PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Model Generation Theorem Provers on a Parallel Inference Machine

Masayuki Fujita Ryuzo Hasegawa
Miyuki Koshimura* Hiroshi Fujita†

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
{mfujita, hasegawa, koshi}@icot.or.jp fujita@sys.crl.melco.co.jp

Abstract

This paper describes the results of the research and development on parallel theorem provers being conducted at ICOT. We have implemented a model-generation based parallel theorem prover called MGTP in KL1 on a distributed memory multi-processor, Multi-PSI, and on a parallel inference machine with the same architecture, PIM/m. Currently, we have two versions of MGTP: one is MGTP/G, which is used for dealing with ground models, and the other is MGTP/N, used for dealing with non-ground models. While conducting research and development on the MGTP provers, we have developed several techniques to improve the efficiency of forward reasoning theorem provers. These include model generation and hyper-resolution theorem provers. First, we developed KL1 compilation techniques to translate the given clauses to KL1 clauses, thereby achieving good efficiency. To avoid redundancy in conjunctive matching, we developed RAMS, MERC, and Δ-M methods. To reduce the amount of computation and space required for obtaining proofs, we proposed the idea of *Lazy Model Generation*. Lazy model generation is a new method that avoids the generation of unnecessary atoms that are irrelevant to obtaining proofs, and provides flexible control for the efficient use of resources in a parallel environment. For MGTP/G, we exploited OR parallelism with a simple allocation scheme, thereby achieving good performance on the Multi-PSI. For MGTP/N, we exploited AND parallelism, which is rather harder to obtain than OR parallelism. With the lazy model generation method, we have achieved a more than one-hundred-fold speedup on a PIM/m consisting of 128 PEs.

*Present address: Toshiba Information Systems
2-1 Nissin-cho, Kawasaki-ku, Kawasaki, Kanagawa 210, Japan
†Present address: Mitsubishi Electric Corporation
8-1-1 Tsukaguchi-honmachi, Amagasaki, Hyogo 661, Japan

1 Introduction

The research on parallel theorem proving systems has been conducted under the Fifth Research Laboratory at ICOT as a part of research and development on the problem-solving programming module. This research aims at the realization of highly parallel advanced inference mechanisms that are indispensable in building intelligent knowledge information systems.

The immediate goal of this research project is to develop a parallel automated reasoning system on the parallel inference machine, PIM, based on KL1 and PIMOS technology [Chikayama *et. al.* 88]. We aim at applying this system to various fields such as intelligent database systems, natural language processing, and automated programming.

The motive for the research is twofold.

From the viewpoint of logic programming, we try to further extend logic programming techniques that provide the foundation for the Fifth Generation Computer System. The research will help those aiming at extending languages and/or systems from Horn clause logic to full first-order logic. In addition, theorem proving is one of the most important applications that could effectively be built upon the logic programming systems. In particular, it is a good application for evaluating the abilities of KL1 and PIM.

From the viewpoint of automated reasoning, on the other hand, it seems that the logic programming community is ready to deal with more classical and difficult problems[Wos *et. al.* 84][Wos 88] that remain unsolved or have been abandoned. We might achieve a breakthrough in the automated reasoning field if we apply logic programming technology to theorem proving. In addition, this trial would also cause feedback for logic programming technology.

Recent developments in logic programming languages and machines have shed light upon the problem of how to implement these classical but powerful methods efficiently. For instance, Stickel developed a model-

elimination[Loveland 78] based theorem prover called PTTP[Stickel 88]. PTTP is able to deal with any first-order formula in Horn clause form (augmented by contrapositives) without loss of completeness or soundness. It works by employing unification with occurrence check, the model elimination reduction rule, and iterative deepening depth-first search. A parallel version of PTTP, called PARTHENON[Bose *et. al.* 89], has been implemented by Clarke et al. on a shared memory multiprocessor. Schumann et al. built a connection-method[Bibel 86] based theorem-proving system, SETHEO[Schumann 89], in which a method identical to model elimination is used as a main proof mechanism. Manthey and Bry presented a tableaux-like theorem prover, SATCHMO[Manthey and Bry 88], which is a very short and simple program in Prolog.

As a first step for developing KL1-technology theorem provers, we adopted the model generation method on which SATCHMO is based. Our reasons were as follows:

(1) A useful feature of SATCHMO is that full unification is not necessary, and that matching suffices when dealing with range-restricted problems. This makes it very convenient for us to implement provers in KL1 since KL1, as a committed choice language, provides us with very fast one-way unification.

(2) It is easier to incorporate mechanisms for lemmatization, subsumption tests, and other deletion strategies that are indispensable in solving difficult problems such as condensed detachment problems [Wos 88][Overbeek 90][McCune and Wos 91].

In implementing model generation based provers, it is important to avoid redundancy in the *conjunctive matching* of clauses against atoms in model candidates. For this, we proposed the RAMS [Fujita and Hasegawa 91] and MERC [Hasegawa 91a] methods.

A more important issue with regard to the efficiency of model generation based provers is how to reduce the total amount of computation and memory required for proof processes. This problem becomes more critical if we try to solve harder problems that require deeper inferences (longer proofs) such as Lukasiewicz problems. To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes and that generation should be performed only when testing requires it. We proposed the *Lazy Model Generation* method in which the idea of demand-driven computation or 'generate-only-at-test' is implemented. Lazy model generation is a new method that avoids the generation of unnecessary atoms that are irrelevant to obtaining proofs, and provides flexible control for the efficient use of resources in a parallel environment.

We have implemented two types of model generation prover: one is used for ground models (MGTP/G) and the other is used for non-ground models (MGTP/N).

In implementing MGTP/G, we developed a compiling technique to translate the given clauses into KL1 clauses by using advantage (1) listed above. This makes MGTP/G very simple and efficient. MGTP/G can prove non-Horn problems very efficiently on a distributed memory multi-processor, the Multi-PSI, by exploiting OR parallelism.

MGTP/N, on the other hand, aims at proving difficult Horn problems by exploiting AND parallelism. For MGTP/N, we developed new parallel algorithms based on lazy model generation method. They run with optimal load balancing on a distributed memory architecture, and require a minimal amount of computation and memory to obtain proofs.

In the next section, we explain the model generation method on which our MGTP provers are based. In Section 3, we discuss the problem of meta-programming in KL1, and outline the characteristics of MGTP/G and MGTP/N. In Section 4, we describe the essence of the main techniques developed for improving the efficiency of model generation theorem provers. In Section 5, we present OR parallelization and AND parallelization methods developed for MGTP/G and MGTP/N. Section 6 provides a conclusion.

2 Model Generation Theorem Prover

Throughout this paper, a clause is represented in an implicational form:

$$A_1, A_2, \ldots, A_n \rightarrow C_1; C_2; \ldots; C_m$$

where $A_i (1 \leq i \leq n)$ and $C_j (1 \leq j \leq m)$ are atoms; the antecedent is a conjunction of $A_1, A_2, \ldots, A_n$; the consequent is a disjunction of $C_1, C_2, \ldots, C_m$. A clause is said to be *positive* if its antecedent is $true (n = 0)$, and *negative* if its consequent is $false (m = 0)$. A clause is also said to be *tester* if its consequent is $false (m = 0)$, otherwise it is called *generator*.

The model generation method incorporates the following two rules:

- Model extension rule: If there is a generator clause, $A \rightarrow C$, and a substitution σ such that $A\sigma$ is satisfied in a model candidate M and $C\sigma$ is not satisfied in M, then extend M by adding $C\sigma$ into M.
- Model rejection rule: If a tester clause has an antecedent $A\sigma$ that is satisfied in a model candidate M, then reject M.

We call the process of obtaining $A\sigma$ a *conjunctive matching* of the antecedent literals against elements in a model. Note that the antecedent ($true$) of a positive clause is satisfied by any model.

The task of model generation is to try to construct a model for a given set of clauses, starting with a null set as a model candidate. If the clause set is satisfiable, a model should be found. The method can also be used to prove that the clause set is unsatisfiable, by exploring every possible model candidate to see that no model exists for the clause set.

For example, consider the following set of clauses, S1[Manthey and Bry 88]:

$$
\begin{aligned}
C1: &\quad p(X), s(X) \rightarrow false.\\
C2: &\quad q(X), s(Y) \rightarrow false.\\
C3: &\quad q(X) \rightarrow s(f(X)).\\
C4: &\quad r(X) \rightarrow s(X).\\
C5: &\quad p(X) \rightarrow q(X); r(X).\\
C6: &\quad true \rightarrow p(a); q(b).
\end{aligned}
$$

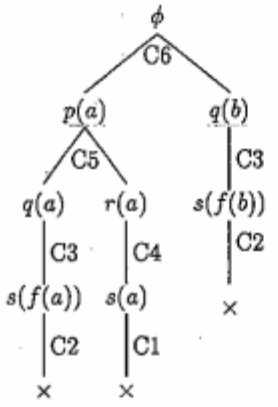


Figure 1: A proof tree for S1

A proof tree for the S1 problem is depicted in Fig. 1. We start with an empty model, $M_0 = \phi$. M_0 is first expanded into two cases, $M_1 = \{p(a)\}$ and $M_2 = \{q(b)\}$, by applying the model extension rule to $C6$. Then M_1 is expanded by $C5$ into two cases: $M_3 = \{p(a), q(a)\}$ and $M_4 = \{p(a), r(a)\}$. M_3 is further extended by $C3$ to $M_5 = \{p(a), q(a), s(f(a))\}$. Now with M_5 the model rejection rule is applicable to $C2$, thus M_5 is rejected and marked as closed. On the other hand, M_4 is extended by $C4$ to $M_6 = \{p(a), r(a), s(a)\}$ which is rejected by $C1$. In a similar way, the remaining model candidate M_2 is extended by $C3$ to $M_7 = \{q(b), s(f(b))\}$, which is rejected by $C2$. Now that there is no way to construct any model candidate, we can conclude that the clause set S1 is unsatisfiable.

