TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Cooperative Parallel Automated Theorem Proving

Andreas Wolf Marc Fuchs

TUM-I9732
SFB-Bericht Nr. 342/21/97 A
Juni 97

TUM–INFO–06-I9732-200/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Cooperative Parallel Automated Theorem Proving

Andreas Wolf, Marc Fuchs
Computer Science Department
Munich University of Technology
D-80290 Munich
Germany
e-mail: {wolfa,fuchsm}@informatik.tu-muenchen.de

June 20, 1997

Abstract

Automated Theorem Proving can be interpreted as the solution of search problems which comprise huge search spaces. Parallelization of the proof task as well as cooperation between the involved provers offer the possibility to develop more efficient search procedures. In this paper we want to investigate concepts for the development of cooperative parallel theorem provers. We deal with architectural questions as well as with possibilities of how to realize cooperation. Particularly, we discuss requirements on an efficient load distribution mechanism for cooperative parallel theorem provers. As a result of this discussion we develop the model of the new prover CPTHEO. This prover allows for a dynamic load distribution that fulfills the requirements introduced before. Furthermore, the cooperation possibilities of CPTHEO offer the potential for reaching superlinear speed-ups.

1 Introduction

Up to now, the sequential Automated Theorem Proving (ATP) has set a very powerful standard. But when dealing with more difficult problems ATP systems are still inferior to a skilled human mathematician. Thus, methods to increase the performance of existing ATP systems are, besides the development of new proof calculi, a focus of interest in the ATP area. One important technique to increase performance is to employ parallelism on parallel hardware or networks of workstations. Possible parallelization concepts vary from the parallel use of different configurations of the employed provers, to the partitioning of the proof task into subtasks that are tackled in parallel. Moreover, one can expect further improvements by interactions between the different parallel provers (cooperation), e.g. by exchanging intermediate results.

In this article we want to analyze the requirements for automatic load distribution for parallel theorem proving. The article is a contribution to the discussion on the possibilities of cooperation of ATP's, especially concerning questions of load distribution. A generic approach to cooperative concepts is presented. The costs needed for implementation and runtime, especially for communication, are estimated. Furthermore, an existing parallel prover, *SPTHEO*, and a new cooperative prover based on *SPTHEO*, *CPTHEO*, are introduced.

Developers of theorem provers are often more logicians than experts on parallel hardware. Therefore, function libraries are needed for the communication between two or more processes. Tools to achieve an optimally balanced load distribution should be as transparent as possible.

We assume the existence of a widely used interface for parallel applications, e.g. the *Parallel Virtual Machine (PVM)* [GBD$^+$94]. It makes the access to network or parallel hardware fully transparent to the user. Unfortunately, broadcasting messages to a group of processes in *PVM* is simulated only by single sequential transmissions resulting in a decrease in performance. Another as yet non-implemented function is the possibility to estimate the expected performance of the involved processors to spawn new processes to convenient machines. The following sections will consider which aspects have to be dealt within this context by an automated load distribution tool. Making migration of the processes to other processors possible can use the potential of the processors better than the performance snapshots before the spawn. A better load distribution can thus be achieved. As we shall explain later, there is, in general, an unsolvable difficulty in estimating which resources are needed for a particular prover in the context of a parallel proof system.

In the following, a short introduction to parallel theorem proving is given. Section 2 contains a classification of aspects to be considered when constructing cooperative provers. Section 3 describes the *SPTHEO* prover and introduces the model of a new cooperative prover *CPTHEO* which is based on *SPTHEO*. We conclude with some conclusions and an outlook at possible future work.

1.1 What is Automated Theorem Proving (ATP)?

1.1.1 General Remarks.

Theorem proving deals with the search of proofs of certain conjectures from a given theory. Both conjecture and theory are formulated in some language $\mathcal{L}$. An frequently used language is e.g. the First Order Predicate Logic (PL1). PL1 can be used to formulate many mathematical problems as well as problems taken from the real life. Unfortunately, PL1 is undecidable in general (Church, 1936). One can show that the set of theorems of a given theory is recursive enumerable. Specifically, there exist proof procedures able to recognize each valid formula of a given theory after a finite amount of time. Thus, at least semi-decision algorithms can be constructed although the problem of deciding the validity of a formula is, in general, undecidable.

In the past, a lot of different logic calculi have been developed and implemented. Important properties of such calculi are *soundness* and *completeness*. Soundness of a calculus means that it does not give incorrect answers and announce a formula as valid although it is not. Completeness means that the calculus has the potential to prove every valid formula.

Basically, proof calculi (or automated theorem proving systems based on such calculi) can be divided into two different classes. On the one hand there are synthetic calculi that work in a bottom-up manner, on the other hand one can employ analytic calculi that work top-down. Analytic calculi attempt to recursively break down and transform a goal into sub-goals that can finally be proven immediately with the axioms (*goal-oriented provers*). Synthetic calculi go the other way by continuously producing logic consequences of the given theory until a fact describing the goal is deduced (*saturating provers*). For further information on these issue we refer to [Bib82].

Note that the problem of proving the validity of a given conjecture by using a certain calculus can be interpreted as a search problem. A proof calculus is a *search calculus* that is represented by a transition system consisting of a set of allowed states, a set of rules for changing states, and a termination test for identifying final states. Basically, search calculi are divided into two classes: *Irrevocable* search calculi and *tentative* search calculi. Using irrevocable calculi, an application of a search step never needs to be undone in order to reach a final state. Tentative search calculi require provision for the case that a sequence of search steps does not reach a final state. Usually *backtracking* is needed in order to try a solution with each alternative search step that can be applied within a search state.

