (*clauses kept*). The first one is a measure of the amount of work done by the prover. The second one is a measure of the effectiveness of contraction in deleting redundant clauses.

We recall that in Peers-mcd a process generates a clause, forward-contracts it, and, if it is not deleted, keeps it and broadcasts it as an inference message. Only the process which generated the clause increases the count of the clauses kept. All the other processes which receive it as an inference message do not. Otherwise, each clause would be counted N times, if there are N processes. Thus, the statistic *clauses kept* does not reflect the duplication due to the parallelization. We chose to do it in this way, because we want to preserve the original meaning of *clauses kept* as the number of clauses kept after contraction out of those generated. When reading the numbers of clauses kept by the Peers-mcd processes, one needs to keep in mind that those clauses are broadcast.

In addition to the statistics, EQP breaks down the run time into several components, which helps to find out in which activities the theorem prover spends the time. We shall see that most of the computation time is spent in contraction. This holds for both the sequential and the distributed prover, and throughout all experiments. This is expected, for at least two reasons. First, the problems are equational problems. Since equational replacement, implemented as well-founded simplification, is a natural way of reasoning with equations, it is also natural that the system spends most of its time in rewriting. Second, contraction-based strategies require that the database of equations is kept inter-reduced, and this generates a lot of backward-contraction work. On the other hand, it is very well-known that inter-reduction is a main reason of the power of these strategies, so that this requirement is essential.

Since most of the computation time is spent in contraction, we select the *demodulation time* and the *back-demod-find time* among the times reported by EQP and Peers-mcd. The first one is the time spent in normalization, including both forward and backward contraction. The second one is the time spent in finding clauses that can be backward-contracted.

Then we compare the sequential proof and the distributed proof. If they are different, we count how many clauses appear in both proofs, and how many do not, as a simple measure of the degree of difference between the two proofs. We also give the length of the proofs, defined as the number of clauses in the proof.

As a general remark, we observe that in all the experiments, all processes of Peers-mcd contribute clauses to the proof (clauses they own and clauses they generated). This shows that the proofs are produced in a truly distributed manner. Also, only one of the parallel processes finds the proof in each run, with extremely rare exceptions. This is a good sign in terms of low redundancy induced by parallelization, and effective subdivision of work.

The following tables contain the values for both statistics and times. The statistics are pure numbers, whereas the times are expressed in seconds. For the statistics, "gen." is *clauses generated*, "kept" is *clauses kept*, "found" is *proofs found*, and "length" is *proof length*. For the times, "demod-t" stands for *demodulation time*, "b-d-f-t" stands for *back-demod-find time*, "w-c-t" is the wall-clock time, and "cpu-t" is the CPU time, called *user time* in EQP. For the processes, assume that we are considering *6-Peers*, that is, Peers-mcd with six nodes. Then, *Peer0*, *Peer1*, *Peer2*, *Peer3*, *Peer4*, and *Peer5* denote the six parallel processes.

For each of the problems in Section 6 we compare a sequential execution by EQP and a parallel execution by Peers-mcd. We select the derivations from those used to determine the average wall-clock times of Section 6. Thus, the strategies, e.g., the values of *max-weight*, are as described in Section 6. Since the purpose is to explain the speed-up, for

each problem we select a configuration of Peers-mcd which exhibited the highest speed-up on that problem. In order to avoid repetitions, we give first the tables with the statistics, and we summarize all observations and comments in Section 8.

7.1 Comparison for the ring problem

We begin with the problem **x3** (Section 6.1) with strategy *super0-n-pair*. For Peers-mcd we consider a derivation with four nodes and allocation criterion para-parents:

	EQP	*Peer0*	*Peer1*	*Peer2*	*Peer3*
gen.	6053	1036	708	403	928
kept	55	14	24	2	6
found	1	0	0	0	1
length	28	N/A	N/A	N/A	28
w-c-t	42	4	4	6	4
cpu-t	37.36	3.42	4.28	4.10	3.54
demod-t	31.51	2.19	1.97	2.94	1.92
b-d-f-t	0.19	0.17	0.17	0.18	0.19

The sequential prover generates 6,053 clauses and retains 55 (0.9%). The distributed prover generates globally only 3,075 clauses, and retains 46 of them (1.5%). The successful process *Peer3* keeps only 6 clauses of the 928 it generates. Those 6 clauses are all part of the proof. The two proofs are the same except for two clauses.

