SiCoTHEO — Simple Competitive parallel Theorem Provers based on SETHEO*

J. Schumann
Institut für Informatik,
Technische Universität München
email: `schumann@informatik.tu-muenchen.de`

Abstract

In this paper, we present SiCoTHEO, a zoo of parallel theorem provers for first order predicate logic. They are based on the sequential prover SETHEO. Parallelism is exploited by competition: on each processor, an identical copy of SETHEO tries to prove the entire formula. However, certain parameters which influence SETHEO's behavior are set differently for each processor. As soon as one processor finds a proof, the entire system is stopped. Three different versions of SiCoTHEO are presented in this paper. We have used competition on completeness mode (parallel iterative deepening, SiCoTHEO-PID) and completeness bounds (parameterized combination of completeness bounds, SiCoTHEO-CBC), and on the search mode (top-down combined with bottom-up, SiCoTHEO-DELTA). The experimental results were obtained with a prototypical implementation, running on a network of workstations. This parallel model is fault-tolerant and does not need communication during run-time (except start and stop message). We found that only little efficiency is gained for SiCoTHEO-PID, which reaches peak performance with only 4 processors. SiCoTHEO-CBC and SiCoTHEO-DELTA, however, showed significant speed-up, and improved performance up to the 50 processors used.

1 Introduction

Automated theorem provers, like many other AI tools must explore large search spaces. This leads to long (and often too long) run-times of such systems. One possibility to reduce run-times is the exploitation of parallelism.

*This work is supported by the Deutsche Forschungsgemeinschaft within the Sonderforschungsbereich 342, Subproject A5: PARIS (Parallelization in Inference Systems).

Automated theorem proving in general seems to be suitable for parallelization, as many implemented parallel theorem provers show (see [13] for an extensive survey).

Current trends lead away from special purpose, tightly coupled multiprocessor systems (often with shared memory). Much more interesting (and feasible) seem to be networks of workstations, connected by a local area network (e.g., Ethernet, ATM). Such a hardware configuration is readily available in many places. It features processing nodes with high processing power and (comparatively) large resources of local memory and disk space. The operating system (mostly UNIX) allows multi-tasking and multi-user operation (which are not necessarily available on a multiprocessor machine). The underlying communication principle is message passing. Common data (e.g., the formula) can be kept in file-systems which are shared between the processors (e.g., by NFS). However, the bandwidth of the connection between the workstations is comparatively low and the latency for each communication is rather high.

Models of parallelism which are ideally suited for such networks of workstations must therefore obey the following requirements: small (or ideally no) necessary communication between the processors and no dependency on short latencies and a high communication bandwidth.

A parallel model which fulfills these requirements is *competition*: each processor tries to solve the entire problem, using different methods or parameters. As soon as one processor found a solution, the entire system can be stopped. In a competitive system there is no need for communication except for the start and stop messages.

Competitive parallel models have been studied in various approaches (cf. [4] and [13] for competitive parallel theorem provers). A competitive parallel theorem prover based on the sequential prover SETHEO [8, 7] is RCTHEO (*Random Competition*) [3]. The (literal) selection function for each processor is determined by a pseudo-random number generator which is initialized with a different number on each processor. Therefore, when the system has been started, the search-space is processed in a different way on each processor. A detailed evaluation of the RCTHEO system is contained in [4].

In this paper, we will focus on models where competition is accomplished by a different setting of parameters of the proof algorithm for each processor. This is in contrast to RCTHEO which exploits parallelism by randomizing the proof algorithm. Also, we only discuss systems, which are based on *one* sequential prover (*homogeneous competition*). In an inhomogeneous system, different theorem provers (e.g., OTTER, METEOR, SETHEO, ...) could run compete for a proof.

This paper proceeds as follows: First, we will define competition between homogeneous processes and discuss important properties of competitive theorem provers, such as efficiency, scalability, soundness and completeness. We will also give the definitions of the mean values for speed-up, which have been obtained in the experiments. After a short introduction into the sequential theorem prover SETHEO which is the basis for SiCoTHEO, we will discuss which parameters are suitable for competition on parameter level. Then, we will sketch the basics of the prototypical implementation of all SiCoTHEO systems on a network of workstations. Finally, we describe in detail the different SiCoTHEO provers, present results of experiments and assess them. We will conclude with an outlook on

possible applications for SiCoTHEO and future work.