Independent from the search calculus to be applied, the general problem of ATP is that the search spaces one has to deal with when searching for a given conjecture are tremendous. Thus, the development of more efficient methods to reduce and examine the search space is one of the aims of the automated theorem proving community.

1.1.2 Model Elimination.

As a concrete proof procedure we want to mention the *Model Elimination* calculus (*ME*). This calculus can be interpreted as an analytic calculus. Furthermore, backtracking is usually needed in order to prove a theorem. In this paper we only want to recall some basic concepts. An introduction to *ME* can be found in [Lov78]. Within this paper, tableau style representation of *ME* proofs is used. *ME* deals with clauses, so we cope with a set of universally quantified clauses as OR-connected lists of literals.

A *clausal tableau* is a tree with nodes labeled with literals. To simplify the following, it is assumed that we want to prove the validity of a literal under a given set of clauses. Thus, the initial tableau consists of the negation of the literal to be proved. Of course, more complex formulae can be treated. The tableau can be *expanded* by appending instances of the literals of a clause to a leaf of the tableau tree as new leaves. A branch of the tree is *closed* if it contains complementary literals. A substitution on the whole tree may be required to make the literals complementary.

In *ME*, after an expansion of the tree, one of the newly created leaves must be closed against its immediate ancestor. Additional closing of other branches after an expansion is called *reduction*. Without loss of generality we can assume that one of the nodes closing a branch is a leaf. A tableau is a *proof* if all its branches are closed.

In order to prove a given literal it is necessary to systematically construct all possible tableaux. Since a lot of different expansion and reduction steps are usually applicable to any given tableau one has to search for a proof in a *search tree* whose nodes are marked with (different) tableaux. A node in this search tree that represents an open tableau is also called a *proof task*.

1.2 Developing Parallel Provers

1.2.1 Fundamentals.

There are powerful sequential automated theorem provers in the theorem prover community. As examples, we give here the (incomplete) list *DISCOUNT* [ADF95], *OTTER* [McC90] and *SETHEO* [LSBB92] which belong to the most popular automated theorem provers. But when dealing with “hard problems” these provers are inferior to a skilled human mathematician because of the tremendous search spaces the provers have to deal with.

The performance of sequential provers, however, can often be significantly increased by developing parallel versions of them. The following table classifies examples of existing parallel provers according to the criteria given below.

	non-cooperative	cooperative
partitioning completeness based	PARTHEO [LS90] METEOR [Ast92] SPTHEO [Sut95]	PARROT [JOK92] ROO [LMS91] DARES [CMM90]
partitioning soundness based	MGTP/G [FHKF92] SPTHEO [Sut95]	
competitive different calculi		HDPS [Sut92]
competitive unique calculus	RCTHEO [Phi92] SICOTHEO [Sch95]	DISCOUNT [ADF95]

Parallel theorem provers can be *cooperative* or *non-cooperative*, depending on their behavior in information exchange. *Cooperative* systems exchange information during a proof run and not only in the phases of their initialization and termination. The paper deals especially with the possibilities of designing provers of this kind (cf. the following sections).

Theorem provers can *partition* their work, for instance by splitting the whole task into a set of subtasks, or by partitioning the considered search space. Partitioning systems can be classified analogously to the usual classification of parallel algorithms into *OR and AND parallel* algorithms: *Completeness based* systems create completely independent subtasks when partitioning the search space, i.e. the solution of one single problem means the solution of the whole problem. *Soundness based* systems require the combination of the solutions of the subtasks to a common solution. These two strategies can be combined, so that sub-provers can deal with independent and dependent subproblems. Partitioning systems usually use instances of the *same inference machine*, because the partitioning of the search space is difficult for different logic calculi or even for different variants of the same calculus.

Competitive systems are proof systems where each system works on the whole proof task but differ in the way how the involved provers are parameterized. Even the use of different logics in parallel is possible. In constructing competitive systems, it seems to be useful to encourage *different calculi*; advantages of some of them can balance disadvantages of others. This strategy assumes that for each involved provers there exists a class of tasks which have a higher performance than all other provers included in the common system.

Developing parallel provers is sensible since the parallelization of a proof system yields the potential to achieve speedups compared with the original system, regardless of the method of parallelism that is applied. Partitioning the search space offers the possibility to reduce the amount of time spent exploring unnecessary parts of the search space. This is because of the chance that at least one of the involved provers immediately starts exploring an "interesting" part of the search space (assuming that the search space is completely assigned to the different provers). Competitive systems can result in super-linear speedups because one has a higher probability to use a search strategy that is well suited for the given problem. Since usually a lot of different configurations of a prover are imaginable and no a priori knowledge is available to decide which configuration is well suited for a specific problem it is reasonable to use different configurations in parallel.

The highest gains in efficiency, however, can be expected when using cooperative systems due to the expected synergetic effects. Thus, the development of cooperative systems is a main research area when dealing with parallel theorem provers.

It is known, that on relatively easy problems, parallel provers require a higher amount of time than a similar sequential prover due to the enlarged overhead the parallel system needs to launch the program. Furthermore, communication overhead occurs that additionally decreases the inference rate of the involved provers. But these disadvantages will be compensated when dealing with really hard problems if synergetic effects occur.

