P A R T H E O:
A High Performance Parallel Theorem Prover

J. Schumann and **R. Letz**

Forschungsgruppe Künstliche Intelligenz
Institut für Informatik, Technische Universität München
Augustenstr. 46, D-8000 München 2
Tel: +49–89–52 10 98
e-mail: NAME*@lan.informatik.tu-muenchen.dbp.de*

Abstract. PARTHEO, a sound and complete or-parallel theorem prover for first order logic is presented. The proof calculus is model elimination. PARTHEO consists of a uniform network of sequential theorem provers communicating via message passing. Each sequential prover is implemented as an extension of Warren's abstract machine. PARTHEO is written in parallel C and running on a network of 16 transputers. The paper comprises the system architecture, the theoretical background, details of the implementation, and results of performance measurements.

Keywords. Automated deduction, theorem proving, first order logic, connection method, model elimination, or-parallelism, message passing, transputers.

1 Introduction

AI systems are in constant need of powerful and efficient inference mechanisms. Within many areas of application logic-based languages seem to be an appropriate choice. A popular and successful realization is Prolog, which is moderately efficient, but lacks in expressiveness, mainly because of its restriction to Horn clauses. Certainly, full first order logic would be more powerful and comfortable, but at the price of a more difficult implementation. One promising approach is to use the model elimination calculus, which is complete for first order logic. As this calculus is sequence-based and needs no factoring it seems even more machine-oriented than resolution, which is a set-based calculus. Model elimination can efficiently be implemented, e.g. using Prolog-like technology. This has been convincingly demonstrated with Stickel's PTTP [Sti88] and, recently, with our SEquential THEOrem prover SETHEO [LBSB89].

A further promising potential for gaining high efficiency is the use of parallelism. In the field of theorem proving, particularly, the huge search space poses a severe problem. Therefore it seems a good decision to distribute and explore the search space in parallel, a method known as or-parallelism. This type of parallelism involves that on each node there must reside an efficient inference machine. Since with SETHEO we already had developed a very efficient inference machine, it was natural to employ it as the kernel in the design of PARTHEO, with the additional advantage that almost all

[0]This work was supported by the European Community and Nixdorf Computer AG within the ESPRIT project 415F.

features of the sequential SETHEO carry over into our parallel system. Thus PARTHEO can be used both as a theorem prover and an interpreter of LOP, a logic programming language under development. It offers, e.g. the use of (non-logical) global variables and destructive assigment, with a clear and simple declarative semantics [LS88].

From the hardware point of view we aim at a large number of processors. Current technology seems to favour networks of processors communicating via message passing without common memory. Such systems also allow for a good scalability. Practical considerations led us to the decision to implement PARTHEO on a network of transputers.

The paper is organized as follows. The next section sets out the modular architecture of the entire system. The third section describes the underlying proof calculus. The exploitation of parallelism is described in the fourth section. In the fifth section we present details of the implementation of the network and the parallel abstract machine APM. Some results of performance measurements are given in the sixth section. Related work is briefly summarized in the seventh section. We conclude with assessing the main features and perspectives of our approach.

2 System Architecture

Our system is strongly modularized. It consists of five independent modules: *transformation part, preprocessor, compiler, main proof procedure*, and *user interface*. PARTHEO differs from the sequential SETHEO only in the main proof procedure: in PARTHEO it is implemented and executed on a parallel machine, a network of transputers. The modules of the system and the control flow are depicted in Figure 1.

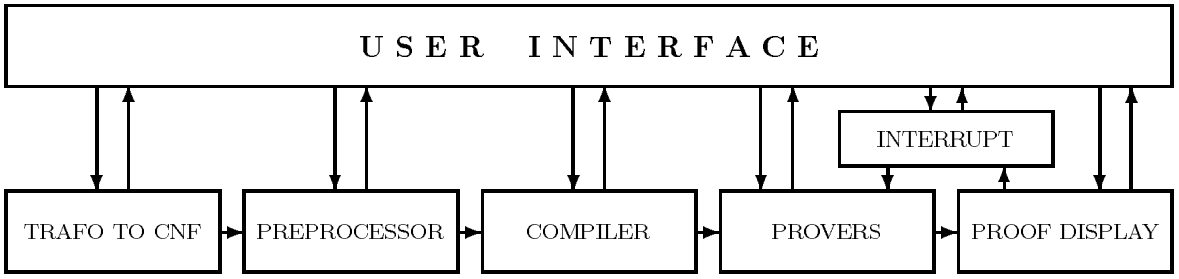

Figure 1: System architecture

Since the core of the system can handle clausal form formulae only, a *transformation module* is added which converts formulae in ordinary logical notation, i.e. formulae containing quantifiers, logical operators, etc. into Skolemized clause normal form. We offer two transformations: one is the standard equivalence preserving method, which is of exponential complexity in the worst case; the other one is the method described in [Ede85], which is polynomial. The other modules work with formulae in clause normal form only.

The *preprocessing part* aims at reducing the size of the input formula before initiating the main proof procedure. It combines features that just eliminate redundant parts of the input formula with others that anticipate inferences of the proof process itself. Currently our preprocessing module consists of five independent submodules: tautology reduction, restricted unit resolution, subsumption reduction, purity reduction, and isolated connection reduction. For a detailed description consult [LBSB89].

