PARTHENON:
A Parallel Theorem Prover for Non-Horn Clauses

Soumitra Bose Edmund M. Clarke David E. Long Spiro Michaylov

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Abstract

We describe a parallel resolution theorem prover, called *Parthenon*, that handles full first order logic. Although there has been much work on parallel implementations of logic programming languages, Parthenon is apparently the first general purpose theorem prover to be developed for a multiprocessor. The system is based on a modification of Warren's SRI model for or-parallelism and implements a variant of Loveland's model elimination procedure. It has been evaluated on various shared memory multiprocessors including a 16-processor Encore Multimax. We have found that typical theorem proving problems exhibit a great deal of potential parallelism. Parthenon has been able to exploit much of this parallelism, producing both impressive absolute run times and near-linear speedup curves.

1 Introduction

Parthenon is a parallel resolution theorem prover that has been implemented on several shared memory multiprocessors, including a 16 processor Encore Multimax. It handles arbitrary first order formulas in clause form. A major theme of the Parthenon project has been the use of implementation techniques originally developed for or-parallel versions of Prolog. Although there has been much research on parallel implementations of logic programming languages, Parthenon is the first general theorem prover based on these techniques. Our experience in solving a large number of well known problems shows that a high degree of parallelism is available in these problems. In particular, the branching factors that we observe are typically much higher than those of Prolog programs. On many examples Parthenon has exhibited linear speedup with respect to the number of processors used. This paper describes the implementation of Parthenon and gives experimental results to support our conclusions.

The first major decision of our project was the selection of the inference mechanism for the theorem prover. We chose to use the *model elimination* procedure of Loveland [5] because it is readily amenable to the use of Prolog implementation techniques, which we consider important for achieving high inference rates. Stickel's sequential Prolog Technology Theorem Prover [8] also uses model elimination and has heavily influenced our work. Although we expect the reader to be familiar with the ideas of resolution theorem proving, we do not assume any previous knowledge of the model elimination procedure.

The model elimination procedure is inherently parallel, since alternative branches of the search tree can be explored independently. This method of exploring the tree is called *or-parallel search*. The straightforward way of implementing this type of search would be to allocate a separate processor for each node in the search tree. However, since the number of processors is always limited, this scheme cannot be used in practice. Instead, Parthenon uses a modification of the scheme proposed by D. H. D. Warren in his SRI model for or-parallelism [10]. Although it was originally developed for Horn clause logic programming, we show that it can be also be modified to handle general first order clauses with the model elimination procedure. In our adaptation of Warren's scheme, each processor performs a depth-first search of a subtree of the model elimination proof tree. When a processor has completely explored its current subtree, it must find another subtree to explore. This may involve *stealing* an unexplored subtree from some other processor. Our procedure for finding new work is simpler and potentially more efficient than the schemes originally proposed by Warren. We give a detailed description of the procedure and of how to handle the extension and reduction operations of the model elimination procedure within the framework of the SRI model. Additionally, we describe modifications to the model which take advantage of the high branching factors found in a theorem proving context. We believe that our experience with the SRI model is of interest to researchers in logic programming as well as in automatic theorem proving.

The Encore Multimax that we use has 16 processors and 32 megabytes of shared memory. Each processor is a National Semiconductor 32332 and is rated at roughly 2 MIPS. The Multimax is suitable for medium and coarse grained parallel applications [7]. Parthenon is implemented in C and uses the *C-Threads* package [3], which allows parallel programming under the MACH operating system [1]. Interlocks are used for process synchronization instead of general semaphores in order to avoid the expense associated with system calls. Contention for access to shared memory is light, so the time required for an individual lock operation is a few tens of microseconds. Since each model elimination inference takes much longer, the overhead for synchronization is not excessive. In the near future, we plan to run the system on several larger multiprocessors, including IBM's RP3 and larger versions of the Encore.

We have tested our implementation on a large number of ex-

This research was partially supported by NSF grant CCR-87-226-33. The third author holds an NSF graduate fellowship and the fourth author holds an IBM graduate fellowship.