The model generation method, as its name suggests, is closely related to the model elimination method. However, the model generation method is a restricted version of the model elimination method in the sense that the polarity of literals in a clause of implicational form is fixed to either positive or negative in the model generation method, whereas it is allowed to be both positive and negative in the model elimination method. Moreover, from the procedural point of view, model generation is restricted to proceeding bottom-up (as in forward-reasoning) starting at positive clauses (or facts). These restrictions, however, do not hurt the refutation completeness of the method.

Model generation can also be viewed as *unit hyper-resolution*. Our calculus, however, is much closer to tableaux calculus in the sense that it explores a tree, or a tableau, in the course of finding a proof. Indeed, a branch in a proof tree obtained by the tableaux method corresponds exactly to a model candidate.

3 Two Versions of MGTP

3.1 Meta-programming in KL1

Prolog-Technology Theorem Provers such as PTTP and SATCHMO utilize the fact that Horn clause problems can be solved very efficiently. In these systems, the theorem being proven is represented by Prolog clauses, and most deductions are performed as normal Prolog execution. However, that approach cannot be taken in KL1 because a KL1 clause is not just a Horn clause; it has extra-logical constructs such as a guard and a commit operator.

We should, therefore, treat the clause set as data rather than as a KL1 program. In this case, the inevitable problem is how to represent variables appearing in a given clause set. Two approaches can be considered for this problem:

(1) representing object-level variables with KL1 ground terms, or

(2) representing object-level variables with KL1 variables.

The first approach might be the right path in meta-programming, where object- and meta-levels are strictly separated, thereby providing clear semantics. However, it forces us to write routines for unification, substitution, renaming, and all the other intricate operations on variables and environments. These routines would become extremely large and complex compared to the main program, and would make the overhead bigger.

In the second approach, most operations on variables and environments can be performed beside the underlying system, rather than as routines running on top of it. This means that a meta-programmer does not have to write tedious routines, and gains high efficiency.

Also, a programmer can use the Prolog var predicate to write routines such as occurrence checks in order to

make built-in unification sound, if such routines are necessary. This approach makes the program much more simple and efficient, even though it makes the distinction between object- and meta-levels ambiguous.

In KL1, however, the second approach is not always possible. This is because the semantics of KL1 never allow us to use a predicate like var. In addition, KL1 built-in unification is not the same as its Prolog counterpart in that unification in the guard part of a KL1 clause can only be one-way, and a unification failure in the body part is regarded as a program error or exception that cannot be backtracked.

3.2 Characteristics of MGTP/G and MGTP/N

Taking the above discussions into consideration, we decided to develop both the MGTP/G and MGTP/N provers so that we can use effectively them according to the problem domains dealt with.

The ground version, MGTP/G, aims to support finite problem domains, which include most problems in various fields, such as database processing and natural language processing.

For ground model cases, the model generation method makes it possible to use just matching, rather than full unification, if the problem clauses satisfy the *range-restrictedness* condition [1] [Manthey and Bry 88]. This suggests that it is sufficient to use KL1's head unification. Thus we can take the KL1 variable approach for representing object-level variables, thereby achieving good performance.

The key points of KL1 programming techniques developed for MGTP/G are as follows: (Details are described in the next section.)

- First, we translate a given set of clauses into a corresponding set of KL1 clauses. This translation is quite simple.
- Second, we perform conjunctive matching of a literal in a clause against a model element by using KL1 head unification.
- Third, at the head unification, we can automatically obtain fresh variables for a different instance of the literal used.

The non-ground version, MGTP/N, supports infinite problem domains. Typical examples are mathematical theorems, such as group theory and implicational calculus.

[1] A clause is said to be range-restricted if every variable in the clause has at least one occurrence in its antecedent. For example, in the S1 problem, all the clauses, $C1$-$C6$, are range-restricted since no variable appears in clause $C6$; the variable X in clauses $C1, C3, C4$ and $C5$ has an occurrence in their antecedents; and variables X and Y in $C2$ have their occurrences in its antecedent.

```
c(1,p(X),□,    R):-true|R=cont.
c(1,s(X),[p(X)],R):-true|R=false.
c(2,q(X),□,    R):-true|R=cont.
c(2,s(Y),[q(X)],R):-true|R=false.
c(3,q(X),□,    R):-true|R=[s(f(X))].
c(4,r(X),□,    R):-true|R=[s(X)].
c(5,p(X),□,    R):-true|R=[q(X),r(X)].
c(6,true,□,    R):-true|R=[p(a),q(b)].
otherwise.
c(_,_,_,R):-true|R=fail.
```

Figure 2: S1 problem transformed to KL1 clauses

For non-ground model cases, where full unification with occurrence check is required, we are forced to follow the KL1 ground terms approach. However, we do not necessarily have to maintain variable-binding pairs as processes in KL1. We can maintain them by using the vector facility supported by KL1, as is often used in ordinary language processing systems. Experimental results show that vector implementation is several hundred times faster than process implementation.

In this case, however, we cannot use the programming techniques developed for MGTP/G. Instead, we have to use a conventional technique, that is, interpreting a given set of clauses instead of compiling it into KL1 clauses.

To ease the programmer's burden, we developed *Meta-Library*[Koshimura *et. al.* 90]. This is a collection of KL1 programs to support meta-programming in KL1. The meta-library includes facilities such as full unification with occurrence check, variable management routines, and term memory[Stickel 89][Hasegawa 91c].

4 Technologies Developed for Efficiency

4.1 KL1 Compiling Method

This section presents the compiling techniques developed for MGTP/G to translate given clauses to KL1 clauses. It also shows a simple MGTP/G interpreter obtained by using the techniques[Fuchi 90][Fujita and Hasegawa 90][Hasegawa *et. al.* 90a].

4.1.1 Transforming problem clauses to KL1 clauses

Our MGTP/G prover program consists of two parts: an interpreter written in KL1, and a set of KL1 clauses representing a set of clauses for the given problem. During conjunctive matching, an antecedent literal expressed in the head of a KL1 clause is matched against a model element chosen from a model candidate which is retained in the interpreter.

Although conjunctive matching can be implemented simply in KL1, we need a programming trick for support-

ing variables shared among literals in a problem clause. The trick is to propagate the binding for a shared variable from one literal to another.

To understand this, consider the previous example, S1. The original clause set is transformed into a set of KL1 clauses, as shown in Figure 2. In `c(N,P,S,R)`, `N` indicates clause number; `P` is an antecedent literal to be matched against an element taken from a model candidate; `S` is a pattern for receiving from the interpreter a stack of literal instances appearing to the left of `P`, which have already matched model elements; and `R` is the result returned to the interpreter when the match succeeds.

Notice that original clause $C1$ $(p(X), s(X) \rightarrow false.)$ is translated to the first two KL1 clauses. The conjunctive matching for $C1$ proceeds as follows. First, the interpreter picks up a model element, E_1, from a model candidate, and tries to match the first literal $p(X)$ in C1 against E_1 by initiating a goal, $c(1, E_1, [], R_1)$. If the matching fails, then the result $R_1 = fail$ is returned by the last KL1 clause. If the matching succeeds, then the result $R_1 = cont$ is returned by the first KL1 clause and the interpreter proceeds to the next literal $s(X)$ in C1, picking up another model element, E_2, from the model candidate and initiating a goal, $c(1, E_2, [E_1], R_2)$. Since the literal instance in the third argument, $[E_1]$, is ground, the variable X in $[p(X)]$ in the head of the second KL1 clause gets instantiated to a ground term. At the same time, the term $s(X)$ in that head is also instantiated due to the shared variable X. Under this instantiation, $s(X)$ is checked to see whether it matches E_2, and if the matching succeeds then the result, $R_2 = false$, is returned.

4.1.2 A simple MGTP/G interpreter

With the problem clauses are transformed to KL1 clauses as above, a simple interpreter is developed as shown in Figure 3[2].

The interpreter, given a list of numbers identifying problem clauses and a model candidate, checks whether the clauses are satisfiable or not. The top-level predicate, `clauses/5`, dispatches a task, `ante/7`, to check whether each clause is satisfied or not in the current model. If all the clauses are satisfied in the current model, the result, `sat`, is returned by `sat/4` combining the results from the `ante` processes.

For each clause in the given clauses, conjunctive matching is performed between the elements in the model candidate and the literals in the antecedent of the clause with `ante/7` and `ante1/9` processes. The conjunctive matching for the antecedent literals proceeds from left to right, by calling `c/4` one by one. An `ante` process retains a stack, S, of literal instances. If the match succeeds at a literal, L_i, with a model element, P, then P is pushed onto the stack S, and the task proceeds to matching the next literal, L_{i+1}, together with the stack, $[P|S]$.

According to the result of `c/4`: `fail`, `cont`, `false` or `list(F)`, an `ante1/9` process determines what to do next. If the result is `cont`, for example, `ante1` will fork multiple `ante` processes to try to make every possible combination of elements out of the current model for the conjunctive matching.

If the conjunctive matching for all the antecedent literals of a clause succeeds, a `cnsq/6` process is called to check the satisfiability of the consequent of the clause. `cnsq1/8` checks whether a literal in the consequent is a member of the current model. If no literal in the consequent is a member of the current model, the current model cannot satisfy the clause. In this case, the model will be extended with each disjunct literal in the consequent of the clause by calling an `extend/5` process.

After extending the current model, a `clauses/5` process is called for each extension of the model, and the results are combined by `unsat/4`. When a `clauses` process for some of the extended models returns `sat` as the result, it means that a model is found and the clause set is known to be satisfiable. If every extension of the model leads to `unsat`, the current model is not a part of any model for the given set of clauses.