The inference machine, the sequential version as well as the parallel one, is realized as an abstract machine. Hence the formula must be compiled into abstract machine code. In the *compilation phase* the clauses of the input formula are translated into contrapositives. Optionally, syntactical pruning constraints are inserted. For optimization purposes, the clauses and literals are reordered and a connection graph, expressing all unifiable connections between the literals of the formula, is generated. The latter helps to avoid a lot of useless unification attempts. Afterwards the contrapositives are translated into extended Warren code, which is loaded onto the network.

During the *main proof procedure* these abstract machine instructions are interpreted in parallel by a network of Abstract Partheo Machines.

A *user interface* gives information about the result of the proof proces. For instance, the proof tree can be displayed graphically. This interface can also be used during the proof process for visualizing the current state of the parallel system.

A detailed description of the entire system and its modules can be found in [LBSB89] and [BEvdK+89].

3 The Proof Calculus

The proof calculus underlying the inference system is model elimination. Model elimination has first been introduced in the light of resolution (see, e.g. [Lov78]), which is a formula-producing *synthetic* calculus. But by its very nature model elimination belongs to the family of *analytic* calculi based on semantic trees [Bet59], like the tableau calculus [Smu68].

More precisely, model elimination can be seen as a special variant of the connection method [Bib87]. The connection method offers a general framework for the proof-theoretic treatment of logical formulae. It is based on the idea of reducing the task of proving a formula to the task of checking all paths through the formula for tautology or contradiction.[1] Model elimination can be seen as a very simple realization of this idea in a linear format, which is easily described with labeled trees or tableaux.[2]

Definition 3.1 A *connection tree* or *tableau* T for a formula F is a pair (t, μ), where t is a finite tree and μ is a function from the non-root nodes of t onto literals such that for any complete set of nodes N with the same immediate predecessor (*parent*) the range μN (called *tableau clause*) is an instance of a clause in F.

[1] To follow common practice we are dealing with the refutation of formulae, which is in contrast to the affirmative approach in [Bib87].

[2] We prefer the framework of the connection method, because in the original setting (as, e.g. in [Lov78]) model elimination is restricted to a fixed depth-first right-most execution model.

Definition 3.2 A tableau is called *closed* if every complete branch (*path*) contains nodes K and L such that $\mu(K)$ contradicts $\mu(L)$, i.e. K and L are *complementary* nodes.

It holds that a formula is unsatisfiable if and only if there is a closed tableau for the formula (for a proof see, e.g. [Bib87]). Model elimination imposes further restrictions on connection trees which do not affect completeness.

Definition 3.3 A *model elimination tableau* is a tableau satisfying the condition that every node, which is not the root or a leaf, has at least one complementary node among its immediate successors (*children*).

In practice, a closed tableau is constructed stepwise starting from the root by attaching new nodes to intermediate tableaux and by explicitly labelling branches as closed at their leaf nodes. In these steps substitutions are imposed onto the variables of the affected literals as induced by unifiers[3]. It is useful to differentiate the single construction steps of such a tableau starting from the children of the root (the *start clause*) into *extension steps* and *reduction steps*. For convenience we often identify a node with its value, the literal.

Extension step: suppose there is a unifier σ for a leaf L in a tableau and a literal $\neg L$ in a (renamed) clause c of the underlying formula, then perform the unification, take the literals in $c\sigma$ as children of $L\sigma$, and label the branch at $\neg L\sigma$ as closed.[4]

Reduction step: suppose there is a unifier σ for a leaf L and some ancestor $\neg L$ in a tableau, then perform the unification and label the branch at $L\sigma$ as closed.[5]

A node (tableau) holds as *open* as long as it (all its nodes) is (are) not explicitly closed. It holds that each model elimination tableau may be constructed by performing a sequence of extension and reduction steps (for a proof see [Lov78]).

Let us consider for illustration of these concepts a simple example of first order logic. It asserts that if for any pair of objects in a domain either a binary relation P or the inverse relation holds, then for every object in the domain there exists an object such that both relations hold:

$$\forall x \forall y (P(x,y) \vee P(y,x)) \rightarrow \forall v \exists z (P(v,z) \wedge P(z,v)).$$

Aiming at refuting the negation of the proposition, we shift to negated Skolemised conjunctive clause normal form, and get the following set of clauses.

Example 3.1 $\{\{P(x,y), P(y,x)\}, \{\neg P(a,z), \neg P(z,a)\}\}$

A closed tableau for this formula is shown in Figure 2 below. The respective substitutions for the variables in the copies of the original clauses are given on the right, and closed branches are labeled with an asterisk for extension ($*_E$) and reduction ($*_R$) respectively.

[3]For completeness it is sufficient to use *most general* unifiers.

[4]This is the counterpart of binary input resolution within model elimination.

[5]This terminology must be distinguished from the one in [Bib87] where the combination of both steps is called *extension step*.