So it should be the aim of parallel provers to obtain profits in those domains where existing sequential or non-cooperative parallel systems do not find any proof, or where they do find one, only in a comparatively long time. During the development of the parallel version of *SETHEO* which deals with partitioning of the search space (*SPTHEO* [Sut95]), a comparison with the results of the sequential version was performed using a runtime of 1000 seconds. Problems of this complexity seem to be convenient examples to test the performance of parallel provers.

1.3 Cooperation – reasons and a definition

Sequential search procedures often employ a large set of heuristics and refinements of the underlying calculus to prevent unnecessary search, e.g. to avoid redundant search steps. If the search is shared among different processes, it may be useful to exchange information on such redundancies, as well as on important intermediate results. Basically, there are two methods to exchange data between theorem provers: *demand driven* (as realized in the *DARES* system) and *success driven* (as realized in *DISCOUNT*). Demand driven cooperation means that a prover is interested in certain information and asks the other provers for it. Success driven cooperation means that a prover communicates information it judges to be important to the other provers. As we can see in the table below, the unlimited exchange of information usually is not sensible due to the tremendous amount of information that is generated during the search.

The table shows two examples from the *TPTP* [SSY94]. It contains the number of smaller proof tasks to which the original task can be simplified after *level* inferences of a ME based prover. It also shows the number of possibly generated messages (when asking for solved tasks of another prover). Altogether, one observes the importance of restricting and filtering information to be exchanged.

example	level	original	sort|uniq (percent)	subsumption (percent)
BOO003-1	7	272	83	72
	8	1006	61	36
PRV007-1	5	145	84	79
	6	826	70	55
	7	5677	65	51

In this article we want to employ the following definition of cooperation:

Definition. An automated theorem prover is cooperative, if and only if it offers on request immediately all its data and results to other provers. Immediately means that the prover utilizes the available bandwidth of the communication channels and handles the request with a higher priority than its own proof attempt. $\diamond$

2 Cooperative Theorem Proving

We discuss a classification of the problems that are to be solved when constructing a cooperative parallel theorem prover with some remarks on communication and load distribution needs. The aspects to be considered can be divided into two parts.

- *How* can cooperating provers work together, i.e. which topics concerning the system architecture have to be considered?
- *What* can they do to work together, i.e. which kinds of cooperation can occur?

2.1 System Architecture

The problems to be considered in relation to the first question, i.e. the system architecture, can be classified as follows.

- Are the involved inference machines of the same type? Are different types of inference machines used, i.e. is the parallel system *homogeneous* or *heterogeneous*?
- Is the exchange of information planned in such a way that messages must be confirmed by the receiver or not? Shall the processes wait for some events or not? These questions lead to the decision on a *synchronous* or *asynchronous* mode of information exchange.
- Which scheme of process control and control on the progress of the proof shall be selected, i.e. which *hierarchical structures* of sub-provers occur?
- Proof procedures can be saturating (deduce all valid formulae until the proof goal occurs) or goal oriented (reduce the proof task until all subtasks can be solved using known - given - formulae). It is possible to construct a system with only *goal oriented*, only *saturating*, or with *hybrid* proof processes.

2.1.1 Homogeneous and Heterogeneous Systems.

Parallel provers developed up to now ordinarily use (with minor changes) the implementation of the inference machine of a sequential prover. These sub-provers are united by partitioning the search tree in a completeness or soundness based manner as previously described, or they can use different search strategies in their different instances.

Due to the use of the same inference machine and an equal coding of formulae and control parameters, the expected additional costs for the implementation of an information exchange in such *homogeneous systems* are relatively low. Moreover, we can assume that heuristics estimating the needed resources for a proof task (a process, in terms of load distribution) can be found much easier than in the heterogeneous case.

Homogeneous systems do not necessarily require the same search procedure for all involved provers. For example, the connection of *SETHEO* [LSBB92] with the *DELTA Iterator* [Sch94], which are based on the same inference machine, combines the *top-down* search with the *bottom-up* one. The top-down prover can integrate lemmata of the bottom-up prover in its proof tree. That kind of co-operation was tested successfully in the sequential case. Homogeneous systems allow an easy way to partition the search space of the involved systems by control parameters for the provers and startup configurations containing different pre-calculated proof segments.

So we can classify homogeneous systems into

- systems using the *same* inference machine with the *same* search procedure and
- systems using the *same* inference machine with *different* search procedures.

Because of the necessary syntactical transformations and different semantics *heterogeneous systems* probably need more effort during implementation. In the interactive proof system *ILF* [DGH$^+$94] the main part of implementation required for communication is syntactical transformation and adaption of theories. Considering the runtime information exchange, the context needed for different systems will significantly increase the amount of information to be exchanged. Different instances of the same inference system canonically interpret received formulae in the right way. Different systems obviously need additional information on the used calculus and the coding of proof structures within the program.

Sometimes a problem can be separated into two relatively independent subproblems which belong to different problem classes. In such cases, it makes sense to give these subproblems to different provers adapted to the problem considered. This is not necessarily an argument for heterogeneous structures, but absolutely an argument for the use of special provers as subsystems.

Heterogeneous systems are justified only if the connection of the systems leads to a significant increase of the performance of the common system. That can be done, for instance, by integrating a prover for equational problems such as *DISCOUNT* [ADF95] into a system that is poor on equations. Another possibility is the integration of provers applying meta-mathematical knowledge as for instance *TreeLat* [DGH$^+$94], a special prover and model checker for lattice ordered groups.

A similar approach is the integration of model checking, e.g. using *FINDER* [Sla95], as a semantic tool into a common proof system which will reject many intermediate proof tasks as false. If model checking is not integrated into the inference machine itself, the model checker has to be invoked such that the communication connection to the prover can reach a high throughput of data.