2 Parallelism by Competition

Given a sequential theorem proving algorithm $\mathcal{A}(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ where $\mathcal{P}_i$ are parameters which may influence the behaviour of the system and its search (e.g., literal selection function or completeness modes). Then, a *homogeneous competitive theorem prover* running on P processors is defined as follows: on each processor p ($1 \leq p \leq P$) a copy of the sequential algorithm $\mathcal{A}(\mathcal{P}_1^p, \ldots, \mathcal{P}_n^p)$ tries to prove the *entire* given formula. Some (or all) parameters $\mathcal{P}_i^p$ are set differently for each processor p. All processors start at the same time. As soon as one processor could find a solution, this solution is reported to the user and the other processors are stopped ("winner-takes-all strategy").

The efficiency of the resulting competitive system strongly depends on the influence the resp. parameter setting has on the run-time. The larger the difference of run-times, which are created by different values of the parameters $\mathcal{P}_i^p$, the better the speed-up. If the influence of the parameters on the search is only weak, all processes will have a run-time which is quite similar. Then, the efficiency will be very low (the speed-up is about 1).

Good scalability and efficiency can be obtained only, if there are enough different values for a parameter, and if no good estimation to set that parameter in an optimal way is known. Only then, a large number of processors can be employed reasonably. A good estimation for parameters will in general result in poor speed-up values for a competitive system. Then, a different model of parallelisation will be appropriate. For a detailed theoretical discussion see [4].

When we want to develop and evaluate competitive parallel theorem provers, we have to address the following issues:

Soundness: although our parallel theorem provers are based on the sound sequential prover SETHEO, care must be taken that we obtain correct proofs in any case[1].

Completeness: the entire system must be complete, i.e., if there exists a proof, at least one processor should eventually find it. Since our system is intended to run on a network of workstations (where processors or communication links may fail), the competitive system should be complete even in cases with a reduced number of processors (*fault-tolerance*). In a standard partitioning scheme, like OR-parallelism (e.g., PARTHEO [11]), such a failure would lead to an incomplete system.

Efficiency: a central question regarding the use of parallel hardware is: is the result worth the effort? I.e., can we obtain a reasonable speed-up s? Efficiency η is defined as $\eta = s/P$ where P is the number of processors.

[1] Consider the case where we run a SETHEO without occurs-check in parallel with other provers. This may easily lead to incorrect proofs.

Scalability: our parallel theorem prover should show a good scalability. Often, networks contain hundreds of workstations which can (and should) be used reasonably. Furthermore, the number of available processors may vary strongly from proof task to proof task (e.g., due to system load or the activity of other users). Therefore, the system should also be robust w.r.t. changes in the number of processors. This means that $\frac{\partial \eta}{\partial P}$ should be smooth.

The run-times given in this paper are those of the SETHEO Abstract Machine, including the time to load the compiled formula. All times are CPU-times and measured with a granularity of 1/60 seconds. As the sequential run-time T_{seq}, we always use SETHEO, running with default parameters[2]. The run-time of the parallel system $T_{||}$ is the time until *one* of the processors has found a proof, $T_{||} = \min_p(T_{\mathcal{A}(\mathcal{P}_1^p,\ldots,\mathcal{P}_n^p)})$. The time needed to start and stop the system is not considered here. For a discussion of these times, see [4] and Section 4. All proof attempts (sequential and parallel) have been aborted after a maximal run-time of $T_{max} = 300s$. Hence, for all examples $T_{seq} \leq 300s$ and $T_{||} \leq 300s$. Given the run-times, we define the speed-up $s = T_{seq}/T_{||}$.

It is rather difficult to give a good estimation of a mean value for the speed-up for a set of examples, since in many cases the speed-up shows a high variance. Therefore, different definitions of mean values result in extremely varying results. For a discussion of this topic see e.g. [5]. For our measurements, we give four common mean values:

$$\begin{array}{ll} \bar{s_a} = \frac{1}{n}\sum_i s_i & \text{(arithmetic mean)} \\ \bar{s_g} = \sqrt[n]{s_1 \cdot \ldots \cdot s_n} & \text{(geometric mean)} \\ \bar{s_h} = \frac{n}{\sum_i 1/s_i} & \text{(harmonic mean)} \\ \bar{s_t} = \sum_i T_{seq}(i) / \sum_i T_{||}(i) & \text{(waiting times)} \end{array}$$

The arithmetic mean is often too optimistic, resulting in a mean value too large. The harmonic mean gives rather low values, because examples with speed-up values near 1 are taken unproportionally high into account. Often, the geometric mean is considered appropriate. $\bar{s_t}$, as defined in [1], relates the time needed to solve all problems (one after the other) with one processor to the time needed with p processors. This measure is especially useful for applications of the theorem prover, where one proof obligation after the other is to be solved. For the calculation of $\bar{s_t}$, a time-limit T_{max} must be set.