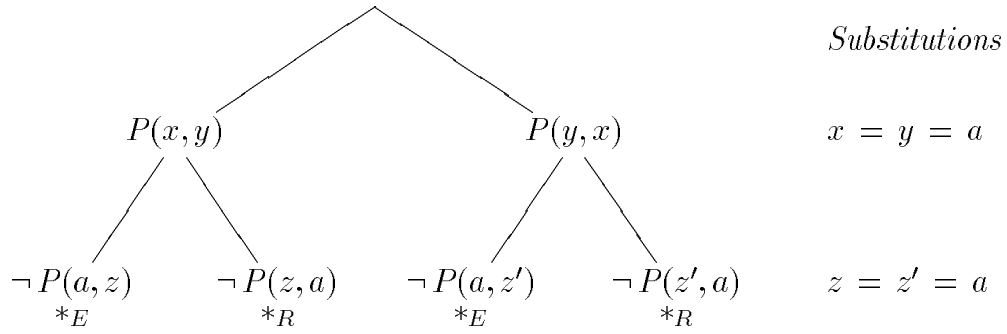

Figure 2: A model elimination tableau

An crucial property of model elimination is its independence from the node selection function.

Definition 3.4 A *tableau node selection function* $\mathcal{S}$ assigns to each open tableau one of its open leaves, at which the next inference step must be performed.

Let T be a closed model elimination tableau constructed by the sequence I of extension and reduction steps using a node selection function $\mathcal{S}$. Then it holds for each node selection function $\mathcal{S}'$ which yields a permutation of I and a tableau T', that T and T' are (renamable) variants of each other. In other words, it does not matter in which order extension or reduction steps are applied to open nodes of intermediate tableaux. (for a proof see, e.g. [BEK+89]).

The model elimination calculus may be refined and extended in various ways. We are investigating on *factorization*,*lemmata*, and *proof schemata*. For details see [LBSB89]. We have implemented a very powerful pruning mechanism, which is described in the following.

3.1 Regular model elimination

Model elimination is already a rather refined calculus in the sense that it avoids a lot of suboptimal connection proofs. But the model elimination calculus can be further refined without losing completeness.

Definition 3.5 A *regular*[6] *model elimination tableau* is a model elimination tableau with the following two properties:

- no tableau clause is a tautology[7],
- no node in the tableau has an ancestor node with an identical literal as value[8].

[6]This restriction is in some analogy to regular resolution [Tse70].

[7]This condition ordinarily is included in resolution provers, because it can be more easily verified than in model elimination implementations.

[8]This condition is also investigated in [Sti88].

For any unsatisfiable formula there is a closed regular model elimination tableau (for a proof see, e.g. [BEK+89]).

Unfortunately, the regularity of a tableau is difficult to detect. Therefore, in the current version this property is only approximated by checking it from time to time. For a detailed description of this procedure and its advantage over the full model elimination calculus consult [LBSB89] (compare also [Sti88]).

4 Exploitation of Parallelism

PARTHEO has been designed with the emphasis on theorem proving. In this field the main problem is not the length of a proof[9], but the large search space induced by the branching factor of the underlying calculus. To cope with this problem currently two main directions may be distinguished. One is to make the search algorithm more intelligent, i.e. to apply heuristics and pruning strategies in order to reduce the search space, the other is to do things in parallel. Although work is done in both directions, we here want to focus on parallelism.

If a parallel algorithm has to be designed it is useful to keep in mind the parallel machine on which this algorithm is to be implemented and executed. We have decided to aim at hardware relying on message passing and local memories, and not on common memory. The target hardware was a network of transputers. This choice promises *scalable* implementations, in that the topology and the size of the hardware may be varied easily.

4.1 Or-parallelism

One obvious way to distribute the search space over different processors is to apply *or-parallelism*. The basic idea of this notion is explained as follows. The search space induced by the model elimination calculus together with a given node selection function can be described as a tree of tableaux.

Definition 4.1 Let F be a formula consisting of n clauses, and let $\mathcal{S}$ be a tableau node selection function. The *model elimination or-search tree* for F and $\mathcal{S}$ is a tree, where all nodes are labeled with model elimination tableaux such that

1. the root is labeled with the empty tableau,
2. there are n children of the root, which are labeled with the n different tableaux constructable by just taking any clause in F as start clause,
3. if a leaf node N has an open tableau T as label, then the children of N are precisely the nodes with those tableaux for F as labels that may be constructed by applying exactly one inference step to the selected open leaf node $\mathcal{S}(T)$.

An example of such a structure is displayed in Figure 3 below. The triangles encode open ($\circ$) and explicitly closed ($*$) tableaux respectively. For each selected node $\mathcal{S}(T)$

[9] Except for very artificial examples. Some of these may be tackled by extensions of the calculus mentioned above (see [LBSB89]).

there exist as many succeeding nodes (and tableaux) as there are successful unifications with literals from F (extension) or with literals from the branch on which the node lies (reduction). As during their generation the tableaux get more and more instantiated we do not know in advance which of all possible unification attempts are successful. To take this into account we have added another sort of nodes (FAIL) to the or-tree, which represent the cases of unification failure[10].