CH2753-2/89/0000/0080$01.00 © 1989 IEEE

amples originally collected by Stickel. Our experimental results are discussed in detail in a later section of the paper. Although the two systems are perhaps not directly comparable, on some examples Parthenon is an order of magnitude faster than the sequential Prolog Technology Theorem Prover. On almost all examples we observe significant speedup as the number of processors is increased. One reason for this speedup is the large branching factor in many of the examples. We conjecture, in fact, that typical theorems will have a much larger branching factor than is observed in logic programs. As a result, we believe that or-parallel theorem proving is likely to be feasible regardless of whether this turns out to be the case for Prolog programs. Moreover, our results lead us to believe that such problems have sufficient parallelism to make effective use of the much larger scale machines currently being developed.

The paper is organized as follows: Section 2 describes the model elimination procedure. Section 3 discusses various implementations of or-parallelism on shared memory multiprocessors. An overview of the system is given in Section 4, and the following two sections describe key procedures in greater detail. Section 5 describes our parallel implementation of the model elimination procedure. We show how the additional operations of the model elimination procedure can be implemented within the same basic framework that is used for logic programming languages. Section 6 focuses on the algorithm that a processor uses to find work when the subtree that it has been searching is completely explored. This algorithm is the heart of our system, and we outline a proof that the procedure is correct. Section 7 discusses the performance of Parthenon on a number of standard examples collected by Stickel. The paper concludes in Section 8 with some observations on the role of parallelism in automatic theorem proving and some directions for future research.

2 The Model Elimination Procedure

The model elimination proof procedure was first introduced by D. Loveland [5] in 1968. Our implementation is based on a simpler format for the procedure that was also developed by Loveland [6]. There have been several successful sequential implementations of the procedure. The most recent was the Prolog Technology Theorem Prover developed by Stickel [8]. By using techniques originally developed for sequential implementations of Prolog, Stickel was able to obtain very high inference rates. Our own project has been heavily influenced by Stickel's work. In this section we describe the model elimination procedure and indicate why it is useful in first order theorem proving. We will assume that the reader is familiar with the basic definitions of resolution theorem proving as described in [2] or [4].

To motivate our discussion of the model elimination procedure we first consider a simple example from propositional logic:

$$\frac{\begin{array}{c} N \vee P \\ \neg P \vee N \\ \neg N \vee P \end{array}}{N \wedge P}$$

To obtain a resolution proof that the conclusion follows from the three premises, we negate the conclusion and add it to the list of premises.

1. $N \vee P$
2. $N \vee \neg P$
3. $\neg N \vee P$
4. $\neg N \vee \neg P$

These four clauses are called the *input clauses*. Note that the first clause is not a Horn clause since two literals occur positively. In this simple example we can easily use the resolution rule to derive the empty clause and obtain a refutation proof.

5.	$\neg N$	resolving 4 and 3
6.	P	resolving 5 and 1
7.	N	resolving 6 and 2
8.	$\square$ (the empty clause)	resolving 7 and 5

In each step but the last, one of the two clauses being resolved was from the original set of clauses. However, in the last step we resolved two clauses that were previously obtained by resolution. In fact, any resolution proof for this example must resolve two resolvents. To see why this is true, first observe that resolving any two clauses of the original set will result in a single literal clause involving N or P. Next, observe that resolving any such single literal clause with one of the original clauses with give another single literal clause. The possibility of having to resolve two resolvents is a major disadvantage of the resolution procedure. In logic programming where all of the clauses are Horn clauses, this problem does not arise. One clause in each resolution step will be an input clause and the branching factor in the search tree will be bounded by the number of input clauses. A proof procedure for first order formulas in clause form that has this property is called an *input procedure*. The above example shows that simple input resolution is not complete.