Here, we will give values for all four mean values. Furthermore, we present a graphical representation of the ration $T_{||}$ over T_{seq} for each measurement; a representation which allows to make reasonable estimations of the system's behaviour even in cases of varying speed-ups. In this representation, areas of super-linear speed-up and areas where $s < 1$ are marked.

[2]The default parameters of the SETHEO system are using all possible constraints (`-cons`), and performing iterative deepening over the depth of the proof-tree (A-literal depth, `-dr`).

3 Parameter Competition for SETHEO

3.1 The Sequential Prover SETHEO

SETHEO is a theorem prover for First Order Predicate Logic based on the Model Elimination Calaculus [9]. Given a set of clauses SETHEO tries to construct a closed Model Elimination Tableau (a labelled tree) by performing extension and reduction steps. The SETHEO Abstract Machine performs top-down search. Completeness is guaranteed by limiting the number of inferences of the current tableau or its depth, and performing iterative deepening.

Furthermore, SETHEO features a variety of efficient methods for pruning the search space. For details about SETHEO see e.g. [8, 7, 6].

3.2 Parameters for Competition

The Model Elimination Calculus and SETHEO's proof procedure can be parameterized in several ways. Table 1 shows a number of typical ways for modifying the basic algorithm. For each parameter, common values are shown. Values which are default for SETHEO are given in bold-face. Whereas some parameters can directly be given as command line options, other parameters cause small modifications of the source code of SETHEO.

parameter	values
Clause selection function	as in formula/random/**heuristically ordered**
Literal selection function	as in formula/random/**heuristically ordered**
Search mode	**top-down**/bottom-up/combination
addt'l inference rules	-/fold-up/unit-lemmata
completeness modes	**iterative deepening**/ other fair strategies
completeness bounds	**depth**/#inferences/#copies/combinations
pruning methods	**constraints**/-/

Table 1: Basic parameters for SETHEO's calculus and proof-procedure. Values shown in bold-face are default values for SETHEO.

Regardless of the parameter setting, the proof procedure of SETHEO is always sound and complete. This means for a competitive parallel theorem prover based on SETHEO that soundness and completeness of the entire system is always ensured.

Given the parameters and their possible values from Table 1, a large variety of different parallel theorem provers based on competition can be designed. Combinations of parameters furthermore increase this number.

In the following, we will focus on three parallel competitive systems, based on SETHEO. Since they compete on rather simple settings of parameters, the system is called SiCoTHEO (*Si*mple *Co*mpetitive provers based on SETHEO). The three systems compete via different

completeness modes ("parallel iterative deepening", SiCoTHEO-PID), via a combination of completeness bounds (SiCoTHEO-CBC), and a combination of top-down and bottom-up processing (SiCoTHEO-DELTA). Competition on selection functions has been implemented in the parallel prover RCTHEO and has been explored in detail in [4]. Before we go into details of each prover, we sketch the common prototypical implementation for all SiCoTHEO provers.

4 Prototypical Implementation

SiCoTHEO has been implemented in a prototypical way, using the same basic tools. SiCoTHEO is running on a (possibly heterogenous) network of UNIX-workstations. The control of the proving processes, the setting of the parameters and the final assembly of the results is accomplished by the tool *pmake* [2]. This implementation of SiCoTHEO is a further development of a prototypical implementation of RCTHEO.

Pmake is a parallel version of *make*, a software engineering tool used commonly to generate and compile pieces of software given a number of source files, using a dependency graph. Pmake exploits parallelism because it tries to export as many independent jobs as possible to other processors. Hereby it assumes that all files are present on all processors (e.g., via NFS).

Pmake stops, as soon as all jobs are finished or an error occurred. In our case, however, we need a "winner takes all strategy" which stops the system, as soon as *one* job is finished. This can be accomplished easily, using pmake's default action in case an error is encountered: an error causes pmake to abort the entire processing. Therefore, SETHEO had to be adapted only in such a way that it returns an error value (i.e., a value $\neq 0$), if it found a proof.

In contrast, the implementation of RCTHEO had to transfer the output generated by all provers to the master processor. There, a separate process searched for a success message. This resulted in heavy network traffic and long delays.

A critical issue in using pmake is its behaviour w.r.t. the load of workstations: as soon as there is activity (e.g., keyboard entries) on workstations used by pmake, the current job will be aborted (and restarted later). Therefore, the number of active processors (and even the start-up times) can vary strongly during a run of SiCoTHEO.