Thus, if the top-level `clauses/5` process returns `sat` as the result, then the given clause set has a model and is satisfiable, and if it returns `unsat`, then the given clause set has no model and is unsatisfiable.

4.2 Avoiding Redundant Conjunctive Matching

To improve the performance of the model generation provers, it is essential to avoid, as much as possible, redundant computation in conjunctive matching.

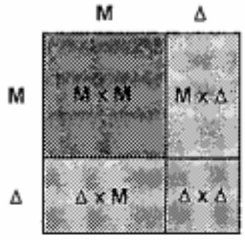
Let us consider a clause, C, having two antecedent literals. To perform conjunctive matching for the clause, we need to pick a pair of atoms out of the current model candidate, M. Imagine that, as a result of a satisfiability check of the clause, we are to extend the model candidate with Δ, which is an atom in the consequent of the clause, C, but not in M. Then, in the conjunctive matching for the clause, C, in the next phase, we need to pick a pair of atoms from $M \cup \Delta$. The number of pairs amounts to:

$$(M \cup \Delta)^2 = M \times M \cup M \times \Delta \cup \Delta \times M \cup \Delta \times \Delta.$$

[2] In the program, '`alternatively`' is a KL1 compiler directive which gives a preference among clauses to evaluate their guards in such a way that clauses above `alternatively` are evaluated before those below it. The preference, however, may not be strictly obeyed. This depends on implementation.

```
clauses(_,_,_,_,quit):-true|true.
alternatively.
clauses([J|Cs],C,M,A,B):-true|
    ante(J,[true|M],[],C,M,A1,B),
    sat(A1,A2,A,B), clauses(Cs,C,M,A2,B).
clauses([],_,_,A,_):-true|A=sat.

ante(_,_,_,_,_,_,quit):-true|true.
alternatively.
ante(J,[P|M2],S,C,M,A,B):-true|
    mgtp:c(J,P,S,R),
    ante1(J,R,P,S,M2,C,M,A,B).
ante(_,[],_,_,_,A,_):-true|A=sat.

ante1(J,fail,_,S,M2,C,M,A,B):-true|
    ante(J,M2,S,C,M,A,B).
ante1(J,cont,P,S,M2,C,M,A,B):-true|
    ante(J,M,[P|S],C,M,A1,B),
    sat(A1,A2,A,B), ante(J,M2,S,C,M,A2,B).
ante1(_,false,_,_,_,_,M,A,B):-true|
    A=unsat,B=quit.
ante1(J,F,_,S,M2,C,M,A,B):-list(F)|
    cnsq(F,F,C,M,A1,B),
    sat(A1,A2,A,B), ante(J,M2,S,C,M,A2,B).

cnsq(_,_,_,_,_,_,quit):-true|true.
alternatively.
cnsq([D1|Ds],F,C,M,A,B):-true|
    cnsq1(D1,M,Ds,F,C,M,A,B).
cnsq([],F,C,M,A,_):-true|
    extend(F,M,C,A,_).

cnsq1(D,[D|_],_,_,_,_,A,_):-true|A=sat.
cnsq1(_,[],Ds,F,C,M,A,B):-true|
    cnsq(Ds,F,C,M,A,B).
otherwise.
cnsq1(D,[_|M2],Ds,F,C,M,A,B):-true|
    cnsq1(D,M2,Ds,F,C,M,A,B).

extend(_,_,_,_,quit):-true|true.
alternatively.
extend([D|Ds],M,C,A,B):-true|
    clauses(C,C,[D|M],A1,_),
    unsat(A1,A2,A,B), extend(Ds,M,C,A2,B).
extend([],_,_,A,_):-true|A=unsat.

sat(sat,sat,A,_):-true|A=sat.
sat(unsat,_,A,B):-true|A=unsat,B=quit.
sat(_,unsat,A,B):-true|A=unsat,B=quit.

unsat(unsat,unsat,A,_):-true|A=unsat.
unsat(sat,_,A,B):-true|A=sat,B=quit.
unsat(_,sat,A,B):-true|A=sat,B=quit.
```

Figure 3: A simple MGTP/G interpreter



It should be noted here that $M \times M$ pairs were already considered in the previous phase of conjunctive matching. If they were chosen in this phase, the result would contribute nothing since the model candidate need not be extended with the same Δ. Hence, redundant consideration on $M \times M$ pairs should be avoided at this time. Instead, we have to choose only the pairs which contain at least one Δ. This discussion can be generalized for cases in which we have more than two antecedent literals, any number of clauses, and any number of model candidates.

We have taken two approaches to avoid the above redundancy. One approach uses a stack to keep the intermediate results obtained by matching a literal against an element out of the model candidate. The other approach recomputes the intermediate matching results without keeping them.

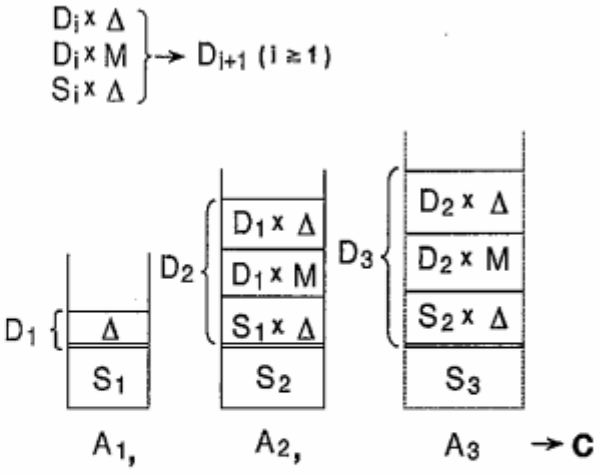


Figure 4: RAMS method

4.2.1 RAMS Method

The RAMS (ramified-stack) method [Hasegawa *et. al.* 90a][Hasegawa *et. al.* 90b][Fujita and Hasegawa 91] retains in a stack an instance which is a result of matching a literal against a model element. The use of this method

for a Horn clause case is illustrated in Figure 4, where M is a model candidate and Δ is an atom picked from a model-extending candidate.

- A stack called *a literal instance stack* (LIS), is assigned to each antecedent literal, A_i, in a clause for storing literal instances. Note that LIS for the last literal expressed in dashed boxes needs not actually be allocated.
- LIS is divided into two parts: D_i and S_i where $D_i (i \geq 1)$ is a set of literal instances generated at the current stage triggered by Δ; and S_i is those created in previous stages.
- A task, being performed at each literal, A_i, computes the following:

$$D_1 := \Delta;$$

$$D_{i+1} := D_i \times \Delta \cup D_i \times M \cup S_i \times \Delta \ (i \geq 1)$$

where $A \times B$ denotes a set of pairs of an instance taken from A and B. The above tasks are performed from left to right.

For non-Horn clause cases, each LIS branches to make a tree-structured stack when case splitting occurs. The name 'RAMS' comes from this. The idea is as follows:

- A model is represented by a branch of a ramified stack, and the model is extended only at the top of the current stack.
- After applying the model extension rule to a non-Horn clause, the current model may be extended to multiple descendant models.
- Every descendant model that is extended from a parent model can share its ancestors with other sibling models just by pointing to the top of the stack corresponding to the parent.
- Each descendant model can extend the stack for itself, independent of other sibling models.

The ramified-stack method not only avoids redundancy in conjunctive matching but also enables us to share a common model. However, it has one drawback: it tends to require a lot of memory to retain intermediate literal instances.

4.2.2 MERC Method

The MERC (Multi-Entry Repeated Combination) method [Hasegawa 91a] tries to solve the above problem using the RAMS method. This method does not need a memory to retain intermediate results obtained in the conjunctive matching. Instead, it needs to prepare $2^n - 1$ clauses for the given clause having n literals as its antecedent.

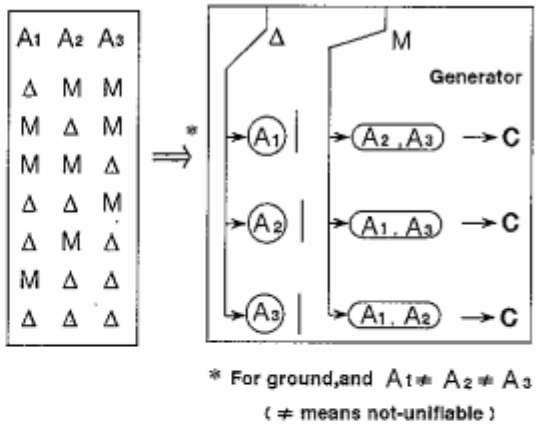


Figure 5: MERC method

An outline of the MERC method is shown in Figure 5. For a clause having three antecedent literals, $A_1, A_2, A_3 \rightarrow C$, we prepare seven clauses. Each of these clauses corresponds to a repeated combination of Δ and M, and performs conjunctive matching using the combination pattern. For example, a clause corresponding to a combination pattern $[M, \Delta, M]$ first matches literal A_2 against Δ. If the match succeeds, it proceeds to match the remaining literals, A_1 and A_3, against an element picked from M. Note that each combination pattern includes at least one Δ, and that the $[M, M, M]$ pattern is excluded.

For ground model cases, optimization can be used to reduce the number of clauses by testing the unifiability of antecedent literals. For example, if any antecedent literal in the given clause is not unifiable with the other antecedent literal in that clause, it is sufficient to consider the following three combination patterns: $[\Delta, M, M]$,$[M, \Delta, M]$ and $[M, M, \Delta]$. The right-hand side in Figure 5 shows the clauses obtained after making the unifiability test.