A second disadvantage of conventional resolution is the need for *factoring*. Consider the set of clauses $C' = \{P(x) \vee P(y). \neg P(x) \vee \neg P(y)\}$. If conventional resolution is applied to C', it is impossible to obtain a clause with fewer than two literals even though C' is unsatisfiable. We can obtain the empty clause in two steps, however, if we collapse the first clause to $P(x)$ by the substitution y/x. This procedure is called factoring, and $P(x)$ is said to be a *factor* of $P(x) \vee P(y)$. Factoring is so basic in conventional resolution theorem proving that it is often combined in a single step with the binary resolution operation. Unfortunately, when it is not needed for completeness, factoring can result in a significant increase in the branching factor of the search.

The model elimination procedure is an input procedure that is complete without the need for factoring. It uses a type of generalized clause called a *chain*. A chain is an ordered sequence of two types of literals: *framed* literals and *unframed* literals. Unframed literals behave very much like the literals in conventional resolution theorem proving. We will indicate that the literal L is framed by enclosing it in a box (e.g. $\boxed{L}$). We will say that two literals (framed or unframed) *match* if they have opposite signs and if there is a substitution that makes

their atoms identical. The *matching substitution* will be the most general unifier of the two atoms.

The procedure begins in much the same way as conventional input resolution. Suppose that we want to test the set of clauses $\mathcal{S}$ for satisfiability. We pick some clause C in $\mathcal{S}$ to start the procedure. In a refutation proof of some theorem B from the axioms $A_1, \ldots, A_n$, C will usually be obtained from the clause form of the negation of the conclusion B. We will regard C as a chain with all literals unframed and call it the *center chain*. There are three types of operations on chains: *extension*, *reduction*, and *contraction*. The first operation takes as input a chain and some *side clause* in $\mathcal{S}$ and produces another chain. The other two operations are used to simplify chains—each takes a chain as input and produces a chain as output. The procedure terminates successfully if the empty chain is obtained.

The *extension* rule is like the standard resolution operation applied to the rightmost literal of the center clause and a matching literal of some side clause in $\mathcal{S}$. Instead of discarding the last literal in the center clause, it is converted to a framed literal. Any clause in $\mathcal{S}$, including C itself, can act as a side clause in this step provided that it contains a literal that matches the rightmost literal in the center chain. More formally, the extension operation takes as input a chain C_1 and a side clause C_2 and produces a chain C_3 provided that there exists a matching substitution θ for the rightmost literal in C_1 and some literal of C_2/η, where η is a substitution that renames the variables of C_2 so that it has no variables in common with C_1. To form C_3, the rightmost literal of $C_1\theta$ becomes framed. The matching literal in $C_2/\eta\theta$ is deleted and the remainder of $C_2/\eta\theta$ is attached to the right of $C_1\theta$. For example, if $C_1 = P(f(x), g(y)) \vee Q(f(x))$ and $C_2 = R(x, y) \vee \neg Q(y)$, then $C_2/\eta = R(u, v) \vee \neg Q(v)$ and $C3 = P(f(x), g(y)) \vee \boxed{Q(f(x))} \vee R(u, f(x))$.

The *reduction* rule takes as input a chain C_1 and produces as output a chain C_2 provided that there is a matching substitution θ for the rightmost literal of C_1 and some framed literal that occurs earlier in C_1. C_2 is $C_1\theta$ with the rightmost literal deleted. For example, if $C_1 = \boxed{P(f(x), y)} \vee R(x, y) \vee \neg P(f(a), b)$, then $C_2 = \boxed{P(f(a), b)} \vee R(a, b)$.

The *contraction* rule is the simplest of the three rules; it takes as input a chain C_1 that ends on the right with at least one framed literal and produces a chain C_2 as output by deleting all of the framed literals to the right of the rightmost unframed literal. For example, if $C_1 = P(x, f(x)) \vee \boxed{Q(y)} \vee \boxed{R(z)}$ then $C_2 = P(x, f(x))$. For simplicity, we will combine contraction with extension and reduction and assume that it is performed whenever possible after the other two operations.

The propositional example at the beginning of this section illustrates all three operations. We assume that the first four clauses are exactly the same as before and start numbering model elimination steps at five.