5 Evaluation and Results

In this section we look in detail at the results, obtained with the three different versions of SiCoTHEO. The experiments have been carried out on a network of HP-750 workstations, connected via Ethernet. All formulae for the experiments have been taken from the TPTP-problem library [12].

5.1 SiCoTHEO-PID

Parallel iterative deepening is one of the simplest forms of competition: each processor explores the search space to a specific bound. For example, if we have the A-literal depth as the bound, we might want to start processor number i with depth bound i. In order to ensure completeness with a limited number of processors, we obtain a slightly different scheme (for P processors):

```
for k=1,...,P in parallel do
   for i=0,... do
      depth_bound = k + i*P;
      setheo(depth_bound)
```

Then, each processor i explores the search space to a bound of $i, i + P, i + 2P, \ldots$.

Due to time and resource restrictions, results on SiCoTHEO-PID have been obtained by evaluating existing run-time data of SETHEO[3]. Figure 1 shows the resulting mean values of the speed-up for different numbers of processors.

As can be seen immediatedly, the variance of the speed-up values is extremely high. This fact results in a high arithmetic mean, whereas the $\bar{s_g}, \bar{s_h}$ and $\bar{s_t}$ are very close to 1. This behaviour can also be seen in Figure 2 which shows the ratio of $T_{||}$ over T_{seq} for each example, using 5 processors. The speed-up s is always ≥ 1 since the entire search space (which has to be searched in the sequential case) is partitioned. Nevertheless, the prover on each processor is complete, because if there exists a proof with depth d_0, the proof can be found as well with a larger depth bound $d \geq d_0$.

Furthermore, it is evident from Figure 1 that SiCoTHEO-PID is not scalable. The speed-up values reach a saturation level already with 4 processors. Adding more processors does not increase the speed-up any more. This behaviour is obvious, since about two third of the examples (67%) could be solved with a depth bound of 3 or 4. The number of examples which need a higher depth (and thus can occupy more processors) is rather low, as the histogram in Figure 3 shows.

Although in many cases, high speed-up values can be obtained, SiCoTHEO-PID should be used in applications only where deep proofs are expected.

5.2 SiCoTHEO-CBC

The completeness bound which is used for iterative deepening determines the shape of the search space and therefore has an extreme influence on the run-time the prover needs to find a proof. There exist many examples, for which a proof cannot be found using iterative deepening over the depth of the proof tree, whereas iterative deepening over the number of inferences almost immediatedly reveals a proof, and vice versa[4].

[3]The data have been obtained by running SETHEO (V3.0) on all examples of the TPTP [14] with $T_{max} = 1000s$. For our experiments, we have selected all examples which have a run-time $T_{seq} \leq 1000s$ on a HP-750.

[4]This dramatic effect can be seen clearly in e.g. [8], Table 3.

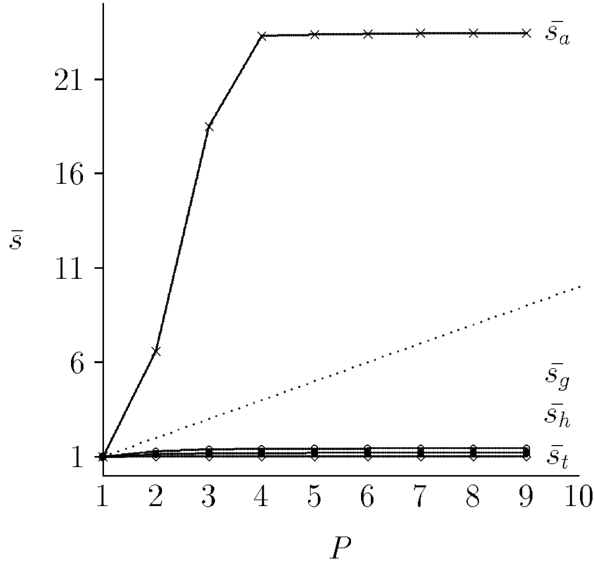

Figure 1: SiCoTHEO-PID: mean speed-up values ($\times$ for $\bar{s}_a$, $\circ$ for $\bar{s}_g$, $\bullet$ for $\bar{s}_h$, $\diamond$ for $\bar{s}_t$) for different numbers of processors. The dotted line marks linear speed-up.

In order to level both extremes, R. Letz[5] proposed to combine the A-literal-depth bound d with the inference bound i. When iterating over depth d, the inference bound i is set according to $i = f(d)$ for some function f. For our experiments, we use a quadratic polynome:

$$i = \alpha d^2 + \beta d$$

where $\alpha, \beta \in R_0^+$[6].