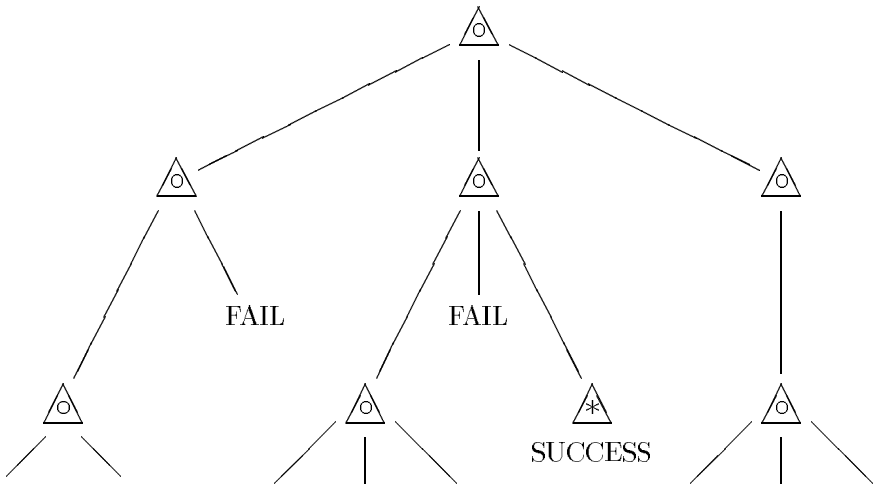

Figure 3: Structure of a model elimination or-search tree including failure nodes

Apparently, for a given formula each node selection function uniquely determines an or-search tree and its branching behaviour. Therefore some selection functions may keep the branching rate smaller than others. In the current implementation we use a very simple node selection function, viz., depth-first left-most selection, as in Prolog[11].

During the proof process the or-search tree of a formula must be explored from the root to the leaves. Or-parallelism distributes the *branches* of the or-search tree between the processors.

4.2 The basic algorithm

We assume that each processor has as program precisely the same inference machine and the compiled formula. The variety comes in by giving different input to each processor by communicating with other processors. This input is given essentially in form of proof *tasks*, where each task corresponds to one node in the or-search tree including failure nodes of the formula. Each processor consists of three modules, the proof unit, the communication unit and the local task memory.

[10]Note that in the compilation phase we apply pre-unification to construct a connection graph in order to reduce the number of these failure nodes.

[11]Note that during compilation we also apply some static selection mechanism according to such branching criteria by reordering the literals inside each clause.

During the execution process each processor may behave differently according to the three possible situations occurring during the treatment of tasks. First of all, a task may be *solvable*, viz., if the encoded tableau is closed. In this case a proof has been found, which is then reported to the user. Secondly, a task may *fail*, which is in case that for the encoded tableau no expansion is possible under the given constraints. This holds for instance if no unifiable partner may be found for the selected node, or if some completeness mode prohibits an expansion, as explained below. Finally, a task may *ramify* into finitely many subtasks, if there exist different inference possibilities for the selected open leaf of the encoded tableau. This last case makes up the potential for parallelism. The respective proof task gives rise to a number of new tasks, which encode just the children in the or-search tree including failure nodes. The processor cancels the old task, encodes the new tasks, and puts them into its local task memory, thus making them available to other processors[12].

4.3 The task stealing model

Parallelism is achieved by propagating tasks over the network. We are striving for a good load balancing without too much communication overhead. To this end we subscribe to the philosophy that a processor should only send tasks to his neighbours *on request*.

More precisely, PARTHEO employs a *task-stealing* model [BCLS89]. If a processor executes a task which fails, it tries to get a new one from its local task store. In case it is empty, the processor asks its direct neighbours for work. Since all processors must check from time to time for incoming task requests, the respective idle processor will get tasks from one of its neighbours which has tasks — say half of them. Thus tasks are only transmitted over the network, if no work is available locally.

The proof process is initialized by setting all processors into the state of task request, and by simply giving the root task, which encodes the empty tableau, to an arbitrarily chosen processor.

4.4 The representation of tasks

A crucial issue for systems using message passing is the size of the transferred data. The main data to be communicated in our model are the proof tasks. As mentioned above, our inference machine is a variant of the Warren abstract machine. This means that the data corresponding to a logical proof task are not compactly represented but encoded in the *stack*, the *heap*, the *trail*, and the *control registers* of the abstract machine. For transferring a task this involves that a lot of data had to be copied to another processor, which is not feasible on a message passing system.

We have solved the problem of generating the same processor state on another processor in analogy to the method how genetic information is stored in a living cell: the DNA contains all the information for reproducing the cell. More precisely, we are

[12]Actually, the processor keeps one of the new tasks to execute it immediately.

encoding a task as the sequence of choices that have to be made at each branching point in the search tree. This sequence can be given as a list of integers, one for each inference step. For a fixed node selection rule, this sequence uniquely determines the desired processor state. This representation is very compact and puts no transmission problems. The *decoding* of a task consists in deterministically recomputing the desired state, and hence the respective tableau. A lot of double work, one could say. But recalling that most proofs in theorem proving are rather short, this provision does keep the loss in efficiency small[13]. Actually, we employ an optimised version, viz., the *partial restart*, which involves that in many cases it is not necessary to reproof the whole tableau from scratch. This is the case if the old (failed) task and the new task from the task store have a non-trivial common initial segment. Then the recomputation starts at the first inference for which both tasks disagree. Obviously this method reduces the amount of time needed to switch to a new task in the search tree. Additionally, if the tasks are organised in a LIFO manner, for the sequential execution the partial restart works nearly as efficient as an ordinary fail.