7.2 Comparison for the lemma Robbins7

The following table compares one of the executions by *2-Peers* with the AGO criterion majority with a sequential execution by EQP:

	EQP	*Peer0*	*Peer1*
gen.	1122	111	210
kept	378	53	117
found	1	0	1
length	9	N/A	9
w-c-t	72	8	8
cpu-t	71.09	6.93	7.50
demod-t	46.10	1.85	2.59
b-d-f-t	19.61	3.71	3.82

The sequential and the distributed prover find the same proof, made of 9 clauses. Peers-mcd finds it by generating, and keeping, significantly fewer clauses: 221 clauses generated versus 1,122, and 170 clauses kept (77%) versus 378 (34%).

7.3 Comparison for the lemma Robbins3

The next two tables shows the statistics for the lemma Robbins3 of Section 6.3. We consider first the strategy *start-n-pair*. Therefore, we compare an execution by EQP and one by Peers-mcd with six nodes and the AGO criterion para-parents:

	EQP	*Peer0*	*Peer1*	*Peer2*
gen.	14198	1594	1371	1268
kept	2042	403	242	172
found	1	0	0	1
length	25	N/A	N/A	28
w-c-t	3685	494	494	493
cpu-t	3580.22	451.31	473.33	459.09
demod-t	2837.32	164.83	179.08	161.89
b-d-f-t	614.58	267.08	268.57	273.56

	Peer3	*Peer4*	*Peer5*
gen.	1327	1536	1376
kept	177	176	220
found	0	0	0
length	N/A	N/A	N/A
w-c-t	494	494	495
cpu-t	469.26	477.55	471.28
demod-t	170.46	189.06	171.62
b-d-f-t	276.54	270.59	277.41

The distributed derivation generates globally 8,472 clauses, and keeps 1,390 (16%), whereas the sequential derivation generates 14,198 clauses, and keeps 2,042 (14%). The generated proofs are different: the sequential proof comprises 25 clauses, the distributed proof is made of 28 clauses, and they have 20 clauses in common.

We consider next the statistics with the strategy *basic-n-pair*. The best behaviour of Peers-mcd is again with six nodes and the AGO criteria parents, so that we select one of the derivations by *6-Peers* with all-parents:

	EQP	*Peer0*	*Peer1*	*Peer2*
gen.	5609	1529	1390	1292
kept	1799	562	259	199
found	1	0	0	0
length	24	N/A	N/A	N/A
w-c-t	1629	511	511	518
cpu-t	1582.15	472.61	487.86	457.77
demod-t	1002.90	183.93	198.23	165.44
b-d-f-t	481.04	261.56	261.95	266.50

	Peer3	*Peer4*	*Peer5*
gen.	1485	1487	366
kept	157	115	52
found	1	0	0
length	30	N/A	N/A
w-c-t	511	511	522
cpu-t	485.69	472.03	503.28
demod-t	196.35	195.19	71.01
b-d-f-t	267.46	262.59	417.92

The most significant change affects the number of clauses generated by EQP. It drops from 14,198 with *start-n-pair* to 5,609 with *basic-n-pair*, as a consequence of the more restrictive, and also more radically incomplete, inference system. On the other hand, the total number of clauses generated by Peers-mcd decreases much less, from 8,472 with *start-n-pair* to 7,549 with *basic-n-pair*. Unlike all other experiments, the distributed theorem prover generates more clauses than the sequential theorem prover, and keeps a smaller percentage of them. Peers-mcd keeps 1,344 clauses (18%), while EQP keeps 1,799 clauses (32%). Similar to the execution with *start-n-pair*, the sequential proof and the distributed proof are different. The former is made of 24 clauses, and the latter of 30, with 20 clauses in common.