SiCoTHEO-CBC (*Com*B*ine* *C*ompleteness bounds) explores a set of parameters $\langle \alpha, \beta \rangle$ in parallel by assigning different values to each processor. For the experiments we selected $0.1 \le \alpha \le 1$ and $0 \le \beta \le 1$. In our first experiment we used 50 processors with the following parameter setting:

$$\begin{array}{rrcr} p_1 : \langle 0.1, 0.0 \rangle & \langle 0.1, 0.2 \rangle & \ldots & \langle 0.1, 0.8 \rangle \\ \langle 0.2, 0.0 \rangle & \langle 0.2, 0.2 \rangle & \ldots & \langle 0.2, 0.8 \rangle \\ \ldots & \ldots & \ldots & \ldots \\ \langle 1.0, 0.0 \rangle & \langle 1.0, 0.2 \rangle & \ldots & p_{50} : \langle 1.0, 0.8 \rangle \end{array}$$

Note, that this grid does not reflect the architecture of the system. It just represents a two-dimensional arrangement of the parameter values.

For Exp. 2 and Exp. 3, the number of processors was reduced to 25 and 9 respectively by equally thinning out the grid. For all experiments, a total of 99 different formulae from the TPTP have been used. 48 examples show a sequential run-time T_{seq} of less than one

[5] Personal communication.

[6] For $\alpha = 0, \beta = 1$ we yield inference-bounded search, $\alpha = \infty, \beta = \infty$ corresponds to depth-bounded search.

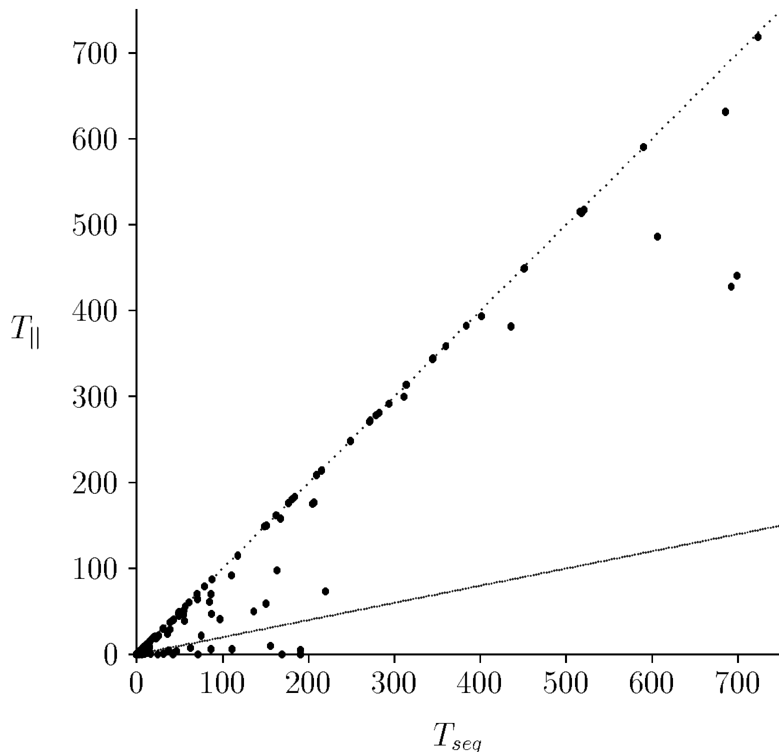

Figure 2: SiCoTHEO-PID: parallel run-time $T_{||}$ over sequential run-time T_{seq} for $P = 5$. The dotted line corresponds to $s = 1$, the solid line to $s = P$.

second CPU-time. 36 of the remaining examples have a run-time which is higher than 100 seconds. Although measurements have been made with all examples, we do not present the results for those with run-times of less than one second. In that case, the resulting speed-up ($\bar{s_a} = 1.57$ for $P = 50$) is by far outweighted by the time, SiCoTHEO needs to export proof tasks to other processors. In a real application, this could be reflected by the following strategy: first, start one sequential prover with a time-limit of 1 second. If a proof cannot be found within that time, SiCOTHEO-CBC would start exporting proof tasks to other processors.

Table 2 shows the mean values for all three experiments and different intervals of sequential run-times. These figures can be interpreted more easily when looking at the graphical representation of the ratio between T_{seq} and $T_{||}$, as shown in Figure 4. Each dot represents one example. The area above the dotted line contains examples where the parallel system is *slower* than the sequential prover, i.e., $s < 1$. The solid line has a gradient of $1/P$. All examples, located in the area below that line have a superlinear speed-up $s > P$.