4.2.3 Δ-M Method

The problem with the MERC method is that the number of prepared clauses increases exponentially as the number of antecedent literals increases. In actual implementation, we adopted a modified version of the MERC method, which we call the Δ-M method. In place of multiple entry clauses, the Δ-M method prepares a template like:

$$\{[\Delta, \Delta], [\Delta, M], [M, \Delta]\}$$

for clauses with two antecedent literals, and

$$\{[\Delta, \Delta, \Delta], [\Delta, \Delta, M], [\Delta, M, \Delta], [M, \Delta, \Delta],$$

$$[\Delta, M, M], [M, \Delta, M], [M, M, \Delta]\}$$

for clauses with three antecedent literals, and so forth. According to this pattern, we enumerate all possible combinations of atoms for matching the antecedent literals of given clauses.

There are some trade-offs between the RAMS method and the MERC and Δ-M methods. In the RAMS method, every successful result of matching a literal A_i against model elements is memorized so that the same literal is not rematched against the same model element. On the other hand, both the MERC and Δ-M methods do not need to memorized information on partial matching. However, they still contain a redundant computation. For instance, in the computation for $[M, \Delta, \Delta]$ and $[M, \Delta, M]$ patterns, the common subpattern $[M, \Delta]$, will be recomputed. The RAMS method can eliminate this sort of redundancy.

4.3 Lazy Model Generation

Model-generation based provers must perform the following three operations.

- create new model elements by applying the model extension rule to the given clauses using a set of model-extending atoms Δ and a model candidate set M (model extension).
- make a subsumption test for a created atom to check if it is subsumed by the set of atoms already being created, usually by the current model candidate.
- make a false check to see if the unsubsumed model element derives false by applying the model extension rule to the tester clauses (rejection test).

The problem with the model generation method is the huge growth in the number of generated atoms and in the computational cost in time and space, which is incurred by the generation processes.

To solve this problem, it is important to recognize that proving processes are viewed as *generation-and-test* processes, and that generation should be performed only when testing requires it.

For this we proposed a lazy model generation algorithm [Hasegawa 91b][Hasegawa 91d][Hasegawa *et. al.* 92a][Hasegawa *et. al.* 92b] that can reduce the amount of computation and space necessary for obtaining proofs.

This section presents several algorithms, including the lazy algorithm, for the model generation method, and compares them in terms of time and space. To simplify the presentation, we assume that the problem is given only in Horn clauses. However, the principle behind these algorithms can be applicable to non-Horn clauses as well.

4.3.1 Basic Algorithm

The basic algorithm shown in Figure 6 performs model generation with a search strategy in a breadth-first fashion. This is essentially the same algorithm as the hyper-resolution algorithm taken by OTTER [McCune 90] [3].

```
M := φ;
D := {A | (true → A) ∈ a set of given clauses};
while D ≠ φ do begin
      D := D − Δ;
      if CJM_Tester(Δ, M) ∋ false
            then return(success);
      new := CJM_Generator(Δ, M);
      M := M ∪ Δ;
      new' := subsumption(new, M ∪ D);
      D := D ∪ new';
end return(fail)
```

Figure 6: Basic algorithm

In the algorithm, M represents model candidate , D represents the model-extending candidate (a set of model-extending atoms which are generated as a result of the application of the model extension rule and are going to be added to M), and Δ represents a subset of D. Initially, M is set to an empty set, and D is a set of positive (unit) clauses of the given problem.

In each cycle of the algorithm,

1) Δ is selected from D,
2) a rejection test (conjunctive matching for the tester clauses) is performed on Δ and M,
3) if the test succeeds then the algorithm terminates,
4) if the test fails then model extension (conjunctive matching on the generator clauses) is performed on Δ and M, and
5) a subsumption test is performed on *new* against $M \cup D$.

If D is empty at the beginning of a cycle, then the algorithm terminates as the refutation fails (In other words, a model is found for the given set of clauses).

The conjunctive matching and subsumption test is represented by the following functions on sets of atoms.

$$CJM_{Cs}(\Delta, M) =$$
$$\{\sigma C \mid \sigma A_1, \ldots, \sigma A_n \rightarrow \sigma C$$
$$\wedge\ A_1, \ldots, A_n \rightarrow C \in Cs$$
$$\wedge\ \sigma A_i = \sigma B (B \in M \cup \Delta)(1 \leq \forall i \leq n)$$
$$\wedge\ \exists i (1 \leq i \leq n) \sigma A_i = \sigma B (B \in \Delta)\}$$

$$subsumption(\Delta, M) =$$
$$\{C \in \Delta \mid \forall B \in M (B \text{ dosen't subsume } C)\}$$

[3] OTTER is a slightly optimized version of the basic algorithm where negative unit clauses are tested on literals in *new* as soon as they are generated as the full-test algorithm described in the next section.

$M := \phi$;
$D := \{A \mid (true \rightarrow A) \in$ a set of given clauses$\}$;
while $D \neq \phi$ do begin
    $D := D - \Delta$;
    $new := CJM_{Generator}(\Delta, M)$;
    $M := M \cup \Delta$;
    $new' := subsumption(new, M \cup D)$;
    if $CJM_{Tester}(new', M \cup D) \ni false$
        then return(success);
    $D := D \cup new'$;
end return(fail)

Figure 7: Full-test algorithm

4.3.2 Full-Test Algorithm

Figure 7 shows a refined version of the basic algorithm called the full-test algorithm. The algorithm 1) selects Δ from D, 2) performs model extension using Δ and M generating new for the next generation of Δ, 3) performs a subsumption test on new against $M \cup D$, and 4) performs a rejection test on new', which passed the subsumption test, together with $M \cup D$.

Though this refinement seems to be very small on the text level, the complexity of time and space is significantly reduced, as explained later. The points are as follows. The algorithm performs subsumption and rejection tests on all elements of new rather than on Δ, a subset of new generated in the past cycles. As a result, if a falsifying atom [4], X, is included in new, the algorithm can terminate as soon as $false$ is derived from X. That is, the algorithm neither overlooks the falsifying atom nor puts it into D as the basic algorithm does. Thus, it never generates atoms which are superfluous after X is found.

4.3.3 Lazy Algorithm

Figure 8 shows another refinement of the basic algorithm, the lazy algorithm. In this algorithm, it is assumed that two processes, one for generator clauses and the other for tester clauses, run in parallel and communicate with each other.

The tester process 1) requests Δ to the generator process, 2) performs a subsumption test on Δ against $M \cup D$, and 3) performs a rejection test on Δ.

For the generator process,

1) if a buffer, Buf, used for storing a set of atoms which are the results of an application of the model extension rule, is empty, the generator selects an atom, e, from D and sets a code for model extension (delay CJM) for e and M onto Buf,

2) waits for a request of Δ from the tester process, and

[4] A falsifying atom, X, is an atom that satisfies the antecedent of a negative clause by itself or in combination with $M \cup D$.

process tester:
    repeat forever
        $request(generator, \Delta)$;
        $\Delta' := subsumption(\Delta, M \cup D)$;
        if $CJM_T(\Delta', M \cup D) \in false$
            then return(success);
        $D := D \cup \Delta'$.

process generator:
    repeat forever
        while $Buf = \phi$ do begin
            $D := D - \{e\}$;
            Buf :=delay$CJM_G(\{e\}, M)$;
            $M := M \cup \{e\}$ end;
        $wait(tester)$;
        Δ :=forceBuf;
    until $D = \phi$ and $Buf = \phi$.

Figure 8: Lazy algorithm

3) forces the buffer, Buf, to generate Δ.

delay (above) is an operator which delays the execution of its operand (a function call). Hence, the function call, $CJM_G(\{e\}, M)$, will not be activated during 1), but will be stored in Buf as a code. Later, at 3), when the force operator is applied to Buf, the delayed function call is activated. This generates the values that are demanded. Using this mechanism, it is possible to generate only the Δ that is demanded by the tester process. After the required amount of Δ is generated, a delayed function call for generating the rest of the atoms is put into Buf as a continuation.

The atoms are stored in M and D in a way that makes the order of generating and testing the atoms exactly the same as in the basic algorithm. The point of the refinement in the lazy algorithm is, therefore, to equalize the speed of generation and testing while keeping the order of atoms that are generated and tested the same as that of the basic algorithm. This eliminates any excess consumption of time and space due to over-generation of redundant atoms.

4.4 Optimization of Unit Tester Clauses

Given the unit tester clauses in the problem, the three algorithms above can be further optimized. There are two ways to do this.

One is a dynamic way called the lookahead method. In this method, atoms are generated excessively in the generation process in order to apply the rejection rule with unit tester clauses. More precisely, immediately after generating new, the generator process generates new_{next}, which would be regenerated in a succeeding step. Then

new_{next} is tested with unit tester clauses. If the test fails, then new_{next} is discarded whereas new is stored.

$$< \Delta, M > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new$$

$$< new, M \cup D > \Rightarrow generate(A_1, A_2 \rightarrow C) \Rightarrow new_{next}$$

$$new_{next} \Rightarrow test(A \rightarrow false)$$

The reason why new_{next} is not stored is that testing with unit tester clauses does not require M or D, but can be done with only new_{next} itself. On the other hand, for tester clauses with more than one literal, testing cannot be completed, since testing for combinations of atoms from new_{next} would not be performed.

new_{next} will be regenerated as new in the succeeding step. This means that some conjunctive matching will be performed twice for the identical combination of atoms in a model candidate. However, the increase in computational cost due to this redundancy is negligible compared to the order of total computational cost.

The other method is a static one which uses partial evaluation. This is used to obtain non-unit tester clauses from a unit tester clause and a set of generator clauses by resolving the two.

$$Generator: \ A_1, A_2 \rightarrow C.$$

$$Unit\ tester: \ A \rightarrow false.$$

$$\Downarrow$$

$$Non\text{-}unit\ tester: \ \sigma A_1, \sigma A_2 \rightarrow false.$$

$$where\ \sigma C = \sigma A$$

The computational complexity for conjunctive matching using the partial evaluation method is exactly the same as that using the lookahead method. The partial evaluation method, however, is simpler than the lookahead method, since the former does not need any modification of the prover itself whereas the latter does. Moreover, the partial evaluation method may be able to reduce the search space significantly, since it can provide propagating goal information to generater clauses. However, in general, partial evaluation results in an increase in the number of clauses, Hence it may make performance worse.