7.4 Comparison for the lemma Robbins8

For the lemma Robbins8 of Section 6.4 we compare a derivation by EQP and one by Peers-mcd with two nodes and the AGO criterion majority:

	EQP	*Peer0*	*Peer1*
gen.	14171	3303	2186
kept	2037	680	324
found	1	1	0
length	25	24	N/A
w-c-t	3628	446	448
cpu-t	3589.01	424.09	436.29
demod-t	2843.26	263.48	273.95
b-d-f-t	615.70	137.27	135.41

Here we find again that the distributed prover generates much fewer clauses than the sequential prover. Peers-mcd generates globally 5,489 clauses, and keeps 1,004 (18%), while EQP generates 14,171 clauses, and keeps 2,037 (14%). The sequential and distributed proofs contain 25 and 24 clauses, respectively, and have 20 clauses in common.

7.5 Comparison for the lemma Robbins9

For the lemma Robbins9 (Section 6.5), we report the statistics of a derivation by 4-Peers with the AGO criterion majority. This configuration of Peers-mcd was not the fastest, but it was the most efficient.

	EQP	*Peer0*	*Peer1*	*Peer2*	*Peer3*
gen.	25939	5047	5138	2826	2687
kept	2905	928	556	189	144
found	1	0	0	1	0
length	107	N/A	N/A	123	N/A
w-c-t	4902	705	704	704	704
cpu-t	4665.60	664.79	677.03	603.66	612.11
demod-t	3557.55	375.26	381.78	294.59	314.55
b-d-f-t	876.95	253.81	250.83	252.63	252.82

This table also shows a strong difference between the number of clauses generated by Peers-mcd (15,698 clauses, of which 1,817, or 12%, are kept) and EQP (25,939, of which 2,905, or 11%, are kept). Once again, the two proofs are different, and more so than in the previous lemmas. The distributed proof is made of 123 clauses, 55 of which appear also in the sequential proof, whereas the remaining 68 clauses do not. Thus, the two proofs have less than 50% of the clauses in common.

7.6 Comparison for the lemma Robbins10

For Robbins10, we consider a parallel derivation by *2-Peers* with the AGO criterion all-parents:

	EQP	*Peer0*	*Peer1*
gen.	354477	14712	12445
kept	3730	1941	980
found	1	1	0
length	24	24	N/A
w-c-t	518393	9865	9867
cpu-t	514257.39	9784.46	9793.00
demod-t	510740.25	8072.46	8169.17
b-d-f-t	2495.22	1511.59	1498.45

EQP generates 354,477 clauses and keeps 3,730 (1%). Peers-mcd generates globally only 27,157 clauses and retains 2,921 of them (11%). The sequential proof and the distributed proof have the same length (24 clauses) with 20 clauses in common.

8 Analysis of the experiments

We reported the statistics of Peers-mcd and EQP for seven derivations. In this section we use these data to interpret the results of the experiments. We consider the following aspects:

- number of clauses generated,
- percentage of clauses kept,
- usage of the contraction time,
- the proof,
- scalability, and
- work-load balance.

The first observation is that whenever there is super-linear speed-up, the sum of the numbers of clauses generated by the Peers is significantly smaller than the number of clauses generated by EQP. This happens for x3, Robbins7, Robbins3 with *start-n-pair*, Robbins8, Robbins9, and Robbins10. The case of sub-linear speed-up (Robbins3 with *basic-n-pair*) is also the one where Peers-mcd generates more clauses than EQP.

This means that the advantage of the distributed prover does not consist in searching a larger portion of the space by exploiting the power of more computers. Neither it is based on generating and processing faster the same clauses. Rather, it is due to the subdivision of the search space. The fact that each parallel process generates much fewer clauses than the sequential process shows that the partition of the search space is effective. Each parallel process searches a smaller space, which implies a shorter time to the proof.