Figure 4 shows that even for a small number of processors a large number of examples with super-linear speed-up exist. This encouraging fact is also reflected in Table 2 which exhibits good average speed-up values even for $P = 9$. For the long-running examples and $P = 9$, $\bar{s_g}$ is even larger than the number of processors. This means that in most cases, a

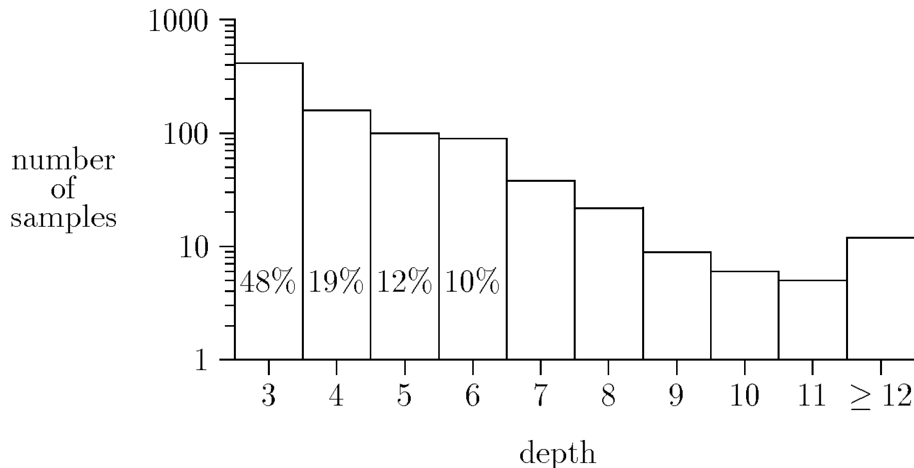

Figure 3: Number of examples with a proof found with A-literal depth d over the depth d. Numbers are % of the total number of 858 samples.

super-linear speed-up can be accomplished.

Furthermore, SiCoTHEO-CBC seems to be comparatively scalable. Table 2 shows that with an increasing number of processors, the speed-up values are also increasing. However, this model has not been tested for larger networks of processors (and possibly different ranges for α and β).

seq. run-time [# samples]		$P = 9$	$P = 25$	$P = 50$
$1s \leq T_{seq} < 10s$[3]	$\bar{s}_a$	3.59	3.58	15.32
—"—	$\bar{s}_g$	2.00	2.00	3.77
—"—	$\bar{s}_h$	1.35	1.35	1.64
—"—	$\bar{s}_t$	2.40	2.40	3.25
$10s \leq T_{seq} < 100s$[12]	$\bar{s}_a$	37.89	39.33	51.81
—"—	$\bar{s}_g$	18.80	20.64	25.26
—"—	$\bar{s}_h$	5.65	7.43	7.68
—"—	$\bar{s}_t$	4.81	6.80	7.00
$100s \leq T_{seq}$ [36]	$\bar{s}_a$	65.40	74.52	110.10
—"—	$\bar{s}_g$	9.65	14.40	16.14
—"—	$\bar{s}_h$	2.86	3.56	3.72
—"—	$\bar{s}_t$	2.66	3.34	3.51

Table 2: SiCoTHEo-CBC: Mean values of speed-up for different numbers of processors P.

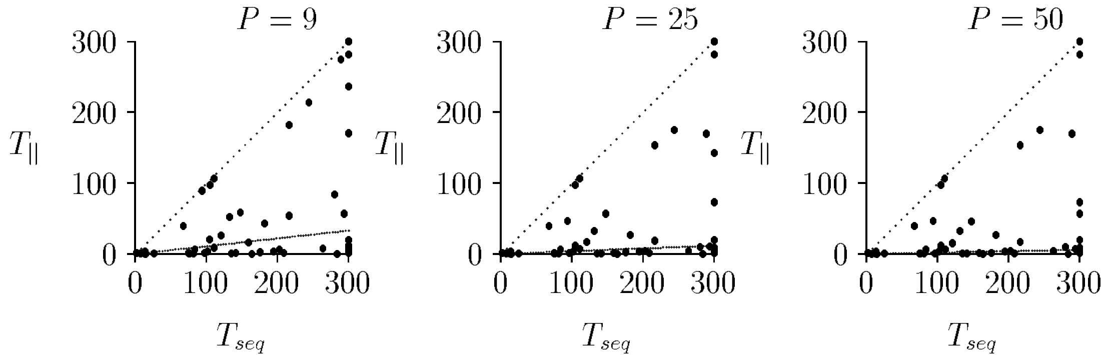

Figure 4: Parallel run-time $T_{||}$ over Sequential run-time T_{seq} for SiCoTHEO-CBC and different numbers of processors. The dotted line corresponds to $s = 1$, the solid line to $s = P$.