The two optimization techniques are equally effective, and will optimize the model generation algorithms to the same order of magnitude when they are applied to unit tester clauses.

4.4.1 Summary of Complexity Analysis

In this section, we briefly describe the time and space complexity of the algorithms described above. The details are discussed in [Hasegawa *et. al.* 92a]. For simplicity, we assumed the following.

1) The problem consists of generator clauses with two antecedent literals and one consequent literal, and tester clauses with at most two literals.

2) Δ is a singleton set of an atom selected from D.

3) The rate at which conjunctive matchings succeed for a generator clause, and atoms generated as the result pass a subsumption test, the survival rate, is $\rho (0 \leq \rho \leq 1)$.

4) The order in which Δ is selected and atoms are generated according to Δ is fixed for all of the three algorithms.

Table 1 summarizes the complexity analysis. T/S/G stands for complexity entry of rejection test /subsumption test/model extension, and M stands for the required memory space. The value of $\alpha (1 \leq \alpha \leq 2)$ represents the efficiency factor of the subsumption test. $\alpha = 1$ means that a subsumption test is performed in a constant order, because the hashing effect is perfect. $\alpha = 2$ means that a subsumption test is performed in a time proportional to the number of elements, perhaps because a linear search was made in the list. As for the condensed detachment problem, the hashing effect is very poor and α is very close to two.

The memory space required for the basic, full-test/lazy and lazy lookahead algorithms decreases along this order by a square root for each. This means that the number of atoms generated decreases as the algorithm changes, which in turn implies that the number of subsumption tests decreases accordingly. In the case of $\alpha = 2$, the most expensive computation of all is a subsumption test, and a decrease in its complexity means a decrease in total complexity. On the other hand, in the case of $\alpha = 1$, the most expensive computation of all is the rejection test with two-literal tester clauses. This situation, however, is the same for all of the algorithms and adopting lazy computation will result in speedup by a constant factor. In any case, by adopting lazy computation, the complexity of the total computation is dominated by that of the rejection test .

4.4.2 Performance Experiment

An experimental result is shown in Table 2. The example, Theorem 4, is taken from [Overbeek 90]. We did not use heuristics such as weighting and sorting, but only limited term size and eliminated tautologies.

Every algorithm is implemented in KL1 and run on a pseudo Multi-PSI in PSI-II [Nakashima and Nakajima 87]. The OTTER entry represents the basic algorithm optimized for unit tester clauses and implemented in KL1. The figures in parentheses are of algorithms for tester clauses with two literals as a result of applying partial evaluation to unit tester clauses. In unify entries,

Table 2: Experimental result (Theorem 4)

		basic	full-test	lazy	lazy lookahead	OTTER
Time (sec)		>14000 (463.86)	409.17 (82.40)	407.58 (81.82)	210.45 (81.69)	409.16 (462.13)
Unify		— (43981+74254)	1656+74800 (43981+4158)	1656+74737 (43981+4158)	81956+4095 (43981+4095)	1656+74800 (43981+74254)
Subsumption test		— (5674)	5736 (596)	5736 (596)	593 (593)	5736 (5674)
Memory	M	— (272)	272 (63)	272 (63)	63 (63)	272 (272)
	D	— (1375)	1384 (209)	1384 (209)	209 (209)	1384 (1375)

Table 1: Summary of complexity analysis

	Unit tester clause			
	T	S	G	M
basic	ρm^2	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	ρm^2	$\mu m^{2\alpha}$	m^2	ρm^2
lazy lookahead	m^2	$(\mu/\rho)m^\alpha$	m/ρ	m

	2-literal tester clause			
	T	S	G	M
basic	$\rho^2 m^4$	$\mu\rho^2 m^{4\alpha}$	$\rho^2 m^4$	$\rho^3 m^4$
full-test / lazy	$\rho^2 m^4$	$\mu m^{2\alpha}$	m^2	ρm^2

† m is the number of elements in model candidate when *false* is detected in the basic algorithm.
‡ ρ is the survival rate of a generated atom, μ is the rate of successful conjunctive matchings ($\rho \leq \mu$), and α is the efficiency factor of a subsumption test.

a figure to the left of + represents the number of conjunctive matchings performed in tester clauses, and a figure to the right of + represents the number of conjunctive matchings performed in generator clauses.

These results are a fair reflection of the complexity analysis shown in Table 1. For instance, to solve Theorem 4 without partial evaluation optimization, the basic algorithm did not reach a goal within 14,000 seconds, whereas the full-test and lazy algorithms reached the goal in about 400 seconds. The most time-consuming computation in all of the three algorithms (basic, full-test and lazy), is rejection testing. The difference in the time complexity between the basic algorithm and the other two algorithms is $(\mu\rho^2 m^{4\alpha})/(\mu m^{2\alpha}) = \rho^2 m^{2\alpha}$, which results in the time difference mentioned above.

The basic algorithm and the full-test/lazy algorithm do not differ in the number of unifications performed in the tester clauses. However, the number of unifications performed in the generator clauses and the number of subsumption tests decreases as we move from the basic algorithm to the full-test and lazy algorithms. The decrease is about one hundredth when partial evaluation is not applied, and about one tenth when it is applied.

By applying lookahead optimization, the lazy algorithm is further improved. Though the lookahead optimization and the partial evaluation optimization are theoretically comparable in their order of improvement, their actual performance is sometimes very different. For Theorem 4, the lazy algorithm optimized with partial evaluation took 81.82 seconds, whereas the same algorithm optimized with lookahead optimization took 210.45 seconds. This difference is caused by the difference in the number of unifications performed in the tester clauses. This is because in the lazy algorithms with lookahead optimization, the generator clause, $p(X), p(\epsilon(X, Y)) \rightarrow p(Y)$, generates an atom before the unit tester clause, $p(A) \rightarrow false$ tests the atom. In the same algorithm, with the partial evaluation optimization, the instantiation information of A is propagated to the antecedent of $p(X), p(\epsilon(X, A)) \rightarrow false$ and the unification failure can be detected earlier.

Partial evaluation optimization is effective for all the algorithms except OTTER. This is because lookahead optimization, in the OTTER algorithm, is already applied to unit tester clauses, and the algorithm remains the basic one for non-unit tester clauses.

5 Parallelizing MGTP

There are several ways to parallelize the proving process in the MGTP prover.

These are to exploit parallelism in:

- conjunctive matching in the antecedent part,
- subsumption test, and
- case splitting

For ground non-Horn cases, it is sufficient to exploit OR parallelism induced by case splitting. Here we use

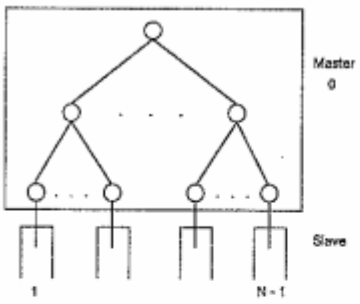


Figure 9: Simple allocation scheme

OR parallelism to seek a multiple model, which produces multiple solutions in parallel.

For Horn clause cases, we have to exploit AND parallelism. The main source of AND parallelism is conjunctive matching. Performing subsumption tests in parallel is also very effective for Horn clause cases.

In the current MGTP, we have not yet considered non-ground and non-Horn cases.

5.1 OR Parallelization for MGTP/G

With the current version of the MGTP/G, we have only attempted to exploit OR parallelism[Fujita and Hasegawa 90] on the Multi-PSI machine[Nakajima *et. al.* 89].

5.1.1 Processor Allocation

The processor allocation methods we have adopted achieve 'bounded-OR' parallelism in the sense that OR-parallel forking in the proving process is suppressed so as to meet restricted resource circumstances.

One simple way of doing this, called *simple allocation*, is depicted in Figure 9. We expanded model candidates, starting with an empty model, using a single master-processor until the number of candidates exceeded the number of available processors. We then distributed the remaining tasks to slave-processors. Each slave processor explored the branches assigned without further distributing tasks to any other processors. This simple allocation scheme for task distribution works fairly well, since the communication cost can be minimized.

5.1.2 Performance of MGTP/G on Multi-PSI

One of the examples we used was the N-queens problem. This problem can be expressed by the following clause set:

Table 3: Performance of MGTP/G on Multi-PSI

	Number of processors				
Problem	1	2	4	8	16
4-queens					
Time (msec)	40	40	39	44	44
Speedup	1.00	1.00	1.02	0.90	0.90
Kred	1.45	1.47	1.48	1.50	1.50
6-queens					
Time (msec)	650	407	266	189	154
Speedup	1.00	1.59	2.44	3.44	4.22
Kred	23.7	23.7	23.7	23.8	23.8
8-queens					
Time (msec)	12,538	6,425	3,336	1,815	1,005
Speedup	1.00	1.95	3.76	6.91	12.5
Kred	460	460	460	460	460
10-queens					
Time (msec)	315,498	159,881	79,921	40,852	21,820
Speedup	1.00	1.97	3.94	7.72	14.5
Kred	11,117	11,117	11,117	11,117	11,117

$$true \rightarrow p(1,1); p(1,2); \ldots; p(1,n).$$
$$true \rightarrow p(2,1); p(2,2); \ldots; p(2,n).$$
$$\ldots$$
$$true \rightarrow p(n,1); p(n,2); \ldots; p(n,n).$$
$$p(X_1,Y_1), p(X_2,Y_2), unsafe(X_1,Y_1,X_2,Y_2) \rightarrow false.$$

The first N clauses simply express every possibility of placing queens on the N by N chess board. The last clause expresses the constraint that a pair of queens must satisfy. The problem can be solved when either a model (one solution) or all of the models (all solutions)[5] are obtained for the clause set.