All in all, the parallel algorithm may be summarized in the following C-like program.

```
on all nodes do
  in parallel do {
    proof: while true do {
             get_task T from local_task_store;
             do partial_restart;
             if unification succeeds then
               if T is closed then          /* a proof has been found */
                 report_success_to_host;
               else {
                 generate_new_tasks T1,...,Tn from T;
                 put T1,...,Tn into local_task_store}
             else ;                         /* unification fails */
             }
    request: if isempty(local_task_store) then
               request_tasks_from_neighbour;

    send: if request_for_tasks_received then {
            n = cardinality of local_task_store;
            if n < 1 then
              send_back("no_tasks");
            else
              send_back_tasks(n/2);
            } }
```

Note that a detailed and formal specification of the informal description given in this section is available [JS89], formulated in the parallel and functional specification language FP2.

[13]Besides, this stays in concordance with our decision to use or-parallelism only.

4.5 The search strategy

In general the or-search-tree of a formula is infinite. Hence a *complete* search strategy must be employed. The most natural approach for achieving completeness in a finitely branching search tree is to search breadth-first, but in terms of efficiency this contradicts the design decision of our parallel algorithm as well as the structure of the Warren machine. Therefore PARTHEO realizes depth-first iterative-deepening search [Kor85], as in [Sti88]. The idea is to consider only a finite initial segment of the search tree. In case this segment is explored and does not contain a solution we take a greater segment, and so forth[14].

The or-search tree is cut by imposing constraints onto the tableaux in the nodes of the tree. These constraints are monotonic, i.e. if a tableau T violates them then do all tableau in the or-search tree dominated by T. We offer three different completeness modes, by bounding alternatively the *number of inferences* in a tableau, the *depth* of a tableau, or the *number of admissible copies of every rule* in a tableau, which are all monotonic conditions. If a task is executed and its tableau violates (one of) the given completeness condition(s), then this task just fails. In case a proof is not found within given bounds, these bounds are increased, and the proof process is resumed. Currently, this process is not fully automatic in PARTHEO for reasons described immediately.

Performing the iterative-deepening step in a sequential system is unproblematic. Whenever a finite segment of the search space has been entirely explored the internal proof algorithm comes up with a total fail. Then the bound(s) are increased and a new internal proof process is started. In the parallel case it gets more complicated. The problem is to determine exactly when the current finite segment of the search tree has been completely explored. In such a case, according to the parallel algorithm mentioned above, all processors must be in the state of requesting their neighbours for work. This can only be detected by stopping the whole system and by investigating the network node by node, which is very inefficient.

We propose another solution to this problem. To avoid the inefficiency arising from stopping the network, we allow tasks from different iterative-deepening levels to be in the network at the same time. To assure completeness, tasks from different levels are labeled with different priority values. And, whenever a task is to be taken from the task store, higher priorities must be preferred. This involves that we are not obliged to wait until one finite segment is completely explored. We represent locally in (some of) the processors the approximate task load of the system, which is updated from time to time. If this value falls short of a certain threshold for these processors the next iterative-deepening level gets initialized. Currently this mechanism is not fully implemented, the iterative-deepening is controlled by hand.

5 Implementation

PARTHEO is designed to run on a network of transputers; both, the number of proces-

[14]Actually, this can be viewed as a generalized version of breadth-first search.

sors and the topology may be arbitrary. On each node of the network, an identical copy of the Abstract Partheo Machine (APM) is running. The APM consists of a communication unit to perform the communication with other processors, a local task store, and a sequential inference machine to execute the tasks. PARTHEO is implemented in 3L-parallel C [3L88]. As the basic communication mechanism, synchronous message passing is used. Inside one transputer the access to global data structures is controlled via semaphores.

5.1 The network

Our hardware system consists of 16 + 1 T800's; the interconnection of processors may be changed by software so that experiments with several different topologies can be carried through. One of the transputers is the *host* processor, which is connected to the host system, in our case a sun 386i.

A typical configuration scheme (similar to a torus) is shown in Figure 4.

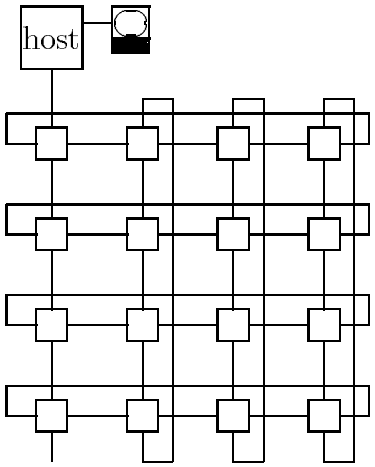

Figure 4: Typical hardware configuration scheme

5.2 The Abstract Partheo Machine – APM

In contrast to all other processors running an APM, the host processor contains no inference machine. Its only task is to interpret the command line, to open and read files, and to send commands to the other processors. Returning messages are interpreted and a success is displayed. During the search for a proof, the host processor also sends statistics requests to the other processors in regular intervals. The returned data are used to make measurements. In the following, however, we only look at the APM in detail.