The second observation is that whenever there is super-linear speed-up Peers-mcd keeps a higher percentage of clauses than EQP. On most problems the difference is small, at most four percentage points. For Robbins7 and Robbins10 the difference is higher: 77% versus 34% in Robbins7 and 11% versus 1% in Robbins10. On the other hand, in Robbins3 with *basic-n-pair* the situation is the opposite, as EQP retains a higher percentage of clauses than Peers-mcd (32% versus 18%). Since the two systems have the same contraction power (they have the same contraction rules, and the partition of contraction in Modified Clause-Diffusion affects the generation of clauses, not the deletion), a higher percentage of kept clauses does not indicate weaker contraction. Rather, it suggests that the search may be better focused. The most striking example is Robbins10, where the spectacular speed-up is accompanied by a difference of two orders of magnitude in number of clauses generated and one order of magnitude in percentage of clauses kept. In summary, the most efficient prover is the one that generates fewer clauses a higher percentage of which is not redundant.

We consider next the *demodulation time*, the *back-demod-find time*, and their sum, which we call *contraction time*. The parallel process of Peers-mcd have smaller contraction time than EQP in all analyzed derivations. This datum is related to those on clauses generated: if fewer clauses are generated, fewer clauses need to be normalized and kept inter-reduced. Also, Modified Clause-Diffusion subdivides the contraction inferences.

Peers-mcd and EQP may use their contraction time differently. For EQP, the *demodulation time* is higher than the *back-demod-find time*. On the other hand, for most problems (Robbins7, Robbins3 with either strategy, Robbins9, and Robbins10), the percentage of contraction time which is *back-demod-find time* is higher for the Peers-mcd processes than for the EQP process. In Robbins3 and Robbins7, the Peers-mcd processes mark a *back-demod-find time* higher than the *demodulation time*.

For sequential provers, the *demodulation time* is generally higher than the *back-demod-find time* because each clause found at *back-demod-find time* is then normalized. This is not necessarily true in a Clause-Diffusion prover. A process p_i can find a clause that can be backward-contracted and throw it away without normalizing it, because it is not one of its clauses in the partition. Thus, the *back-demod-find time* may be higher than the *demodulation time*. Remark that this partition may also help the distributed prover by delaying redundant data. Consider a process p_i which finds clauses that can be backward-contracted and discards them. Other processes will normalize those clauses, and broadcast their normal forms, which will reach p_i as well. However, if they are not needed for the proof, p_i may save time by not generating them right away, and succeed sooner.

We examine next *the proofs*. The proof of a theorem-proving problem is generally not unique. In all the derivations analyzed in Section 7 except Robbins7, the proofs found by EQP and Peers-mcd are different. In most cases, the two proofs have the majority of the clauses in common, except Robbins9. Since the problems and the strategies are the same, the difference in the proofs is due to parallel search. Parallel search traverses the space in a different manner, and therefore may find a different proof.

While parallel search makes super-linear speed-up possible, approaches based on parallel search may not fare as well in terms of *scalability*. For instance, it happens sometimes that the performance scales well up to a certain number of processes, e.g., 1, 2, 4, 6, but it becomes worse with 7 or 8. A pattern of this type suggests that the problem may not be hard enough to justify the use of so many workstations, so that subdividing the space further is counterproductive.

In other cases, the performance oscillates, e.g., *2-Peers* improves over EQP, but *4-Peers* does worse than *2-Peers*, and *6-Peers* speeds-up again. Or, both *4-Peers* and *6-Peers* do not improve, but *7-Peers* or *8-Peers* do. In these instances, we consider that the subdivision of the search space in Clause-Diffusion depends on the number of processes, because it is done by dynamic allocation of generated clauses during the derivation. Assume that we have two processes p_0 and p_1. When we add a third process p_2, the portions of the space assigned to p_0 and p_1 change with respect to what they were with two processes. The three searches developed by p_0, p_1, and p_2, are different than the searches developed by p_0 and p_1 when running as two processes. Since the result depends on the subdivision of the search, it may happen that two processes do better than four on a certain combination of problem and strategy.