Heterogeneous systems can be classified as

- systems using *different* inference machines with the *same* communication language,
- systems using *different* inference machines with *different* communication languages,
- systems using not only inference machines but also *meta-methods* for the proof.

2.1.2 Synchronous and Asynchronous Exchange of Information.

Cooperation between provers needs exchange of information at runtime, not only during a short phase of initialization and termination of one or more of the processes. Communication will take place during the whole time the proof system works. In principle, the information exchange can be done according to one of two general models: information can be exchanged

- *synchronously* or
- *asynchronously.*

Using the *synchronous* mode, all partners are informed on the situation of all other partners. Generally, a message will be confirmed. That means, at least one of the exchanging partners is waiting until the others are ready to receive or to send a message. The synchronous mode of information exchange has the advantage that all involved processes mutually know their standard of information, i.e. if a message was written, it is read at the same predetermined time in the program scheme.

When communicating *asynchronously*, the sender cannot assume that the receiving prover got the message at the predetermined moment in the program cycle. But assuming the *message passing concept*, e.g. *PVM* [GBD+94], it is guaranteed that, at least the temporal sequence of messages between each two of the involved processes will be preserved. Furthermore, it is guaranteed that all messages reach the receiver, if this process still exists. This method has the advantage that no time is wasted by waiting for a communication partner. It has the disadvantage that usually information is not available at the moment it is needed. I. e., important information (for instance a lemma) may not be available in time. The following cases are imaginable.

1. The information read is already available in the prover. That means that this information is redundant (at that moment). Thus, the effort to find that information is done twice.
2. It may happen that the lemma read is more general than a subgoal proved internally in the prover. Then, it is possible that later subgoals can be solved using the lemma but not with the internal subproof. In that case, the internal subproof should be replaced by an application of the lemma.
3. The lemma may be in contradiction to assumptions of the internal proof. Then the subproofs which employ such an assumption must be corrected.

It would be ideal if the lemma solves an actual subgoal or needs only a few inferences to solve this goal.

Asynchronous event-oriented control can be implemented using the *non-blocking* read routines of the *PVM*. Another way is to use signals of the UNIX operating system.

2.1.3 Hierarchical structures of sub-provers.

The structure of the processes belonging to the parallel prover can be selected according to the following models.

- All sub-provers are *on the same hierarchical level*. Because the processes do not need to take their place in a hierarchical order during the initial phase, the start of the whole proof system is very easy in this model. Thus, no communication is needed for that purpose. But using this model, it is not possible exactly to connect two particular processes. Either broadcasting has to be performed, resulting in high costs or the processes must obtain knowledge about the way they can communicate, for instance using process tables.

- The sub-provers can be arranged in a *hierarchical structure*. If a prover generates new subtasks, it will create the subordinated provers and transmit the tasks to them. The information exchange can be done easily in this model but the controlling of the globally used resources is complex. The uniform distribution of the generated processes according to the processor load on the involved machines would have to be realized using *PVM* or additional software.

- The *combination* of different structures following both models is also possible. Nevertheless, each architecture needs facilities for information exchange, i.e. it should be possible to group processes and perform broadcasts to such groups. Information on the structure of the whole prover, e.g. its hierarchical organization and the information exchange relations might be a hint for effective load balancing too, and so it should be accessible to a load distribution system.

2.1.4 Combination of Goal Oriented and Saturating Provers.

It is the aim of using parallelization of theorem provers to decompose the search space for finding a proof, and to treat the parts of that search space at the same time in parallel. Using cooperative concepts it is possible to exchange information about solved subgoals to prevent redundancies, specifically solving a subgoal more than once. Furthermore, it is possible to partition the search space in a "horizontal" manner, i.e. to combine *bottom-up* with *top-down* proof procedures.

This combination already has been successfully tested in a version of the theorem prover *SICOTHEO* [Sch95]. In that system the *DELTA iterator* [Sch94] and *SETHEO* [LSBB92] worked together sequentially in the sub-provers. The depth of the found proof (i.e. depth of the corresponding closed tableau) usually is much lower compared to a single run of *SETHEO*.

In an application of the interactive *ILF system* [DGH+94], a combination of *DELTA* and *SETHEO* was used (only one process of each kind, both working sequentially). In the domain of lattice ordered groups (with a CPU time resource from 30 to 120 seconds) it was possible to increase the average depth of the proofs including the (later) expanded lemmata from 5 to 8, compared with the standard version of *SETHEO*.

Another existing application which works on equational problems is the combination of saturating and goal oriented *experts* in *DISCOUNT* [DF94].

Using a sequential prover, combining top-down and bottom-up means that at first some bottom-up steps are performed followed by top-down calculations. So in parallel, a high priority should be assigned to the bottom-up processes in the beginning, and lowerded during the work.

2.2 Kinds of Cooperation

In the above, the discussion was about aspects of "how" provers can cooperate and did not consider specific inference machines used in the parallel prover. In the following, we will look at the sub-provers, and we discuss what the involved inference machines can do together. A classification of kinds of cooperation is given as follows:

- the exchange and optimization of configurations and control information,
- the exchange of intermediate results (lemmata), and
- the exchange of failure information.

2.2.1 Different Strategies of Search.

In most cases, the provers involved in a parallel proof system partition the search space. For example *SPTHEO* [Sut95] expands the search tree up to a certain depth, and the resulting tableaux are given to the involved single provers. These provers "replay" their initial tableau and start their search on that base. The further search of the sub-provers was done using the same strategy for all provers. This technique yields very good results [Sut95].