Performance was measured on the MGTP/G prover running on the Multi-PSI with the simple allocation method. Table 3 gives the result of the all-solution search on the N-queens problem. Here we should note that the total number of reductions stays almost constant, even though the number of processors used increases. This means that no extra computation is introduced by distributing tasks. Speedup obtained by using up to 16 processors is shown in Figures 10 and 11. For the 10-queens and 7-pigeons problems, the speedup obtained as the number of processors increases is almost linear. The speedup rate is small only for the 4-queens problem. This is probably because the constant amount of interpretation overhead in such a small problem will dominate the tasks required for the proving process.

[5]All models can be obtained, if they are finite, by the MGTP interpreter in all-solution mode.

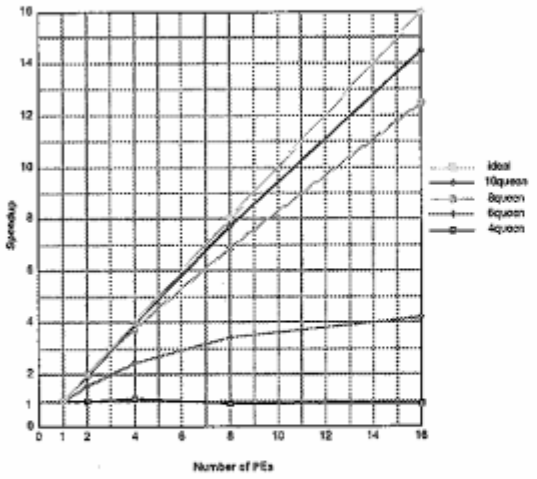


Figure 10: Speedup of MGTP/G on Multi-PSI (N-queens)

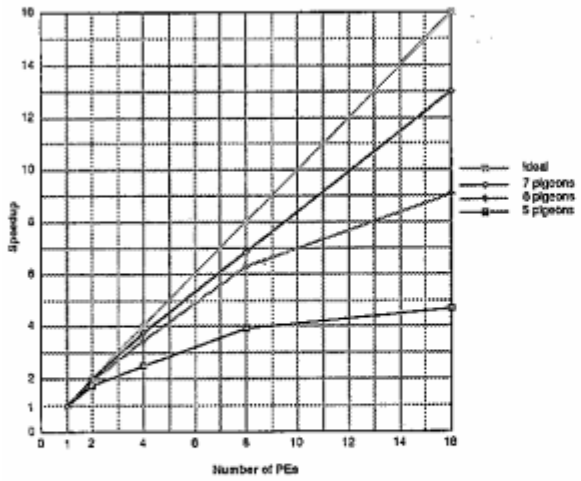


Figure 11: Speedup of MGTP/G on Multi-PSI (Pigeon hole)

5.2 AND Parallelization for MGTP/N

We have several choices when parallelizing model-generation based theorem provers:

1) proofs which change or remain unchanged according to the number of PEs used,

2) model sharing (copying in a distributed memory architecture) or model distribution, and

3) master-slave or masterless.

The proof obtained by a proof changing prover may be changed according to a change in the number of PEs. We might get super-linear speedup if the length of a proof depended on the number of PEs used. However, we cannot always expect an increase in speed as the number of PEs increases.

On the other hand, a proof unchanging prover does not change the length of the proof, no matter how many PEs we use. Hence, we could always expect greater speedup as the number of PEs increased, though we would only get linear speedup at best.

With model sharing, each PE has a copy of the model candidates and distributed model-extending candidates. With model distribution, both the model candidates and model-extending candidates are distributed to each PE.

Model sharing and model distribution both have advantages and disadvantages. From the distributive processing point of view, with model distribution, we can obtain memory scalability and more parallelism than with the model sharing method. For a newly created atom δ, there are n parallelisms in the model distribution method, since we can perform conjunctive matchings and subsumption tests for it in parallel where n is the number of processors. On the other hand, in the model sharing method, we cannot exploit this kind of parallelism for a single created atom unless conjunctive matchings and subsumption tests are made for a different region of model candidates.

From the communication point of view, however, the communication cost with model sharing is less than with model distribution. The communication cost with model distribution increases as the number of PEs increases, since generated atoms need to flow to all PEs for subsumption testing. For example, if the size of model elements finally obtained is M, the number of communications amounts to $O(M^2)$ for a clause having two antecedent literals. On the other hand, with model sharing, we do not have to flow the generated atoms to all PEs. In this case, time-consuming subsumption tests and conjunctive matchings can be performed independently at each PE, with minimal inter-PE communication.

The master-slave configuration makes it easy to build a parallel system by simply connecting a sequential version of MGTP/N on a slave PE to the master PE. However, its devices must be designed to minimize the load on

the master process. On the other hand, a masterless configuration such as ring connection allows us to achieve pipeline effects with better load balancing, whereas it becomes harder to implement suitable control to manage collaborative work among PEs.

Our policy in developing parallel theorem provers is that we should distinguish between the speedup effect caused by parallelization and the search-pruning effect caused by strategies. In proof changing parallelization, changing the number of PEs is merely betting, and may cause a strategy to be changed for the worse even if it results in the finding of a shorter proof.

In order to ensure the validity of our policy, we implemented proof changing and unchanging versions. In the following sections, we describe actual parallel implementations and compare them.

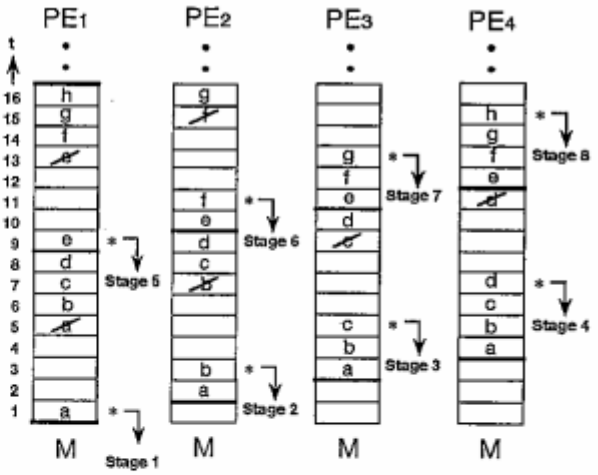


Figure 12: Proof Changing and Model Sharing

5.2.1 Proof Changing Implementation

1. **Model Sharing**
   This implementation uses model sharing, and a ring architecture in which $process_i (1 \leq i < n)$ is connected to $process_{i+1}$ and $process_n$ is connected to $process_1$, where n is the number of PEs[Hasegawa 91a].

   $process_i$ has a copy of model candidates M and distributed model-extending candidates D_i.

   A rough sketch of operations performed in $process_i (1 < i \leq n)$ follows.

   (1) Receive Δ_{i-1} from $process_{i-1}$.

   (2) Pick up an atom δ_i from D_i such that δ_i is not subsumed by any elements in M and Δ_{i-1}. $D_i := D_i - \{\delta_i\}$.

   (3) $\Delta_i := \Delta_{i-1} \cup \{\delta_i\}$.

   (4) If $CJM_{Tester}(\{\delta_i\}, M \cup \Delta_{i-1}) \ni false$ then send a termination message to all processes, otherwise,

   (5) $D_i := D_i \cup CJM_{Generator}(\{\delta_i\}, M \cup \Delta_{i-1})$.

   (6) $M := M \cup \Delta_i$ (update M in $process_i$).

   (7) Send Δ_i to $process_{i+1}$.

   For $process_1$, instead of actions (3) and (6), the following actions are performed.

   (3') $\Delta_1 := \{\delta_1\}$, and

   (6') $M := M \cup \Delta_n$.

   Note that actions (4)~(8) can be performed in parallel.

   Figure 12 shows how models are copied, and conjunctive matching is executed in a pipeline manner in the case of $n = 4$.

   A letter denotes a model candidate element and an asterisk indicates an element on which conjunctive matching is performed. For example, $process_1$ on PE_1 selects an unsubsumed model element a (from its own model-extending candidate) at time t_1, and sends it to $process_2$ on PE_2.

   $process_2$ stores element a into the model candidates in PE_2, proposes a model-extending element b, sends a and b to the $process_3$, and starts conjunctive matching of b and $\{a\} \cup M$.

   Note that conjunctive matching in a $process_i$ can be overlapped. For example, the conjunctive matching in stage 6 does not have to wait for the completion of the conjunctive matching in stage 2. This exploits pipeline effects very well, resulting in low communication cost compared to the computation cost for conjunctive matching.

2. **Model Distribution**
   This implementation takes model distribution and a ring architecture. Each process has its own distributed model candidates and distributed model-extending candidates. The algorithm for each process is similar to the sequential basic algorithm. They differ in that: 1) conjunctive matching cannot be completed in one process because model candidates are distributed. Thus the continuations of conjunctive matching in each process need to go around the ring, and 2) newly created atoms have to go around the ring for subsumption testing.

5.2.2 Proof Unchanging Implementation

We implemented a proof unchanging version in a master-slave configuration, and model sharing based on the lazy model generation. In this implementation, generator and subsumption processes run in a demand-driven mode,

while tester processes run in a data-driven mode. The main advantages of this implementation are as follows:

1) Proof unchanging allows us to obtain greater speedup as the number of PEs increases.

2) By utilizing the synchronization mechanism supported by KL1, sequentiality in subsumption testing is minimized.

3) Since slave processes spontaneously obtain tasks from the master, and the size of each task is well equalized, good load balancing is achieved.

4) By utilizing the KL1 stream data type, demand-driven control is easily and efficiently implemented.