5.	$\neg N \vee \boxed{\neg P} \vee \neg N$	extension by 3
6.	$\neg N \vee \boxed{\neg P} \vee \boxed{\neg N} \vee P$	extension by 1
7.	$\neg N \vee \boxed{\neg P} \vee \boxed{\neg N}$	reduction
8.	$\neg N$	contraction
9.	$\boxed{\neg N} \vee P$	extension by 1
10.	$\boxed{\neg N} \vee \boxed{P} \vee N$	extension by 2
11.	$\boxed{\neg N} \vee \boxed{P}$	reduction
12.	$\square$	contraction

One final warning: Model elimination refutations may have disjunctive solutions if some of the clauses are not Horn clauses. Suppose, for example, we want to prove that the conclusion $\exists x P(x)$ follows from the hypothesis $P(a) \vee P(b)$. When we translate this problem into clause form we obtain two clauses:

1. $\neg P(x)$
2. $P(a) \vee P(b)$

Note that $\neg P(x)$ serves as both the initial center clause and as a side clause. Although there is no single value of x that satisfies $\exists x P(x)$, a model elimination refutation is still easily obtained:

3.	$\boxed{\neg P(a)} \vee P(b)$	extending 1 by 2 with $x = a$
4.	$\boxed{\neg P(a)} \vee \boxed{P(b)}$	extending 3 by 1 with $x = b$
5.	$\square$	contraction

Instead of giving a single substitution for goal variables as in Prolog, the model elimination procedure gives the disjunction $x = a \vee x = b$ where each individual disjunct corresponds to one use of goal clause $\neg P(x)$ in the proof.

3 Or-Parallel Execution

Each node in a model elimination search tree can be viewed as a 4-tuple (C, L, A, B), where C is the *center clause* for the node, L is the *selected literal*, A is a list of unexplored *alternatives*, and B is a set of variable *bindings*. The *center clause* is the clause to be solved at this node, and the *selected literal* is the literal within the clause that will be solved first. The list of *alternatives* contains the untried extensions and reductions that may be used in solving the selected literal. The set of *bindings* gives the variable substitutions defined by unification at this point in the search.

The initial or root node in the search tree has C equal to the goal clause, L equal to the first literal within the goal clause, A equal to the set of extensions for L, and B equal to the empty set. The children of a node are formed by applying the extension and reduction alternatives A to C and L. In the process, new bindings are created during unification of an alternative with C and L. If (C', L', A', B') is one of the children, then C' and L' are obtained from C, L, and the appropriate element of A by an application of one of the rules of Section 2. A' is deduced from L' and C', and B' is B with the new bindings added.

The root of the search tree is at the "top" of the tree, and the leaf nodes are at the "bottom." A node is *dead* if its list of alternative clauses is empty. Otherwise, it is *open* or *live*, and we say that there is work available at the node. A *fork node* has more than one arc below it. A *branchpoint* is a node that is either a fork or is live (and hence is a potential fork).

In an or-parallel search, the descendents of each node can be explored simultaneously by a number of *processes* or *workers*. The central issue in such a search is how the descendants of a node can share bindings for variables. Several alternative schemes have been proposed for handling bindings [9]. We mention the Argonne and SRI models for or-parallelism, since most of the other models which have been proposed are very

closely related to these two. Both of these models divide bindings into two classes, conditional and unconditional, depending on whether they may have different bindings along different branches of the tree. In both cases, unconditional bindings are associated with the nodes in the tree.

The Argonne model handles conditional bindings by associating with each arc in the tree a hash table that stores bindings made to conditional variables when creating the node beneath that arc. In addition, the Argonne model splits conditional bindings into *favored* and *unfavored* bindings in an attempt to reduce the number of bindings that have to be stored in these hash tables. The major criticisms of the scheme concern the cost of maintaining and accessing these tables. In an early implementation of Parthenon, we experimented with a scheme for maintaining variable bindings in hash tables as in the Argonne model, but without favored bindings. The scheme did not perform well in our implementation. Since our proof trees tend to be relatively shallow, the overhead of maintaining the hash tables was simply too high.