Unfortunately, it is possible that the same search strategy produces similar subgoals at the same time on similar proof tasks. That means loss of chance that intermediate results solved by one prover could be interesting for another one: one prover having already solved that subgoal itself cannot use the external results.

If different search strategies are used by the sub-provers (different kinds of *iterative deepening*, most possible *length* and *linearity* of inference chains, search for goals of a certain *structure*, etc.) it is possible that intermediate results generated by one prover can be re-used by one or more of the others for their further work. Therefore, it can be accepted that some of these provers work with *incomplete search strategies*, if they solve tasks from their specific domain especially fast and efficiently. But it should be realized that the whole system remains *fair* and *complete*. So sub-provers with specialized strategies can be used, for instance, to generate lemmata or to deal with subgoals belonging to special problem classes such as equality problems.

The cooperation of parallel provers can be classified with respect to the used search strategies as follows:

- provers with the *same* search strategy,
- provers with *different* search strategies, *each* fair and complete,
- provers with *different* search strategies where only the *ensemble* is complete, and,
- provers with *different*, even in the ensemble *incomplete* strategies.

2.2.2 Cooperation and Competition.

The relation between cooperation and competition has been discussed in [FK87] and [Sut91]. Often, competition is the basic concept of existing parallel provers. The reasons are the low cost of implementation and the small amount of interprocess communication.

Cooperation and Competition are not necessarily contradictory. If competitive provers exchange information, they lose time for their own work, but using the results of their competitors they may solve their tasks faster. An example for such a synergetic effect is the *Teamwork Method* [Den95]. This concept includes the competition of some provers which exchange their intermediate results periodically.

Cooperation *without* competition inheres the risk that sub-provers with bad results on a special class of tasks can decrease the performance of the whole system, if their useless results increases the amount of information to be processed. Competitive concepts can eliminate such provers from the actual configuration of the system.

2.2.3 Exchange and Optimization of Configurations and Control Information.

Speaking about cooperation in the context of parallel theorem provers, one at first thinks about exchange of formulae or sets of formulae. But, a further possibility is the exchange of all information that describes *how* an inference machine works such as

- control parameters,
- used heuristics,
- search strategies,
- inference rates,
- and so on.

It is useful to terminate the sub-provers that were less successful in the last time period and to start new provers with the control information from the more successful ones. It is also imaginable that the existing provers with lower success get the parameters to change their behavior in the intended sense.

Based on this idea a system could be implemented where a parallel prover optimizes a set of control parameters. During a certain interval of time some span of parameters will be tested and only the successful settings will be kept in the next interval where the span can become smaller or the kind of the parameter can be changed.

Furthermore it is to remark that in this model information is exchanged only infrequently. An information transfer only after some cycle of stand alone work is used less frequently than a continuous transfer during the proof process.

It has to be considered which criteria are suitable in determining the quality of the control configuration of a prover. A measure for that purpose can be, for instance, the depth of the proof structure relative to the other sub-provers or the number and quality of the generated lemmata. Using such criteria it should be considered that the resulting system must have a *fair* search strategy to save the completeness of the whole joined system.

Considering existing systems, the results of *DISCOUNT* [DK94] show that generating proof procedures improve searching by means of exchange. Analytical provers will probably cause more problems.

2.2.4 Exchange of Intermediate Results (Lemmata).

In the previous paragraph, we have described the exchange of configurations. Now, we discuss the concept one thinks about at first, namely the exchange of proved intermediate results (proved formulae). Considering their structure, the formulae to be integrated will mostly be *literals*. But it is also possible that *more complex formulae* have to be transmitted.

Considering how these lemmata are exchanged it should be possible to integrate new axioms into the knowledge base during the inference process. It should be easily done using literals (facts) but the possibility of integrating more complex formulae, especially clauses, would be desirable.

If that is not possible, the old situation of the inference machine has to be conserved and must be reconstructed after the integration of the new formulae into the knowledge base. A possible technology to do so is the *description of start configurations* distributing the tasks, as it is used in *SPTHEO* [Sut95]. That method probably has lower costs of implementation than the variant of integration into the running prover, but it has other disadvantages, for instance, the transfer of context can be useless for the generated subgoal or even harmful in the new context. Altogether we get

- provers integrating lemmata "on the fly" without restart, and
- provers restarting after lemma integration.

We can assume that each inference step can lead to a new lemma. So during the proof process a large number of lemmata can be generated. This will be the main conceptual problem on the way towards a cooperative parallel prover. A transmission of all possible lemmata without filtering, even to a subset of the involved sub-provers, will cause an overloading of the network. Thus, the candidate lemmata must be evaluated to decide if they can be exchanged. In order to get a *measure of a lemma* one can use information on

- the syntactical structure of the lemmata (e.g. the high generality), and
- information on the derivation of the lemmata, e.g. the number of inferences.

If provers can ask for goals that are important for their work, then the existence of such a question for a formula should be a criterion for a lemma. These questions should be filtered analogously to the lemmata to avoid network overload. The concept of proof requests is implemented in *DARES* [CMM90] where information is exchanged, if a formula has already been proved by another sub-prover.

2.2.5 Exchange of Failure Information.

In the sections before, we discussed the exchange of information about successful events (successful configurations, lemmata). These informations can be considered to be *positive* knowledge. Furthermore, *negative* knowledge can be exchanged, for instance, the provers can communicate about what they can*not* prove. Such messages should include the conditions under which a proof failed, i.e. the parameter settings and search bounds.