By using demand-driven control, we cannot only suppress unnecessary model extensions and subsumption tests but also maintain a high running rate, which is the key to achieving linear speedup.

The model generation method consists of three tasks:

1) generation,

2) subsumption test, and

3) rejection test.

We provided three processes to cope with this:

- G(generator),
- S(subsumption tester), and
- T(rejection tester).

The $G/T/S$ process has a pointer $i/j/k$ which indicates an element of the stack, shown in Figure 13. The stack elements are model candidates or model-extending candidates. In the figure, M denotes model candidates for which conjunctive matching performed by G is completed and D denotes model-extending candidates on which the subsumption test is completed. $G/T/S$ process iterates the following actions.

G: performs model extensions by using the i-th element (Δ) and the $1,\ldots,i-1$ -th elements (M), and sends newly created atoms to S. $i := i+1$.

S: performs subsumption tests on the newly created atoms against $1,\ldots,k-1$ -th elements ($M \cup D$), and pushes the unsubsumed atoms to the stack. $k := k+l$ where l is the number of unsubsumed atoms.

T: performs model rejection tests on the j-th element and the $1,\ldots,j-1$ -th elements.

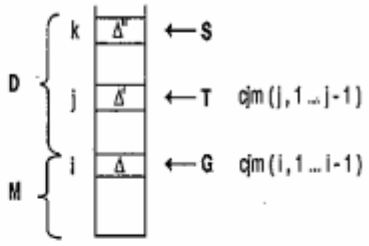


Figure 13: Lazy Implementation

Figure 14 shows a process structure for the proof unchanging parallel implementation. The central box represents the shared model and model-extending candidates.

The upper boxes represent atoms generated by the generator G_i and the arrows indicate the order in which the atoms are sent to the master process. Proof unchanging is realized by keeping this order. To make the system proof unchanging, the sequence order in which M and D are updated must remain the same as the sequence in a sequential case. The master process sends an atom generated by a generator process to a subsumption tester process in the same order as the master receives the atom, that is, the master aligns the elements generated by generator processes so as to be in the same order as in the sequential case.

Many $G/T/S$ processes work simultaneously. The master process is introduced to control task distribution, that is, giving a different task (Δ) to a different process. Each S process requests Δ'' to a G process through the master process. This means that the communication between G and S processes is indirect.

The critical resource for S processes is the model-extending candidates D. The critical regions are the updating of D by $D := D \cup new'$ and a part of $subsumption(new, M \cup D)$ (see Figure 8).

Most elements of $M \cup D$ have already been determined by some subsumption tester process and synchronization in subsumption testing can be minimized so that most parts of subsumption tests should not be critical.

To exclusively access the critical resource D, each S process requests to the master a pair of Δ'' and a key which indicates the right to update. If Δ'' is subsumed by the already determined elements in $M \cup D$, the key is returned to the master process without any reference to the key. In this case, there is no synchronization with other S processes. If Δ'' is not subsumed by the already determined elements in $M \cup D$, the S process refers to the key to see if it has the right to update, and updates D by $D := D \cup \Delta''$ if it has. Otherwise, the process waits until the other S process updates D. If the other S process updates D, the subsumption test is performed on the added elements.

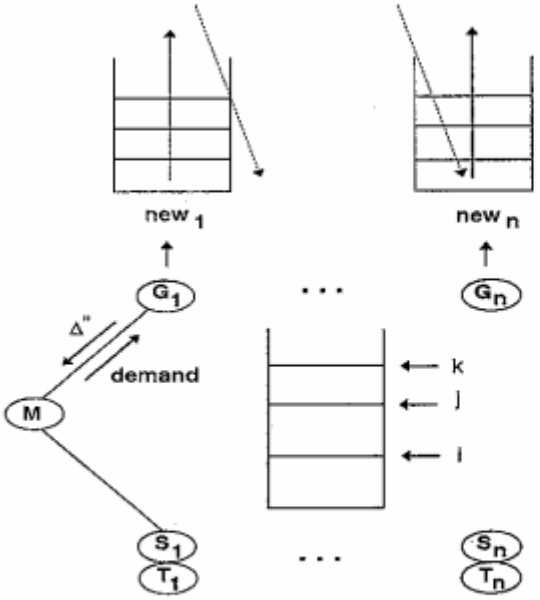


Figure 14: Proof Unchanging

The critical resources for the G processes are both the model candidates M and the model-extending candidates D. This is similar to tester processing.

5.2.3 Performance of MGTP/N on Multi-PSI and PIM

Some experimental results for the proof changing and unchanging versions in model sharing are shown in Tables 4 and 5, and Figures 15 and 16. Each program is implemented in KL1 and runs on the Multi-PSI.

Table 4 shows a performance comparison between the two versions with 16 PEs. In the proof unchanging version (PU column), we limited the term size and eliminated tautologies. In addition to the above, in the proof changing version (PC column), we used heuristics such as weighting and sorting. All problems are condensed detachment problems [McCune and Wos 91].

We measured performance with 1, 2, 4, 8 and 16 PEs. In the PC time entry column, the number of PEs in parentheses indicates the number of PEs which yield the best performance. In the proof unchanging version, we always got the best performance with 16 PEs, whereas we sometimes got the best performance with 8 PEs in the proof changing version. We also have an example in which we got the best performance with 2 PEs.

This comparison implies that super-linear speedup does not always signify an advantage in a parallelization method, because the proof unchanging version always beats the proof changing version in absolute speed with the problems used in the table.

Figures 15 and 16 display the speedup ratio for the problems #3, #58, #77, #66, #92, and #112 using the

Table 4: Performance Comparison (16PEs)

Problem		PU	PC
#3	Time (sec)	218.77	6766 (16 PEs)
	KRPS/PE	34.68	25.99
	Speedup	13.27	-
#6	Time (sec)	3.75	157.63 (16 PEs)
	KRPS/PE	12.47	17.75
	Speedup	3.65	6.75
#56	Time (sec)	3.53	10.37 (8 PEs)
	KRPS/PE	13.39	3.97
	Speedup	3.53	415.57
#58	Time (sec)	12.80	27.32 (16 PEs)
	KRPS/PE	27.51	3.75
	Speedup	9.23	66.32
#63	Time (sec)	4.56	48.37 (16 PEs)
	KRPS/PE	20.01	15.24
	Speedup	6.06	11.07
#69	Time (sec)	6.07	23.41 (16 PEs)
	KRPS/PE	16.69	4.52
	Speedup	4.98	2.90
#72	Time (sec)	3.62	12.17 (16 PEs)
	KRPS/PE	14.02	2.10
	Speedup	4.47	45.51
#77	Time (sec)	37.10	62.07 (8 PEs)
	KRPS/PE	36.66	25.62
	Speedup	12.65	109.24

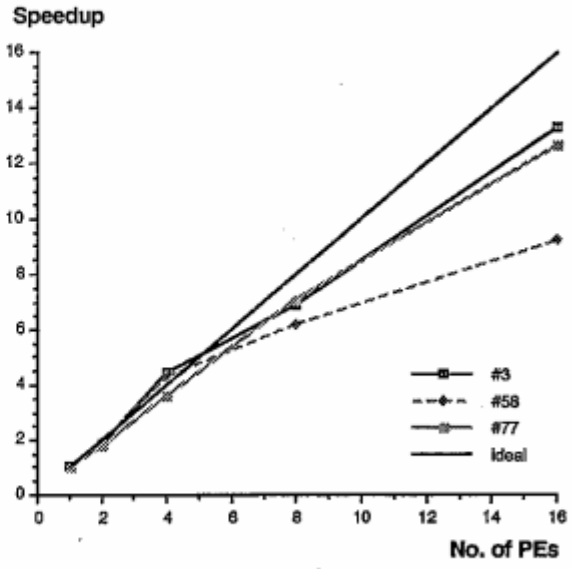


Figure 15: Speedup ratio I

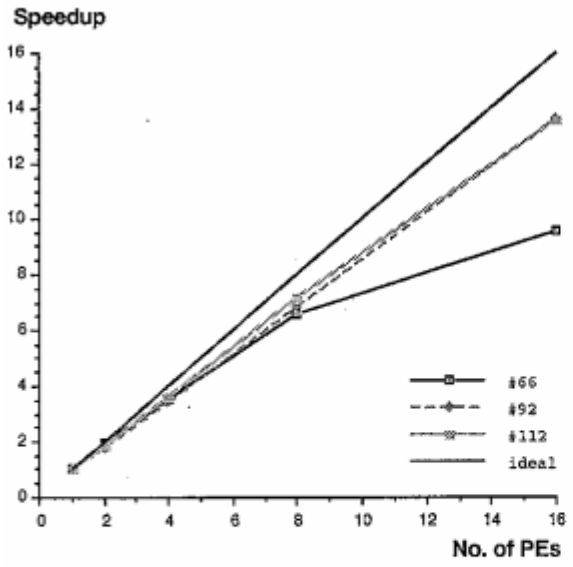


Figure 16: Speedup ratio II

Table 5: Performance for 16/64 PEs

Problem		16 PEs	64 PEs
Th 5	Time (sec)	41725.98	11056.12
	Reductions	38070940558	40759689419
	KRPS/PE	57.03	57.60
	Speedup	1.00	3.77
Th 7	Time (sec)	48629.93	13514.47
	Reductions	31281211417	37407531427
	KRPS/PE	40.20	43.25
	Speedup	1.00	3.60

proof unchanging version. There is no saturation in performance up to 16 PEs and greater speedup is obtained for the problems which consume more time.

Table 5 shows the performance obtained by running the proof unchanging version for Theorems 5 and 7 [Overbeek 90] on Multi-PSI with 64 PEs. We did not use heuristics such as sorting, but merely limited term size and eliminated tautologies. Note that the average running rate per PE for 64 PEs is actually a little higher than that for 16 PEs. With this and other results, we were able to obtain almost linear speedup.