An APM may be subdivided into three major parts, each of them consisting of one ore more threads, running concurrently on the processor:

SAM The kernel of an APM is the sequential inference machine of SETHEO, the SETHEO Abstract Machine SAM, which will be described in detail below. Its main action is to execute tasks and to generate new ones. To do this the original sequential machine had to be augmented with some new instructions to be able to communicate with the other parts of the APM.

Task Store The task store can be accessed by two processes: If the SAM encounters an or-node, newly created tasks are encoded and put into the task store. If the current task fails, the SAM takes a new task from the task store. If another processor requests tasks or if tasks are received, the communication unit gains control over the task store. It also has to check in regular intervals how many tasks are available in the task store. If it is empty, the task requesting mechanism is activated.

Communication Unit The communication unit of one processor contains several processes, which communicate via shared variables controlled by semaphores. The communication unit forwards messages to the host, receives and executes commands and handles the task stealing mechanism. Within the system three types of messages are exchanged between the processors:

- *Commands* are sent from the host processor to each processor or to a designated processor. They are used, e.g. to load the SAM-code into the code memory of the processors, to set parameters, and to start and stop the SAMs. A message from the host to a designated processor contains additional routing data. The communication unit reads this information and routes the message to one of its neighbouring processors. This allows to keep a global routing table on the host processor only.
- *Return messages* are sent from a single processor to the host processor. Such a return message can be a success or failure result, statistical data or error messages. They must be forwarded by the communication unit.
- *Task requests and sends.* Processors with an empty task store request tasks from their neighbouring processors in regular intervals. A timer process sends a task request to all direct neighbours of the processor in sequence. If a processor receives such a message it sends back, e.g. half of its own tasks, provided it has enough. Otherwise it informs the requesting processor, which tries to get tasks from another neighbour.

5.2.1 The SETHEO Abstract Machine (SAM)

The sequential inference machine SAM, which is both used in SETHEO and as the kernel of the APM, is implemented as an abstract, register-oriented machine. The machine and its instructions are based on Warren's abstract machine (WAM) [War83] and its RISC version [VH87]. The concepts introduced there had to be extended and enhanced for ending up in a complete and sound proof procedure based on the model elimination calculus, and for facilitating the use of advanced control structures and heuristics. The layout of the abstract machine is shown in Figure 5.

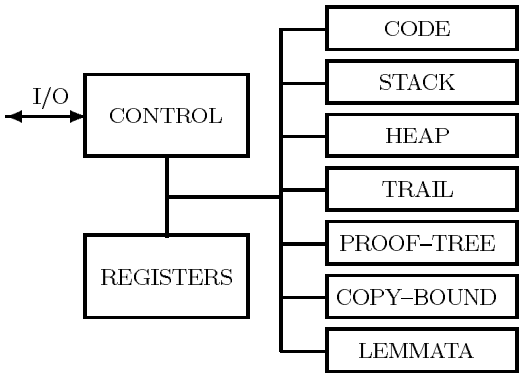

Figure 5: SETHEO Abstract Machine

Similar to the WAM, the *memory* of the SAM is tagged (4 bit) and consists of several parts. Besides the *code* area, the *heap*, *stack* and *trail*, which can also be found in the WAM, some other data areas are used in the SAM. The *proof-tree* stores the current state of the generated tableau, which can be displayed graphically to illustrate the structure of the proof process. In the *copy-bound* it is stored how many copies of each clause are still admissible. Generated lemmata are hold in the *lemmata* space. A detailed description of all the instructions and registers is available as a manual [STv89].

During the proof process full unification is rarely needed. Hence it is more efficient to have different instructions for the unification of one term according to its data type (variable, term, constant, etc.). For full unification the (exponential) Robinson algorithm is used. Optionally, we offer the more sophisticated Corbin algorithm [CM83], which has quadratic complexity. To reduce the number of occur-checks a simple heuristic is employed: an occur-check is omitted in extension steps into clauses for each first occurrence of a variable inside the head (compare also [Pla84]).

As the SAM had to be changed only very little to be integrated into the parallel PARTHEO system all features available in the sequential machine are also available in PARTHEO, e.g. global variables.

6 Performance Measurements

To get representive speed-up results of the system, we made measurements with problems from the field of theorem proving and logic programming. As theorem proving examples we took problems of the Wilson and Minker study [WM76], which are also used in [Sti88] as a set of benchmarks [15]. The *non-obviousness* example is a problem appearing in [PR86]. *Lukas11* is a formulation of a single axiom system for the propo-

[15]These examples have also been used to make performance measurements for SETHEO. For details see [LBSB89].

sitional calculus [Pfe88]. *Eder2-7* is an example of a formula class [Ede85] transformed to Horn clauses (using the transformation described in [Let88]). The other problems are taken from the field of logic programming, because they could be easily modified to serve as good speed-up benchmarks. The *Queens* examples are just the standard Prolog programs for solving the queens problem. The *Maximum* program is a quadratic program for computing the maximum of a list of integers. *PropSat* consists of a program for finding models of a propositional formula plus one example formula. Finally, *FairyTale* is a plan generation problem, which consists in finding the happy end of a fairy tale.