This concept also demands a strong selection, as it is already explained considering lemma and request generation. It must be realized that not every *backtracking* step generates negative information. It should be possible to use analogous criteria as considered in the case of lemma generation.

3 SPTHEO and CPTHEO - two applications

As a basic approach we consider the sequential Model Elimination [Sti84] style theorem prover *SETHEO* which was developed at the Technische Universität München. It is the basic inference machine involved in the prover systems described in this section. Problems in constructing parallel provers with other inference machines should be similar to our consideration, at least with respect to the generic needs of communication and load distribution.

3.1 The parallel theorem prover SPTHEO

Static Partitioning with Slackness (SPS) [Sut95] is a method for parallelizing search-based systems. Traditional partitioning approaches for parallel search rely on a continuous distribution of search alternatives among processors ("dynamic partitioning"). The SPS-model instead proposes to start with a sequential search phase, in which tasks for parallel processing are generated. These tasks are then distributed and executed in parallel. No partitioning occurs during the parallel execution phase. The potentially arising load imbalance can be controlled by an excess number of tasks (slackness) as well as appropriate task generation. The SPS-model has several advantages over dynamic partitioning schemes. The most important advantage is that the amount of communication is strictly bounded and minimal. This results in the smallest possible dependence on communication latency, and makes efficient execution even on large workstation networks feasible. Furthermore, the availability of all tasks prior to their distribution allows optimization of the task set which is not possible otherwise.

SPTHEO is a parallelization of the *SETHEO* system, based on the Static Partitioning with Slackness (SPS) model for parallelization. It consists of three phases.

- In a first phase, an initial area of the search space is explored and tasks are generated. The number of generated tasks exceeds the number of processors by a certain factor (slackness).
- In a second phase, the tasks are distributed.
- Finally, in a third phase the tasks are executed at the individual processors.

In this model the search space is initially developed sequentially until a sufficient number of alternative sub-search spaces (tasks) have been generated. These tasks are distributed to multiple processors that search the alternatives in parallel until one finds a proof. The number of tasks generated typically exceeds the number of processors, and the extent of this is the "slackness" in the system. For reasons of practical search completeness, each processor executes all its tasks concurrently (preemptive execution). *SPTHEO* is implemented in C and PVM, and runs on a network of 110 HP workstations. Extensive evaluations showed significant performance improvements over *SETHEO* and a previous parallelization of it.

Figure 1 displays the number of problems from the TPTP [SSY94] solved by *SETHEO* within runtime limits ranging from 0 to 1000 seconds ($\approx$ 17 minutes) and results of the prover *SPTHEO*.

As an example, given a runtime limit of 1000 seconds, *SETHEO* solves 858 problems. It can be expected that increasing the runtime limit only leads to a small increase in the number of additionally solved problems.

An examination of the proof-finding performance of *SPTHEO* for different runtime limits is also given in figure 1. It shows the number of problems solved by *SPTHEO* for 256 processors. The plot is for 256 generated sub-tasks and 1 to 256 processors. Each curve shows the performance for twice as many processors as for the lower one. The asymptotic upper bound on the number of proved problems is due to the runtime limit of 20 seconds per task.

3.2 A model of the cooperative prover CPTHEO

In this section the most significant components of the *CPTHEO* model are explained:

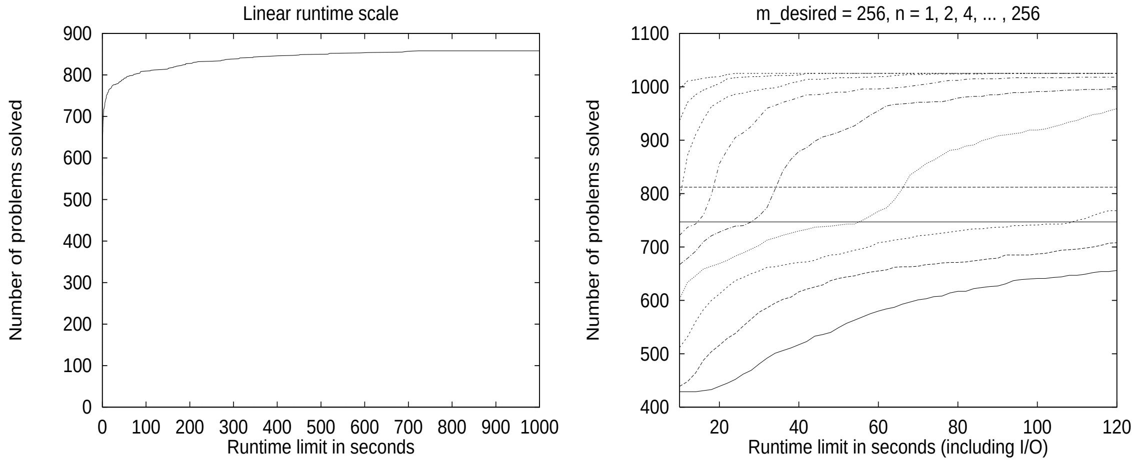


Figure 1: Left: The number of problems solved by SETHEO as a function of the runtime limit. Right: The performance of SPTHEO for overall runtime limits between 10 and 120 seconds (20 seconds per task). Each curve denotes a particular number of processors, starting with one processor for the lowest curve and representing twice as many processors with the next higher curve. The horizontal lines show the SETHEO performance for runtime limits of 10/120 seconds.