Recently we obtained a proof of Theorem 5 on PIM/m [Nakashima *et. al.* 92] with 127 PEs in 2870.62 sec and nearly 44 billion reductions[6] (thus 120 KRPS/PE). Taking into account the fact that the PIM/m CPU is about twice as fast as the Multi-PSI CPU, we found that near-linear speedup can be achieved, at least up to 128 PEs.

[6]The exact figure was 43,939,240,329 reductions

6 Conclusion

We have presented two versions of the model-generation theorem prover MGTP implemented in KL1: MGTP/G for ground models and MGTP/N for non-ground models. We evaluated their performance on the distributed memory multi-processors Multi-PSI and PIM.

When dealing with range-restricted problems in model-generation theorem provers, we only need matching rather than full unification, and can make full use of the language features of KL1, thereby achieving good efficiency.

The key techniques for implementing MGTP/G in KL1 are as follows:

(1) A given set of input clauses of implicational form are compiled into a corresponding set of KL1 clauses.

(2) Generated models are held by the prover program instead of being asserted.

(3) Conjunctive matching of the antecedent literals of an input clause against a model element is performed by very fast KL1 head unification.

(4) Searching for a model element that matches the antecedent is performed by computing a repeated combination of model elements by means of loop executions instead of backtracking.

(5) Fresh variables for a different instance of the antecedent literal are obtained automatically just by calling a KL1 clause.

These techniques are very simple and straightforward yet effective.

For solving non-range-restricted problems, however, we cannot use the above techniques developed for MGTP/G. If the given problem is Horn, it can be solved by the MGTP prover extended by incorporating unification with occurrence check, without changing the basic structure of the prover. For non-Horn problems, however, substantial changes in the structure of the prover would be required in order to manage shared variables appearing in the consequent literals of a clause. Accordingly, we restricted MGTP/N to Horn problems, and developed a set of KL1 meta-programming tools called the Meta-Library to support full unification and the other functions for variable management.

To improve the efficiency of the MGTP provers, we developed RAMS, MERC, and Δ-M methods that enable us to avoid redundant computations in conjunctive matching. We have obtained good performance results by using these methods on the PSI.

Moreover, it is important to avoid very great increases in the amount of time and space consumed when proving hard theorems which require deep inferences. For this we proposed the lazy model generation method, which

can decrease the time and space complexity of the basic algorithm by orders of magnitude. Experimental results show that significant amounts of computation and memory can be saved by using the lazy algorithm.

The parallelization of MGTP is one of the most important issues in our research project.

For non-Horn ground problems, a lot of OR parallelism caused by case splitting can be expected. This kind of problem is well-suited to a local memory multi-processor such as Multi-PSI, on which it is necessary to make the granuality as large as possible so that communication costs can be minimized. We obtained an almost linear speedup for the n-queens, pigeon hole, and other problems on Multi-PSI, using a simple allocation scheme for task distribution.

For Horn problems, on the other hand, we had to exploit the AND parallelism inherent in conjunctive matching and subsumption. Though the parallelism is large enough, it seemed rather harder to exploit than OR parallelism, since the Multi-PSI is not suited to this kind of fine-grained parallelism. Nevertheless, we found that we could obtain good performance and scalability by using the AND parallelization methods mentioned in this paper.

In particular, the recent results obtained by running the MGTP/N prover on PIM/m showed that we could achieve linear speedup for condensed detachment problems, at least up to 128 PEs. The key technique is the lazy model generation method, that avoids the unnecessary computation and use of memory space while maintaining a high running rate.

For MGTP/N, full unification is written in KL1, which is thirty to one hundred times slower than that written in C on SUN/3s and SPARCs. To further improve the performance of MGTP/N, we need to incorporate built-in firmware functions for supporting full unification, or to develop KL1 compiling techniques for non-ground models.

Through the development of MGTP provers, we confirmed that KL1 is a powerful tool for the rapid prototyping of concurrent systems, and that parallel automated reasoning systems can be easily and effectively built on the parallel inference machine, PIM.

Acknowledgment

We would like to thank Dr. Kazuhiro Fuchi, the director of ICOT, and Dr. Koichi Furukawa, the deputy director of ICOT, for giving us the opportunity to do this research and for their helpful comments. Many fruitful discussions took place at the PTP Working Group meeting. Thanks are also due to Prof. Fumio Mizoguchi of the Science University of Tokyo, who chaired PTP-WG, and many people at the cooperating manufacturers in charge of the joint research.

References

[Bibel 86] W. Bibel, *Automated Theorem Proving,* Vieweg, 1986.

[Bose *et. al.* 89] S. Bose, E. M. Clarke, D. E. Long and S. Michaylov, PARTHENON: A Parallel Theorem Prover for Non-Horn Clauses in *Proc. of 4th Annual Symp. on Logic in Computer Science*, 1989.

[Chikayama *et. al.* 88] T. Chikayama, H. Sato and T. Miyazaki, Overview of the Parallel Inference Machine Operating System (PIMOS), in *Proc. of FGCS'88*, 1988.

[Fuchi 90] K. Fuchi, Impression on KL1 programming – from my experience with writing parallel provers –, in *Proc. of KL1 Programming Workshop '90,* pp.131-139, 1990 (in Japanese).

[Fujita and Hasegawa 90] H. Fujita and R. Hasegawa, Implementing A Parallel Theorem Prover in KL1, in *Proc. of KL1 Programming Workshop '90,* pp.140-149, 1990 (in Japanese).

[Fujita *et. al.* 90] H. Fujita, M. Koshimura, T. Kawamura, M. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1, *Joint US-Japan Workshop,* 1990.

[Fujita and Hasegawa 91] H. Fujita and R. Hasegawa, A Model-Generation Theorem Prover in KL1 Using Ramified Stack Algorithm, In *Proc. of the Eighth International Conference on Logic Programming,* The MIT Press, 1991.

[Hasegawa *et. al.* 90a] R. Hasegawa, H. Fujita and M. Fujita, A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis, *Italy-Japan-Sweden Workshop*, ICOT-TR-588, 1990.

[Hasegawa *et. al.* 90b] R. Hasegawa, T. Kawamura, M. Fujita, H. Fujita and M. Koshimura, MGTP: A Hyper-Matching Model-Generation Theorem Prover with Ramified Stacks, *Joint UK-Japan Workshop,* 1990.

[Hasegawa 91a] R. Hasegawa, A Parallel Model Generation Theorem Prover: MGTP and Further Research Plan, In Proc. of the Joint American-Japanese Workshop on Theorem Proving, Argonne, Illinois, 1991.

[Hasegawa 91b]
R. Hasegawa, A Parallel Model-Generation Theorem Prover in KL1, Workshop on Parallel Processing for AI,IJCAI'91,1991.

[Hasegawa 91c] R. Hasegawa, A Parallel Model Generation Theorem Prover with Ramified Term-Indexing, Joint France-Japan Workshop,Rennes,1991.

[Hasegawa 91d] R. Hasegawa, A Lazy Model-Generation Theorem Prover and Its Parallelization, Joint Germany-Japan Workshop on Theorem Proving, GMD,Bonn,1991.

[Hasegawa *et. al.* 92a] R. Hasegawa, M. Koshimura and H. Fujita, Lazy Model Generation for Improving the Efficiency of Forward Reasoning Theorem Provers, ICOT-TR-751, 1992.

[Hasegawa *et. al.* 92b] R. Hasegawa, M. Koshimura and H. Fujita, MGTP: A Parallel Theorem Prover Based on Lazy Model Generation, To appear in *Proc. of CADE 92 (System Abstract)*, 1992.

[Koshimura *et. al.* 90] M. Koshimura, H. Fujita and R. Hasegawa, Meta-Programming in KL1, ICOT-TR-623, 1990 (in Japanese).

[Loveland 78] D. W. Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, 1978.

[Manthey and Bry 88] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, In *Proc. of CADE 88, Argonne, Illinois*, 1988.

[McCune 90] W. W. McCune, OTTER 2.0 Users Guide, Argonne National Laboratory, 1990.

[McCune and Wos 91] W. W. McCune and L. Wos, Experiments in Automated Deduction with Condensed Detachment, Argonne National Laboratory, 1991.

[Nakajima *et. al.* 89] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa and T. Chikayama, Distributed Implementation of KL1 on the Multi-PSI/V2, in *Proc. of 6th ICLP*, 1989.

[Nakashima and Nakajima 87] H. Nakashima and K. Nakajima, Hardware architecture of the sequential inference machine PSI-II, In *Proc. of 1987 Symposium on Logic Programming*, Computer Society Press of the IEEE,1987.

[Nakashima *et. al.* 92] H. Nakashima, K. Nakajima, S. Kondoh, Y. Takeda and K. Masuda, Architecture and Implementation of PIM/m, In *Proc. of FGCS'92*, 1992.

[Overbeek 90] R. Overbeek, Challenge Problems, (private communication) 1990.

[Slaney and Lusk 91] J. K. Slaney and E. L. Lusk, Parallelizing the Closure Computation in Automated Deduction, In *Proc. of CADE 90*, 1990.

[Schumann 89] J. Schumann, SETHEO: User's Manual, Technische Universität München, 1989.

[Stickel 88] M. E. Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, In *Journal of Automated Reasoning*, 4:353-380, 1988.

[Stickel 89] M. E. Stickel, The Path-indexing method for indexing terms, Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1989.

[Wos *et. al.* 84] L. Wos, R. Overbeek, E. Lusk and J. Boyle, *Automated Reasoning: Introduction and Applications*, Prentice-Hall, 1984.

[Wos 88] L. Wos, *Automated Reasoning – 33 Basic Research Problems –*, Prentice-Hall, 1988.