In the SRI model, conditional bindings are placed in a private binding array belonging to the worker that is making the binding. Variables which are conditionally bound are trailed, along with the actual value of the binding. This information is used to update the binding array when a worker switches to a new task. Because of this update, switching jobs may take a large amount of time. This is the major criticism of the model. We implemented a version of Parthenon using the SRI model and obtained good performance, but execution profiling prompted us to switch to the scheme discussed below.

In the current version of Parthenon, each worker has a variable stack, a choicepoint stack, and a trail. When a worker creates a choicepoint, it uses its own choicepoint and trail stacks. Conceptually, the choicepoint and trail stacks are shared to form the search tree, while the variable stacks are local to each worker. Each of these data structures closely resembles the corresponding structure in a standard Prolog implementation. The operations of dereferencing, unbinding, etc. are performed exactly as in Prolog, with one exception. Whenever a variable binding is made, the address of the variable is trailed, while in Prolog, only conditional bindings are trailed.

Each choicepoint has an additional field that indicates which worker created it and whether it has been shared with any other workers. When a worker moves into a node while switching tasks, it examines this field. If the choicepoint has not been shared, the process which created the choicepoint must still be working in the subtree. In this case, the entering process uses the trail to find which variables were bound when the choicepoint was created. It then retrieves the bindings of these variables from the variable stack of the process which created the choicepoint, and it stores the value of these bindings in the trail entries associated with choicepoint and in its own variable stack. Note that, when a node becomes shared, both variable addresses and the corresponding bindings are stored in the trail, in contrast to the situation in Prolog. The choicepoint is then marked as shared. If another worker were to enter this same choicepoint, it would use the bindings from the trail to fill in the bindings in its own variable stack. The reason that this third process cannot fill in its variable stack in the same fashion as the second process is that the process which created the choicepoint may have left the subtree after the node became shared.

We chose this execution model for several reasons. First, our tests indicated that Parthenon spends a large percentage of the time dereferencing variables, checking for unbound variables, and fetching terms. Under this model, these operations take the same amount of time as in the WAM, unlike in the Argonne and SRI models. Second, while variable binding is slightly more expensive than in the WAM, our experience with other models showed us that many of the bindings made would be conditional, so the actual difference in time is small. Finally, because of the abundance of available work in most theorem proving problems, the task switch time is not so important. In any case, this time is not much worse than in the SRI model. Overall, we have observed a speedup of from twenty to fifty percent over the SRI version of Parthenon.

4 Overview of Parthenon

In this section we describe the current version of Parthenon. The system has been implemented in C with synchronization primitives provided by the C-Threads package [3] and has been evaluated on both the VAX-11/784 and Encore Multimax multiprocessors running MACH. The use of parallelism is intended to be completely transparent to the user: no special annotations are needed to make the system consider alternative inferences in parallel, as every branch point is potentially a parallel one. We felt that an annotation system would be too difficult to use effectively. This is because the user is likely to have less of an operational understanding of a set of axioms to be used for proving a theorem than of a Prolog program.

After reading and parsing the clauses, some preprocessing is done to build a table of the positive and negative instances of each predicate symbol. This as an obvious extension of the table built by a Prolog interpreter, since in general extensions can be carried out on both positive and negative literals. Additionally, any given side clause may be used in several ways to solve one literal. Thus, in effect, a clause with n literals may be viewed as n different "rules," each with a different literal, positive or negative, as its "head." There is also local processing for each clause. As in Prolog, the variables are numbered left to right according to their first occurrence in the clause. In addition, for each occurrence of a variable in a literal, we note whether it is the first occurrence in that literal. This is needed both to optimize the use of the occur check and to speed up variable binding in certain cases. The final preprocessing step involves generating the root of the search tree.