5.3 SiCoTHEO-DELTA

The third competitive system which will be considered in this paper affects the search mode of the prover. SETHEO normally performs a top-down search. Starting from a goal, Model Elimination Extension and Reduction steps are performed, until all branches of the tableau are closed. The DELTA iterator [10], on the other hand, generates small tableaux, represented as unit clauses in a bottom-up way during a preprocessing phase. These unit clauses are added to the original formula. Then, in the main proving phase, SETHEO works in its usual top-down search mode. The generated unit clauses now can be used to close open branches of the tableau, thus combining top-down with bottom-up processing.

The DELTA preprocessor has various parameters to control its operation. Here, we focus on two parameters: the number of iteration levels l, and the maximal allowable term depth t_d. l determines how many iterations the preprocessor executes[7]. The number of generated unit clauses increases monotonically with l. In order to avoid an excessive generation of unit-clauses, the maximal depth of a term t_d may be restricted. E.g., for the term $f(a, g(b))$ we have $t_d = 3$. Furthermore, DELTA is configured in such a way that a maximal number of 100 unit clauses are generated. This avoids run-time errors during the top-down phase when it is attempted to process excessively large formulae.

For our experiments, we use competition on the parameters l and t_d of DELTA. Then, a (possibly) different number of unit clauses are generated on each processor. The resulting formula is then processed by SETHEO, using standard parameters[8]. Hence, execution time in the parallel case consists of the time needed for the bottom-up iteration T_{DELTA} plus that needed for the subsequent top-down search T_{td}. As before, the overall execution times of the abstract machine, including the time to load the formula is used.

[7]In case of Horn clauses, one iteration level corresponds to one step of UR-resolutions with the entire formula.

[8]The default parameters for SETHEO are used when the script `setheo` is invoked. Then, all constraints (`-cons`) are used and iterative deepening over the depth of the proof-tree `-dr` is performed.

Our experiment has been carried out with 40 examples, 32 of which have $T_{seq} \geq 100s$. l ranges from 1 to 5, t_d from 1 to 5, resulting in 25 different pairs of parameters. Figure 5 shows the ratio between T_{seq} and $T_{||}$ for all examples. Again, we vary the number of processors (4, 9, 25). Table 3 shows the numeric values for the obtained speed-up.

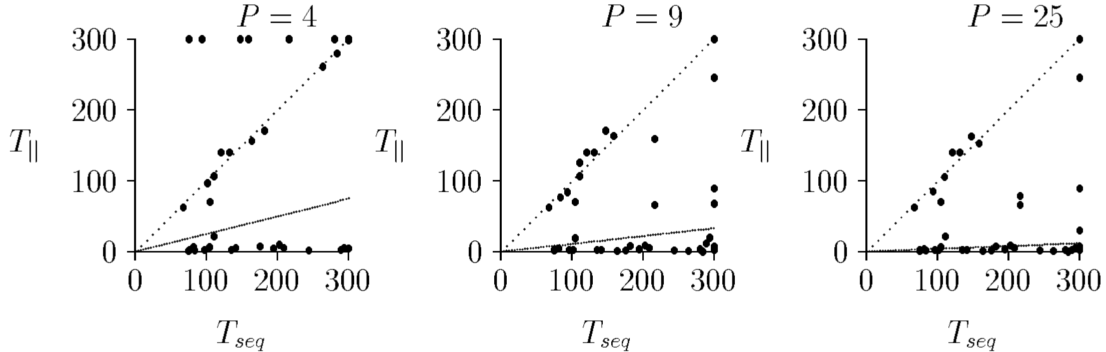

Figure 5: Parallel run-time $T_{||}$ over Sequential run-time T_{seq} for SiCoTHEO-TDBU and different numbers of processors.

seq. run-time [# samples]		$P = 4$	$P = 9$	$P = 25$
$T_{seq} \leq T_{max}$ [40]	$\bar{s}_a$	19.69	62.12	73.50
—"—	$\bar{s}_g$	4.08	11.88	16.63
—"—	$\bar{s}_h$	1.28	2.84	3.52
—"—	$\bar{s}_t$	1.34	3.35	4.00

Table 3: Mean values of speed-up values for SiCoTHEO-DELTA

The speed-up values obtained with SiCoTHEO-DELTA are also very good. For $P = 4$ and $P = 9$ the average speed-up $\bar{s}_g$ is larger than P. This fact is supported by the large number of examples which exhibit a super-linear speed-up as depicted in Figure 5.