Lastly, we consider *the balance of the work-load* among the Peers-mcd processes. If we measure the work-load by the number of generated clauses, we see that it is not always well-balanced, as some processes generate more clauses than others. The AGO allocation criteria do not attempt to balance the work-load. Since most of the derivations with somewhat unbalanced work-load show speed-up, it seems that balancing the work-load may not be as important for parallel search as it is for other forms of parallelization.

For instance, the table for Robbins9 shows that processes *Peer0* and *Peer1* generates and retain many more clauses than *Peer2* and *Peer3*. Process *Peer2* finds the proof. One possible interpretation is that many of the clauses handled by *Peer0* and *Peer1* are not needed for the proof. Thus, it may be that *Peer0* and *Peer1* take care of a lot of redundant work (redundant at least relative to the proof found), enabling *Peer2* to succeed sooner. It is interesting that Robbins9 is also the case where the distributed proof differs the most from the sequential proof.

In most of the analyzed derivations, however, the process which finds the proof has relatively high numbers of clauses generated and kept. Thus, the behaviour described above is possible, but not general. One may consider whether better results could be obtained by designing allocation criteria which take into account both the information from the search space explored so far (the ancestor-graphs) and the work-load of the processes. More generally, however, it can be discussed how notions of work-load and work-load balance apply to parallel computations where each process has an infinite amount of work available. It may be that we need to seek other notions of work-balance, or that the issue of work-balance is not key for parallel search in an infinite space.

9 Discussion

We have presented the Clause-Diffusion prover Peers-mcd, and shown that it achieves super-linear speed-up over its sequential counterpart EQP on some hard problems.

A fundamental aspect of the Clause-Diffusion paradigm of distributed deduction is the subdivision of the search space. For this purpose, Peers-mcd feature a new technique, the AGO criteria for the allocation of clauses to the processes. These criteria use the ancestor-graphs of generated clauses to limit the overlap of the parallel searches in an intuitive sense.

We have reported the performance of Peers-mcd on some problems, mostly in Robbins algebra. They include two lemmas which are part of the proof of the Robbins problem recently produced by EQP [31]. For all these experiments, at least one of the AGO criteria allows Peers-mcd to run faster than EQP. If the default strategy is used, the speed-up is super-linear for some configuration of Peers-mcd. To our knowledge, the proofs found by Peers-mcd for the two lemmas mentioned above are the fastest proofs generated so far.

We have analyzed the experiments in detail to explain the speed-up. The AGO criteria partition the search space of these problems effectively: each parallel process generates a smaller space than the sequential process. Since a higher percentage of the generated clauses turns out to be not redundant, the search is also better focused, and less time is spent in generating and deleting redundant clauses.

In summary, for all theorem-proving strategies, one can find theorems where they work well, and theorems where they do not (e.g., [30, 33]). Thus, high-performance theorem proving needs a variety of tools. We have shown that parallel search by distributed deductive processes is one of them.

10 Future work

We are studying parallelism in theorem proving as a new form of search. Traditionally, one expects parallelism to help solving large problems by making more computational power available. In our case, we are also interested in how parallelism may help by allowing to search the search space in new ways, different than the sequential ones.

We envision several directions for future work. At the implementation level, we plan to investigate more AGO criteria, and conduct more experiments. Also, we intend to interface Peers-mcd with a proof checker, in order to check the distributed proofs.

In addition to proof checking, we are interested in developing tools to compare proofs. Our manual comparison of proofs was limited to counting clauses and checking for their presence, but we did not check that the clauses were used in the same way. If the two proofs differ, even by a few clauses, different inference steps are involved. We think that at least part of this job should be automated. The motivation is to understand better how and why the distributed proof can be different than the sequential proof. In turn, this may shed more light on how the parallel search differs from the sequential search.

At the theoretical level, we plan to extend the model of search of [11] to parallel search. This model provides a notion of search complexity which is suitable for searches in infinite spaces. The next step will be to apply this model to analyze how parallelism may reduce the search complexity.

Acknowledgements

I would like to thank Bill McCune for making the source code of EQP0.9 available, and for answering my questions on EQP0.9.