The structure of nodes in the search tree was described in the previous section. When a node is created the list of alternatives is initialized to point to all the appropriate reduction alternatives and all the clauses with an atom of the same predicate symbol and opposite sign. This list does not remain static throughout a computation—as the various alternative inferences are exhausted, the list is changed to reflect this. The job of the scheduling algorithm is to find a node in the tree where the list of alternatives is non-empty and present this node to the inference algorithm. The inference algorithm then chooses one of the alternatives to perform, removes it, and tries to carry out

the inference. If the inference is successful, the process moves to the resulting node without reconsulting the scheduling algorithm. If the inference failed, the bindings made during the unsuccessful unification are undone and the scheduler is called to find more work.

We use iterative deepening to guarantee completeness. In conjunction with this strategy, it may be possible to prune the search tree during each iteration. For example, if we are about to extend by a nonunit clause, the length of the proof along that path will be increased by at least one less than the number of literals in the side clause. If this number added to the number of unboxed literals remaining in the center clause exceeds the number of steps we have remaining in this iteration, we need not try the unification for this extension. We have also implemented a flexible deepening strategy that adjusts the amount by which the search is deepened at each iteration according to information gathered during the search.

A number of considerations are helpful in cutting down the size of the search tree. We consider three that were proposed and implemented by Stickel [8] in a sequential setting, and discuss their appropriateness in a parallel environment:

- Whenever a reduction can be made without specializing the current center clause, no alternative inferences need be considered. This is easy to implement and not very expensive. It is useful in a few cases.
- If a literal is identical to one of its ancestors, this path may be eliminated. This is easy to implement but has cost proportional to the depth of the node at which it is applied. It often has a dramatic effect on the size of a proof tree—reducing the number of inferences for some of the examples considered in Section 7 by up to an order of magnitude.
- If an extension against a unit clause can be made without specializing any variables in the center clause, no other alternatives need to be considered. This is not difficult to implement but is not as useful as one might expect. There is a much more useful case which is considerably more troublesome: If an atom in the center clause can be solved by some number of steps without specializing the other variables in the clause then no other means of solving that atom need be considered. Like Stickel, we have found this to be extremely helpful in certain cases.

It should be noted that the first and third points above are weakened somewhat by parallel execution, since by the time it is determined that the optimizations may be applied, some other processor may already have begun (or even finished) considering those alternatives. For this reason all we can hope to do is prune those alternate branches that have not yet "sprouted." One optimization which speeds up the actual inference rate is our special treatment of variables appearing for the first time in a literal that we are extending against in a side clause. When we unify such an occurrence of a variable with some term, we save time by not carrying out the occur check.

Because we are dealing with general clauses, the goal clause may also be used as a side clause in the computation. Hence, there may be more than one instance of substitutions for the variables in the goal. It is not sufficient for the theorem prover to return the bindings of variables in the original center clause as an answer: it must also return those bindings made whenever the goal is used as a side clause. The binding to the center clause variables, and each of the subsequent side clause bindings result in a substitution θ, such that *at least one* of these gives the resulting counterexample for variables in the goal clause. This raises the problem of finding the relevant bindings. In the Horn clause case this just involves looking up the variables at the base of the search tree, but in general it is necessary for each node to have a flag indicating whether the clause used for extension was the goal. Then when the computation has been completed, the path from the leaf node to the root is searched for the answer substitutions.

5 The Inference Mechanism

In this section we outline our implementation of the model elimination inference mechanism. We focus on how the inference loop is used to grow the search tree. We briefly describe the nodes in the search tree, and by concentrating on certain important fields in the nodes, will show how a new node is created in the cases of reduction, unit extension, and nonunit extension.

Each node in the tree must have the following details:

- enough information to reconstruct the entire center clause including boxed literals
- information about where to find bindings for the required unifications
- pointers needed to find children and siblings of this node, as well as the predecessor
- stack maintenance information
- information to reconstruct the proof and goal variable bindings once the proof has been found
- information needed for iterative deepening (e.g., depth)
- a lock for mutual exclusion, the active worker count, and the garbage collection status
- the next variable number for initializing uninstantiated variable value cells below this node

Space does not permit us to discuss all of the fields within a node. Instead, we will only describe the fields of particular interest for the inference loop.

nowdo This field holds a list of the literals remaining to be solved from the most recent nonunit side clause. The head of this list indicates the literal to be solved at this node.

unify_fp The *unify_fp* indicates the node where the most recent nonunit extension was performed. This node contains the environment for the variables in the *nowdo* list. This pointer is also used to find the alternatives for reduction.

calling_frame This corresponds to the "caller's environment" in Prolog.

retpt This indicates the next literal to be solved in the *nowdo* field of the node immediately above this one in the search tree.