- the cascading task generation of the top-down inference machine,
- the unit preferring lemma mechanism,
- the redundancy filtering methods, and
- the relevance measuring mechanism.

To conclude, a scheme of the prover is given.

The cooperative prover *CPTHEO* to be developed uses techniques of *SPTHEO* for the scheduling of tasks and the success control. In addition to *SPTHEO*, the partial evaluation of the search space to determine the tasks of the sub-provers will be done iteratively. After each iteration (and after filtering) the generated tasks will be labeled with a number. That number measures the expected importance for the further proof process. The label mechanism is a tool to control the directions of the proof trial, and it can be used for effective load balancing. Due to the iterative development of the set of proof tasks, an adaption of the whole proof attempt to the real hardware configuration is possible even during the proof process. Furthermore, by changing the labels of the proof tasks it is possible to influence the heuristic of the search at runtime.

The information exchanged between the involved sub-provers consists in its main part of unit lemmata and unit proof requests (failed proof attempts of top-down provers). Units are preferred because of their lower complexity as formulae and their higher expected re-usability for other sub-provers. Furthermore, the *SETHEO* inference machine is constructed to read at most units at runtime. Experiments show that the unit preferring heuristic is only a weak restriction, because it is quite probable that units occur.

The filters of the sub-task generators and the lemma and proof request generators consider the following:

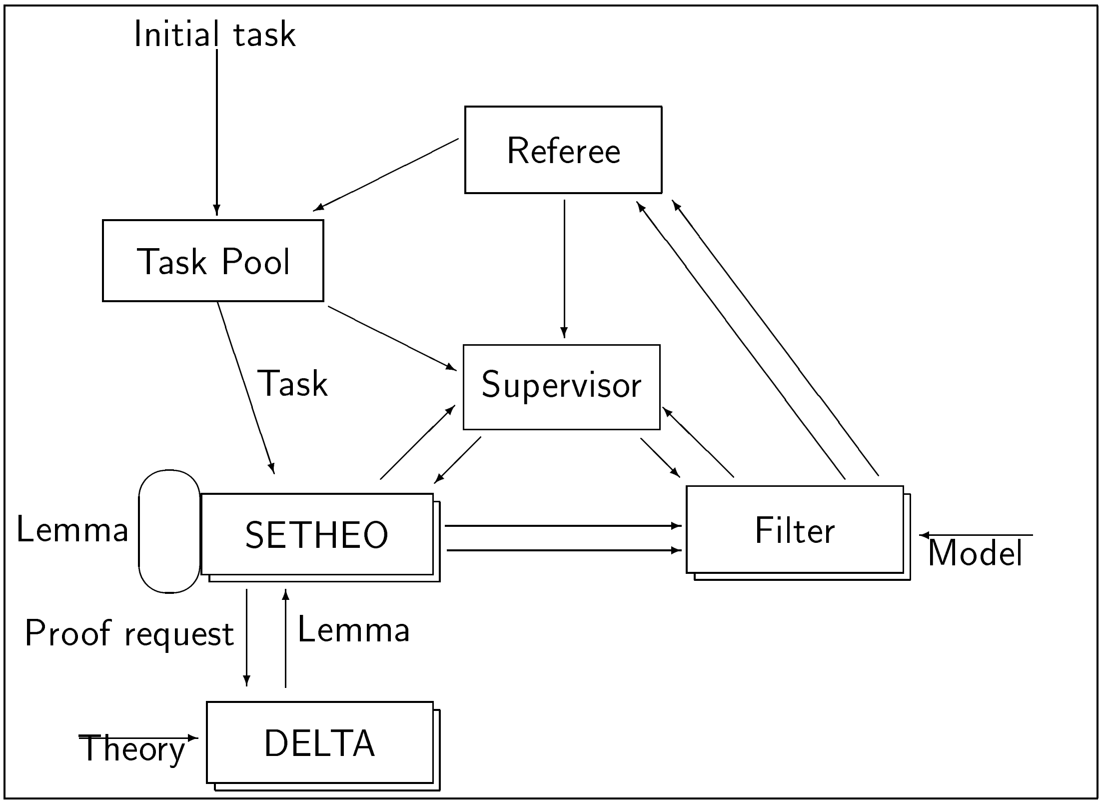
- identical formulae, subsumed formulae,
- tests with models of the considered theory,
- variants (very powerful in experiments),
- models given by the human operator,
- model fragments (if only infinite models exist).

Filters only delete multiply occurring or obsolete formulae. Referees rank the sub-tasks of a proof as well as lemmata or proof requests by labeling them with measures. Using these measures the proof process is controlled. The measures depend on:

- the generality of a literal,
- the derivation cost of a literal (the number of inferences to deduce the literal),
- the relatively isolated position of a subproof leading to a literal that is for instance only a few or no reductions to literals outside that subproof considering *SETHEO* proofs,
- the multiple usability of a literal in the considered proof or in the already existing parts of the proof, and
- the similarity of the generated facts to the task to be proved.

The scheduling has to guarantee the fairness of the whole proof attempt, and so the completeness of the proof procedure.

Now we describe the components. *SETHEO* provers reduce tasks from the task pool and generate new ones as well as proof requests. The *DELTA* iterators generate lemmata. The filters delete redundant tasks and lemmata from the task pool. The referees rank proof tasks, proof requests and lemmata due to their expected importance. All these processes are controlled by a central supervisor that starts and finishes processes and guarantees the liveliness of the whole prover system. To avoid bottlenecks in the information transmission, the task pool is kept locally by the involved processors. Only a certain amount of tasks labeled with high ranking measures is sent to a central task pool. The rest can be sent on request.