References

[1] Avenhaus, J., and Denzinger, J. Distributing equational theorem proving. In *Proc. of the 5th RTA* (1993), C. Kirchner, Ed., vol. 690 of *LNCS*, Springer Verlag, pp. 62–76.

[2] Avenhaus, J., Denzinger, J., and Fuchs, M. DISCOUNT: a system for distributed equational deduction. In *Proc. of the 6th RTA* (1995), J. Hsiang, Ed., vol. 914 of *LNCS*, Springer Verlag, pp. 397–402.

[3] Bachmair, L., and Dershowitz, N. Critical pair criteria for completion. *J. of Symbolic Computation 6*, 1 (1988), 1–18.

[4] Bachmair, L., Dershowitz, N., and Plaisted, D. A. Completion without failure. In *Resolution of Equations in Algebraic Structures*, H. Ait-Kaci and M. Nivat, Eds., vol. II (Rewriting Techniques). Academic Press, New York, 1989, pp. 1–30.

[5] Bachmair, L., Ganzinger, H., Lynch, C., and Snyder, W. Basic paramodulation and superposition. In *Proc. of the 11th CADE* (1992), D. Kapur, Ed., vol. 607 of *LNAI*, Springer Verlag, pp. 462–476.

[6] Bonacina, M. P. On the reconstruction of proofs in distributed theorem proving: a modified Clause-Diffusion method. *J. of Symbolic Computation 21*, 4–6 (1996), 507–522.

[7] Bonacina, M. P. The Clause-Diffusion theorem prover Peers-mcd. In *Proc. of the 14th CADE* (1997), W. McCune, Ed., vol. to appear of *LNAI*, Springer Verlag. (4 pages).

[8] Bonacina, M. P., and Hsiang, J. Parallelization of deduction strategies: an analytical study. *J. of Automated Reasoning 13* (1994), 1–33.

[9] Bonacina, M. P., and Hsiang, J. The Clause-Diffusion methodology for distributed deduction. *Fundamenta Informaticae 24* (1995), 177–207.

[10] Bonacina, M. P., and Hsiang, J. A category-theoretic treatment of automated theorem proving. *Journal of Information Science and Engineering 12* (1996), 101–125.

[11] Bonacina, M. P., and Hsiang, J. On the representation of dynamic search spaces in theorem proving. In *Proc. of the Int. Computer Symp.* (1996), C.-S. Yang, Ed., pp. 85–94.

[12] Bonacina, M. P., and McCune, W. Distributed theorem proving by Peers. In *Proc. of the 12th CADE* (1994), A. Bundy, Ed., vol. 814 of *LNAI*, Springer Verlag, pp. 841–845.

[13] Bündgen, R., Göbel, M., and Küchlin, W. Parallel ReDuX → PaReDuX. In *Proc. of the 6th RTA* (1995), J. Hsiang, Ed., vol. 914 of *LNCS*, Springer Verlag, pp. 408–413.

[14] Bündgen, R., Göbel, M., and Küchlin, W. A master-slave approach to parallel term-rewriting on a hierarchical multiprocessor. In *Proc. of the 4th DISCO* (1996), J. Calmet and C. Limongelli, Eds., vol. 1128 of *LNCS*, Springer Verlag, pp. 184–194.

[15] Bündgen, R., Göbel, M., and Küchlin, W. Strategy-compliant multi-threaded term completion. *J. of Symbolic Computation 21*, 4–6 (1996), 475–506.

[16] Butler, R., and E.Lusk. User's guide to the p4 programming system. Tech. Rep. 92/17, MCS Division, Argonne Nat. Lab., 1992.

[17] Denzinger, J., and Schulz, S. Recording and analyzing knowledge-based distributed deduction processes. *J. of Symbolic Computation 21*, 4–6 (1996), 523–541.

[18] Dershowitz, N. Termination of rewriting. *J. of Symbolic Computation 3*, 1 & 2 (1987), 69–116.