The heart of each process is a loop which repeatedly calls the scheduling routine to find a job, then attempts to perform a single inference. When an inference results in the creation of a new node, the process immediately moves to the new node without calling the scheduler. The basic structure of the body of the inference loop is shown in Figure 1.

```
1   if (node is at depth bound)
2      cutoff and find more work;
3   if (a reduction alternative exists)
4      if (unification successful)
5         new_node - > retpt := node - > nowdo - > next;
6         new_node - > calling_frame := node - > unify_fp;
7         (new_nowdo, new_frame) :=
8         next literal to solve and its environment;
9         if (empty clause derived)
10           signal proof found;
11        new_node - > nowdo := new_nowdo;
12        new_node - > unify_fp := new_frame;
13        node := new_node;
14     else
15        reset bindings made during attempted unification;
16  else
17     if (extension cannot give proof within depth bound)
18        cutoff and find more work;
19     if (unification successful)
20        if (nonunit extension)
21           nowdo := remainder of clause;
22           new_node - > nowdo := nowdo;
23           new_node - > retpt := node - > nowdo - > next;
24           new_node - > calling_frame := node - > unify_fp;
25           new_node - > unify_fp := new_node;
26           node := new_node;
27        else if (unit extension)
28           new_node - > retpt := node - > nowdo - > next;
29           new_node - > calling_frame := node - > unify_fp;
30           (new_nowdo, new_frame) :=
31           next literal to solve and its environment;
32           if (empty clause derived)
33              signal proof found;
34           new_node - > nowdo := new_nowdo;
35           new_node - > unify_fp := new_frame;
36           node := new_node;
37     else
38        reset bindings made during attempted unification;
```

Figure 1: Procedure Inference_Step

There are three cases: reduction (lines 3 through 15), extension by a nonunit clause (lines 19 through 26), and extension by a unit clause (lines 27 through 36). The actions taken after reduction and unit extension cases are almost identical; hence we combine the discussions where appropriate.

The *nowdo* and *unify_frame_ptr* field determine the literal to be solved at the new node. In the nonunit extension case, the *nowdo* field is simply the remainder of the side clause. The *unify_frame_ptr* in this case will just be the new node. For the reduction and the unit extension cases, the situation is more complex. It is necessary to follow the chain of *calling_frame* pointers up the tree beginning with the new node, until we reach a node with a non-null *retpt*. This *retpt* will indicate the next literal L to be solved, and the node in which it occurs will give the *unify_frame_ptr* for L. These two values are installed in the *nowdo* and *unify_frame_ptr* field for the new node. If a non-null *retpt* is not found, the empty clause has been derived. The *calling_frame* for the new node is always simply the *unify_frame_ptr* for the parent node in the tree, and the *retpt* is the tail of the *nowdo* field for the parent node in the tree.

To find possible reduction alternatives for the literal to be solved in the new node, we must look at the framed literals for all prior nonunit extensions. Only framed literals which have the same predicate symbol but opposite sign are actually considered as candidates for reduction. To find the first possible alternative, we begin by following the *unify_frame_ptr* for the new node to a node N. The framed literal to be checked comes from the parent node of N in the search tree. The remaining framed literals can be found by following the chain of *calling_frame* pointers beginning with N, and checking the parent of each node in the chain.

6 Backtracking and Scheduling

This section describes the backtracking procedure used to find a choice point with open alternatives. Since the choice point records are logically arranged as a tree, the algorithm is essentially a tree traversal. Management of the choice point, trail, and environment stacks is related to backtracking and will also be briefly discussed within this section. A high level description of the procedure is given, followed by a bottom up presentation of the various support functions needed to implement it. We also sketch a proof of several key properties of the algorithm. Programs are described in a notation similar to C.