However, there are several cases in which the parallel system is running slower than the sequential one. The reason for this behaviour is that the additional unit-clauses increase the search space. In cases, these clauses cannot be used for the proof, the run-time to find a proof increases. In the case of $P = 4$ this can lead to the case that a proof cannot be found at all within $T_{max} = 300s$, even if $T_{seq} < 300s$. For a larger number of processors, this effect does not occur for our examples. Here, there always exist a parameter setting for DELTA which causes the overall run-time not to increase substantially.

6 Conclusions

In this paper, we have presented a parallel model based on the theorem prover SETHEO. Parallelism is exploited by homogeneous competition. Each processor in the network is running SETHEO and tries to prove the entire formula. However, on each processor, a different set of parameters influence the search of SETHEO. If this influence results in large variations of the run-time, good speed-up values can be obtained. In this work, we had a detailed look at three different systems based on this model: SiCoTHEO-PID performs parallel iterative deepening, SiCoTHEO-CBC combines two different completeness bounds using a parametrized function, and SiCoTHEO-DELTA combines the traditional top-down search of SETHEO with the bottom-up preprocessor DELTA. The parameters of DELTA are the basis for competition.

Although the speed-up values of SiCoTHEO-PID are always larger than one, only little efficiency could be obtained by this prover. Furthermore, its peak performance is reached with only 4 processors.

On the other hand, the parameterized combination of completeness bounds bear a large potential for efficiency. Good efficiency is generally restricted to examples which exhibit longer run-times (at least about $1s$). Then, extremely good speed-up values could be obtained, even with a low number of processors. Up to 50 processors, a reasonable increase in performance could be observed. Further experiments have to reveal, how well SiCoTHEO-CBC scales up to larger networks of processors.

SiCoTHEO-DELTA exhibits a similar and even slightly better behaviour concerning the obtained speed-up values ($\bar{s}_g, \bar{s}_t$). The controlling parameters of DELTA are still rather coarse. Therefore, many processors work on identical preprocessed formulae. The speed-up could be even better, if one succeeds in producing a greater variety of preprocessed formulae.

The implementation of SiCoTHEO, using *pmake* combines simplicity and high flexibility (w.r.t. the network and modifications) and with good performance. Future enhancements of SiCoTHEO will certainly incorporate ways to control DELTA's behaviour more subtly. Furthermore, a combination of SiCoTHEO-DELTA and SiCoTHEO-CBC will increase the overall efficiency and scalability substantially. Finally, experiments with SiCoTHEO can reveal, how to set the parameters of sequential SETHEO and DELTA in an optimal way.

References

[1] F. Buffoli, G. Degli Antoni, and A. Marchese. OR-Parallelism in Theorem Proving: Speedups versus Timeout. Technical report, State University, Milano, 1994.

[2] A. de Boor. *PMake – A Tutorial.* Berkeley Softworks, Berkeley, CA, January 1989.

[3] W. Ertel. OR-Parallel Theorem Proving with Random Competition. In *Proceedings of LPAR'92*, pages 226–237, St. Petersburg, Russia, 1992. Springer LNAI 624.

[4] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.DISKI 25.* Infix-Verlag, St. Augustin, 1993.

[5] W. Ertel. On the Definition of Speedup. In *PARLE, Parallel Architectures and Languages Europe.* LNCS, Springer Verlag, 1994.

[6] C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments (System Abstract) . In *CADE 12*, pages 778–782, June 1994.

[7] R. Letz, K. Mayr, and C. Goller. Controlled Integration of the Cut Rule into Connection Tableau Calculi. *Journal of Automated Reasoning*, 13:297–337, 1994.

[8] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

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

[10] J. Schumann. DELTA — A Bottom-up Preprocessor for Top-Down Theorem Provers. System Abstract. In *CADE 12*, Springer 1994.

[11] J. Schumann and R. Letz. PARTHEO: a High Performance Parallel Theorem Prover. In *CADE 10*, Springer 1990.

[12] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *CADE 12*, Springer 1994.

[13] C.B. Suttner and J. Schumann. Parallel Automated Theorem Proving. In *Parallel Processing for Artificial Intelligence.* Elsevier, 1993.

[14] C.B. Suttner. *Static Partitioning with Slackness.* DISKI. Infix-Verlag, to be published.