Problem	SETHEO [sec]	PARTHEO[16] [sec]	speed-up	rel. speed-up
Wos4	3413	1.843	1852	115.8
Wos9	64.0	2.16	29.7	1.86
Wos10	262.6	135.6	1.94	.121
Wos16	36490	5.89	6194	387
Wos25	575.3	114.2	5.03	.314
Wos29	254.5	48.1	5.29	.331
LS36	1666	352	4.73	.296
LS108	841.6	95.7	8.8	.550
LS121	44.24	10.25	4.32	.270
Non-obv.	4.5	1.84	2.44	.153
Lukas11	8.4	.657	12.78	.799
Eder2-7(H)	53.7	11.8	4.56	.285
Queens8	16.9	1.84	9.20	.575
Queens9	83.6	9.21	9.06	.566
Queens10	396	38	10.4	.650
Maximum	17.1	4.55	3.76	.235
PropSat	48.8	7.31	6.67	.417
FairyTale	24.0	1.81	13.7	.856

Figure 6: Run-time comparison of SETHEO (T 800) and PARTHEO (16 $\times$ T 800)

As a matter of fact, speed-ups in or-parallelism may vary strongly depending on where the solution is located in the search tree. For getting more reliable results, the following three problems are of great significance. *Queens8*, *Queens9*, and *Queens10* are modified in a way that only one chess-board position, which is exactly in the middle of the search tree, is exclusively accepted as the right solution. If then the search tree is distributed between the nodes, statistically, the correct speed-up (for this problem) can be computed. The same holds for *PropSat*, which contains a propositional formula with just one model.

Due to time limitations we were not interested in finding fastest proofs with SETHEO and PARTHEO. So we used depth-bounded search as the only completeness mode.

[16]Average run-time of 5 runs.

Since iterative-deepening is not yet fully supported all examples were tried with a fixed depth-bound sufficient to find a proof.

In the table of results in Figure6 the last row displays the *relative* speed-up, which is defined as the ratio of the speed-up to the number of nodes. We have tested the PARTHEO theorem proving system on a network of 16 transputers T 800 arranged in a hypercube topology, as displayed in Figure 4.

7 Related Work

In the following we want to mention very shortly some parallel systems[17], theorem provers or parallel Prolog systems, which have some similarities with PARTHEO. Most of these systems use or-parallelism and variants of the Warren Abstract Machine. In contrast to PARTHEO, however, all systems except the DelPhi system are using hardware with a common memory.

One of the basic models for or-parallel execution of Prolog and also model elimination based systems is the SRI-model developed by D.H.D. Warren [War88]. It uses shared memory and so-called *cactus stacks* to maintain multiple variable bindings which occur in or-parallelism. An implementation on 8 Balanc 32020's yields 23 Klips. A further refinement, the andorra model [War88] comprises also stream-dependent and-parallelism.

Parthenon [BCLS89] is an or-parallel theorem prover for first order logic based on model elimination. It uses a modification of the scheme proposed in the SRI-model. Parthenon is running on an Encore Multimax with 16 processors and shared memory. Like PARTHEO it uses a task-stealing model to distribute unexplored subtrees over the processors. Its peak performance is about 30 Klips with 15 processors (for detailed figures see [BCLS89]).

METEOR [Ast89] is a model elimination theorem prover for full first order logic which employs or-parallelism. It uses Warren's SRI model for or-parallel logic programming. It runs on a 32 node BBN Butterfly GP1000 under the Mach operating system.

Another parallel theorem prover was developed by V. Kumar et.al. [BSL+89]. It is based on the WAM and the SRI-model on a parallel machine with shared memory and employs and-parallelism within a producer-consumer approach.

The DelPhi inference machine [Clo87] uses multiple communicating processors with a shared memory for executing logic programs (Prolog and TABLOG). Different nodes in the and-or-tree (encoded as *oracles* similar to the tasks in our model) are distributed over the processors. The oracles are generated by one control processor in the system. A prototype has been implemented on a network of VAX and μVAX stations.

PEPSys is a parallel Prolog-system developed at the ECRC and comprises or-parallelism as well as independent and-parallelism [BKH+88]. The problem of maintaining multiple variable bindings (in or-parallelism) is solved using hash-tables. The

[17]Of course this list is by far not complete.

architecture used is similar to the SRI-model. PEPSys is implemented on a shared memory Siemens MX500 (equiv. to Sequent Balance 50) and simulations with a special multi-cluster architecture with up to 100 processing elements have been carried through.

8 Conclusions

PARTHEO is a system combining a number of promising features in the fields of theorem proving and logic programming. It integrates the model elimination calculus, abstract machine technology, and or-parallelism. PARTHEO offers the power and expressiveness of first order logic while retaining the performance of Prolog. First, because of the high efficiency of SETHEO, which runs on each node of the network. And secondly, because the or-parallel distribution of the search space may decrease the proving time tremendously.

Due to time limitations there was no possibility to change a number of interesting parameters, among which are the number of processors and the network topology. Notwithstanding that, we have identified a class of significant and reliable problems, for which a good speed-up has been achieved. It is at about 0.55 for the representative examples. This seems to us a very satisfactory result for the current unoptimized state of the system. The fact that the used processor type, the transputer T 800, is already very fast, makes this relative speed-up also a success in terms of absolute efficiency. For a certain class of examples (see [LBSB89]) we achieved a performance of 17 Klips on *one* transputer node. Moreover, the design decision to rely on processor-processor message passing and local memories, makes the system scalable, so that it may be extended without difficulties to networks with a large number of processors.