[19] Dershowitz, N., and Jouannaud, J.-P. Notations for rewriting. Tech. Rep. 478, LRI, Université de Paris Sud, 1990.

[20] Gropp, W., and Lusk, E. User's guide for mpich, a portable implementation of MPI. Tech. Rep. 96/6, MCS Division, Argonne Nat. Lab., 1996.

[21] Hsiang, J., and Rusinowitch, M. On word problems in equational theories. In *Proc. of the 14th ICALP* (1987), T. Ottman, Ed., vol. 267 of *LNCS*, Springer Verlag, pp. 54–71.

[22] Huet, G. An algorithm to generate the basis of solutions to homogeneous linear Diophantine equations. *Information Processing Letters 7* (1978), 144–147.

[23] Huntington, E. V. Boolean algebra: A correction. *Trans. of the AMS 35* (1933), 557–558.

[24] Huntington, E. V. New sets of independent postulates for the algebra of logic. *Trans. of the AMS 35* (1933), 274–304.

[25] Kapur, D., Musser, D., and Narendran, P. Only prime superposition need be considered in the Knuth-Bendix completion procedure. *J. of Symbolic Computation 6* (1988), 19–36.

[26] Kirchner, C., Lynch, C., and Scharff, C. Fine-grained concurrent completion. In *Proc. of the 7th RTA* (1996), H. Ganzinger, Ed., vol. 1103 of *LNCS*, Springer Verlag, pp. 3–17.

[27] Lankford, D., and Ballantyne, A. The refutation completeness of blocked permutative narrowing and resolution. In *Proc. of the 4th Workshop on Automated Deduction* (1979), W. H. Joyner, Ed., pp. 168–174.

[28] McCune, W. Experiments with discrimination tree indexing and path indexing for term retrieval. *J. of Automated Reasoning 9*, 2 (1992), 147–167.

[29] McCune, W. Otter 3.0 reference manual and guide. Tech. Rep. 94/6, MCS Division, Argonne Nat. Lab., 1994.

[30] McCune, W. 33 Basic test problems: a practical evaluation of some paramodulation strategies. MCS Division, Argonne Nat. Lab., Pre-print P618, 1996.

[31] McCune, W. Solution of the Robbins problem. Submitted manuscript, 1996.

[32] Niewenhuis, R., and Rubio, A. Basic superposition is complete. In *Proc. of the European Symp. on Programming* (1992), B. Krieg-Brückner, Ed., vol. 582 of *LNCS*, Springer Verlag, pp. 371–389.

[33] Plaisted, D. A. The search efficiency of theorem proving strategies. In *Proc. of the 12th CADE* (1994), A. Bundy, Ed., vol. 814 of *LNAI*, Springer Verlag, pp. 57–71. Also Tech. Rep., Max Planck Institut für Informatik, MPI-I-94-233.

[34] Robinson, G., and Wos, L. Paramodulation and theorem-proving in first-order theories with equality. In *Machine Intelligence*, D. Michie and R. Meltzer, Eds., vol. IV. Edinburgh University Press, 1969, pp. 135–150.

[35] Slagle, J. R. Automated theorem proving for theories with simplifiers, commutativity, and associativity. *J. of the ACM 21* (1974), 622–642.

[36] Stickel, M. E. A unification algorithm for associative commutative functions. *J. of the ACM 28*, 3 (1981), 423–434.

[37] Stickel, M. E. A case study of theorem proving by the Knuth-Bendix method: discovering that $x^3 = x$ implies ring commutativity. In *Proc. of the 7th CADE* (1984), R. E. Shostak, Ed., vol. 170 of *LNCS*, Springer Verlag, pp. 248–258.

[38] Suttner, C. B., and Schumann, J. Parallel automated theorem proving. In *Parallel Processing for Artificial Intelligence*, L. Kanal, V. Kumar, H. Kitano, and C. B. Suttner, Eds. Elsevier, Amsterdam, 1994.

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

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

[41] Wos, L. Searching for open questions. *Newsletter of the Association for Automated Reasoning 15* (May 1990).