4 Conclusions

The increased power of automated theorem provers means that they will solve more than only toy examples in the near future. Today ATP systems support the human for example in interactive proof environments as *ILF* [DGH$^+$94]. Within *ILF* provers fill in steps of the proof sketch the human writes down. The more the performance of the automated provers increases, the larger the gaps between the given proof steps can be. One possibility of increasing the performance is the parallelization of provers. Cooperative systems have a need for fast communication between the involved processes and for a transparent load balancing due to the unpredictable progress of proof attempts.

It is known that on relatively easy problems parallel provers have a higher amount of time than a similar sequential prover due to the enlarged overhead the parallel system needs to launch the program and initial communications. So it can be expected that parallel provers, dealing with more communications at run time in addition, have once more lower rating than the corresponding sequential systems, if they have to deal with relatively easy problems. But this disadvantage will be compensated dealing with really hard problems, if synergy takes effect.

So it should be the aim of the development of a new cooperative parallel prover to obtain profits in those domains where existing sequential or non-cooperative parallel systems do not find any proof, or in the cases where they do find one, improving on the amount of time taken to do so.

Up to now, *CPTHEO* exists as a model as described in this article. Prototypical implementations of components of the prover showed the potential of the cooperative concept. An implementation of the whole system is planned for the near future.

References

[ADF95] J. Avenhaus, J. Denzinger, and Matth. Fuchs. Discount: A system for distributed equational deduction. In *Proceedings of 6. RTA*. Springer, 1995.

[Ast92] O. L. Astrachan. *Investigations in Model Elimination based Theorem Proving.* PhD thesis. Duke University, USA, 1992.

[Bib82] W. Bibel. *Automated Theorem Proving.* Vieweg, 1982.

[CMM90] S. E. Conry, D. J. MacIntosh, and R. A. Meyer. Dares: A distributed automated reasoning system. In *Proceedings of AAAI-90*, 1990.

[Den95] J. Denzinger. Knowledge-based distributed search using teamwork. In *Proceedings ICMAS-95*, pages 81–88. AAAI-Press, 1995.

[DF94] J. Denzinger and Matth. Fuchs. Goal oriented equational theorem proving. In *Proceedings of KI-94*. Springer, 1994.

[DGH+94] B. I. Dahn, J. Gehne, Th. Honigmann, L. Walther, and A. Wolf. *Integrating Logical Functions with ILF.* Preprint, Humboldt University Berlin, Department of Mathematics, 1994.

[DK94] J. Denzinger and M. Kronenburg. *Planning for Distributed Theorem Proving: The Team Work Approach.* SEKI-Report SR-94-09, University of Kaiserslautern, 1994.

[FHKF92] M. Fujita, R. Hasegawa, M. Koshimura, and H. Fujita. Model generation theorem provers on a parallel inference machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1992.

[FK87] B. Fronhöfer and F. Kurfess. *Cooperative Competition: A Modest Proposal Concerning the Use of Multi-Processor Systems for Automated Reasoning.* Technical Report, Department of Computer Science, Munich University of Technology, 1987.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[JOK92] A. Jindal, R. Overbeek, and W. C. Kabat. Exploitation of parallel processing for implementing high-performance deduction system. *Journal of Automated Reasoning*, (8), 1992.

[LMS91] E. L. Lusk, W. McCune, and J. K. Slaney. *ROO - A Parallel Theorem Prover.* Technical Report ANL/MCS-TM-149, Argonne Nat. Lab., 1991.

[Lov78] D. W. Loveland. *Automated Theorem Proving: a Logical Basis.* North-Holland, 1978.

[LS90] R. Letz and J. Schumann. Partheo: A High-Performance parallel Theorem Prover. In *Proceedings of CADE-10*. Springer, 1990.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High Performance Theorem Prover. *Journal of Automated Reasoning*, (8), 1992.

[McC90] W. McCune. Otter 2.0. In *Proceedings of CADE-10*. Springer, 1990.

[Phi92] J. Philipps. *RCTHEO II, ein paralleler Theorembeweiser.* Technical Report, Department of Computer Science, Munich University of Technology, 1992.

[Sch94] J. Schumann. Delta - a bottom-up preprocessor for top-down theorem provers. system abstract. In *Proceedings of CADE-12*. Springer, 1994.

[Sch95] J. Schumann. *SiCoTHEO - Simple Competitive parallel Theorem Provers based on SETHEO.* Technical Report, Department of Computer Science, Munich University of Technology, 1995.

[Sla95] J. Slaney. *FINDER Finite Domain Enumerator Version 3.0 Notes and Guide.* Technical Report, Australian National University, 1995.

[SSY94] C. B. Suttner, G. Sutcliffe, and T. Yemenis. The tptp problem library. In *Proceedings of CADE-12*. Springer, 1994.

[Sti84] M. E. Stickel. A prolog technology theorem prover. *New generation computing*, (2), 1984.

[Sut91] C. B. Suttner. *Competition versus Cooperation*. Technical Report, Department of Computer Science, Munich University of Technology, 1991.

[Sut92] G. Sutcliffe. *A Heterogeneous Parallel Deduction System*. Technical Report, ICOT TM-1184, 1992.

[Sut95] C. B. Suttner. *Static Partitioning with Slackness*. PhD thesis, Department of Computer Science, Munich University of Technology. 1995.