References

[3L88] *Parallel C – User Guide.* 3L Ltd., Livingston, Scotland, 1988.

[Ast89] Owen Astrachan. METEOR: Model elimination theorem prover for efficient OR–parallelism. In W.W. Bledsoe, M. Stickel, P. Lincoln, R. Overbeek, and D. Plaisted, editors, *1989 AAAI Spring Symposium on Representation and Compilation in High Performance Theorem Proving: Titles and Abstracts*, Stanford, CA, 3 1989.

[BCLS89] Soumitra Bose, Edmund M. Clarke, David E. Long, and Michaylov Spiro. Parthenon: A Parallel Theorem Prover for Non-Horn Clauses. In *LICS (to appear)*, 1989.

[BEK+89] S. Bayerl, W. Ertel, F. Kurfess, R. Letz, and J. Schumann. D16 / full first order logic parallel inference machine – language and design. ESPRIT 415, Deliverables, Brussels (to appear), 1989.

[Bet59] E.W. Beth. *The Foundations of Mathematics.* North–Holland, 1959.

[BEvdK+89] S. Bayerl, W. Ertel, M. v. d. Koelen, F. Kurfess, R. Letz, J. Schumann, Ch. Suttner, and N. Trapp. PARTHEO/6: PARallel Automated THEorem Prover based on the Connection Method for Full First Order Logic (Implementation and Performance). ESPRIT 415F Deliverable D15, 1989.

[Bib87] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, second edition, 1987.

[BKH+88] U. Baron, J. C. de Kergommeaux, M. Hailperin, M. Ratcliffe, M. Robert, J.-Cl. Syre, and H. Westphal. The parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *FGCS*, ECRC Munich, 1988. ECRC, Pepsys.

[BSL+89] W.W. Bledsoe, M. Stickel, P. Lincoln, R. Overbeek, and D. Plaisted, editors. *1989 AAAI Spring Symposium on Representation and Compilation in High Performance Theorem Proving: Titles and Abstracts*, Stanford, Ca USA, 3 1989.

[Clo87] W.F. Clocksin. Principles of the delPhi Parallel Inference Machine. *Comp. Journal*, 30(5):386–392, 5 1987.

[CM83] J. Corbin and Bidoit M. A Rehabilitation of Robinson's Unification Algorithm. In *Information Processing*, pages 909–914. North–Holland, 1983.

[Ede85] E. Eder. An Implementation of a Theorem Prover based on the Connection Method. In W. Bibel and B. Petkoff, editors, *AIMSA: Artificial Intelligence Methodology Systems Applications*, Varna, Bulgaria, 1985. North-Holland.

[JS89] Ph. Jorrand and Ph. Schnoebelen. Parallel implementation of connection method on an abstract FP2 machine. ESPRIT 415F Deliverable D17, 1989.

[Kor85] R.E. Korf. Depth–first Iterative Deepening: an Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[LBSB89] R. Letz, S. Bayerl, J. Schumann, and W. Bibel. SETHEO – A High–Performance Theorem Prover. (to appear in Journal of Automated Reasoning), 1989.

[Let88] R. Letz. Expressing First Order Logic within Horn Clause Logic. Technical report, ATP–Report, Technische Universität München, 1988.

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

[LS88] R. Letz and J. Schumann. Global Variables in Logic Programming. Technical report, ATP–Report, Technische Universität München, 1988.

[Pfe88] Frank Pfennig. Single Axioms in the Implicational Propositional Calculus. In *9th Int. Conf. on Automated Deduction (CADE 88)*, pages 710–713. Springer, 1988.

[Pla84] D. A. Plaisted. The Occur–check Problem in Prolog. *New Generation Computing*, 2:309–322, 1984.

[PR86] J. Pelletier and P. Rudnicki. Non–obviousness. *AAR Newsletter 6*, pages 4–5, 1986.

[Smu68] R.M. Smullyan. *First Order Logic*. Springer, 1968.

[Sti88] M. A. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[STv89] J. Schumann, N. Trapp, and M. van der Koelen. SETHEO: User's Manual. Technical report, ATP–Report, Technische Universität München, 1989.

[Tse70] G. S. Tseitin. On the Complexity of Derivations in the Propositional Calculus. In A. O. Slisenko, editor, *Studies in Mathematics and Mathematical Logic II*, pages 115–125, 1970.

[VH87] J. Vlahavas and C. Halatsis. A New Abstract Prolog Instruction Set. In *7th International Workshop of Expert Systems and Applications*, pages 1025–1050, Avignon, 1987.

[War83] D.H.D Warren. An Abstract PROLOG Instruction Set. Technical report, SRI, Menlo Park, CA, USA, 1983.

[War88] D.H.D. Warren. Parallel Execution Models and Architectures for Prolog. presented at Working Group Architect. ESPRIT 415, 1 1988.

[WM76] G.A. Wilson and J. Minker. Resolution, Refinements, and Search Strategies: a Comparative Study. *IEEE Transactions on Computers*, C–25:782–801, August 1976.