Warren proposes two basic scheduling strategies in his paper on the SRI model [10]. Both schemes involve maintaining information about available work and idle workers at each node in the tree. In the first scheme, the amount of information required at each node grows with the number of processors used. Moreover, a global data structure is needed to record nodes where work is available. The second strategy involves propagating information to ancestor nodes as work is created and consumed. We do not consider the first to be scalable to large numbers of processors. In the second case, the cost of propagating information grows with the depth of the tree. The method we use has the following advantages:

- No global structures are needed.
- The strategy is independent of the number of processors.
- No additional information is required at the nodes, so the cost of information propagation is avoided.
- The method allows a high degree of parallelism.

The search procedure used by Parthenon is a generalization of the backtracking process in Prolog. It is initiated when a

process finds no alternatives at the current node. During the search, the process is in one of two states: moving up or moving down. When moving down, the process first checks for alternatives at the current node. If none are found, the process moves to the left child (which becomes the new current node); if there is no left child, the process starts moving up. When a process is moving up, it looks for alternatives at the ancestor of the current node. If no work is found and the current node has a right sibling, the process moves to the right sibling and begins moving down. If there is no right sibling, the ancestor becomes the current node, and the process continues moving up. A process may also leave the tree without having found a job. In this case, the process begins the search again, starting at the root.

The method for maintaining the variable bindings is similar to the one used in the SRI model. As a process moves up the tree, the trail is used to remove bindings. The process keeps track of the highest node reached so far in the tree and bindings are removed only if a node higher than the current highest is reached. As an optimization, bindings are not added to the variable stack as the process moves down because the process may not find any work during its descent. These added bindings would then again have to be removed. Bindings are installed only after a node with open alternatives has been locked. At this point, the trail entries created between the highest node reached and the locked node are used to install bindings into the variable stack of the process.

Unlike the strategies proposed by Warren, our scheduling algorithm depends only on the structure of the tree. One objection to our scheme is that a process may try to search subtrees which contain no open alternatives. Because of the large branching factor in many examples, we do not feel that this is significant. Any time lost searching such subtrees is also offset by the time saved by not maintaining information about where work is available. A potentially more serious problem is that the procedure may create many small jobs and that processes may spend a large amount of time switching between tasks. Because of the need to maintain binding arrays, such switches can be costly. Again, we feel that the large branching factor will minimize this effect.

6.1 Details of the Algorithm

The *Exit_Subtree* procedure (Figure 2), is executed when a process is moving up the proof tree from the *child* node to the *parent* node. This procedure removes *child* as a descendent of *parent* and marks *child* for garbage collection if there are no remaining workers at *child*. If all nodes are exhausted, the *computation_done* flag is set. The procedure returns the node where the search is to be continued. The right sibling of *child* is returned if *child* is still linked into the tree. Otherwise, the *right_sibling* field of *child* may be invalid, so the search continues with the leftmost sibling of *child* (if there is no leftmost sibling, the process will continue moving up the tree).

The *Search_Once* procedure (Figure 3) searches for work starting at the node *from* and gives up if it fails to find work in one pass through the tree. It consists of two sections, one for the case when the process is moving down the tree and

```
1  choice_point *Exit_Subtree (parent, child)
2  choice_point *parent, *child;
3
4  choice_point *next := child -> right_sibling;
5  if (child -> left_child = NULL)
6     if (parent = NULL)
7        computation_done := TRUE;
8     else if (child -> free = LINKED)
9        Unlink (parent, child);
10    else next := parent -> left_child;
11 child -> workers := child -> workers - 1;
12 if (child -> workers = 0 AND parent ≠ NULL)
13    child -> free := FREED;
14 unlock (child);
15 return (next);
```

Figure 2: Procedure 1, Exit_Subtree

one for the case when the process is moving up the tree. The search begins with the process moving down. Consequently, the subtree rooted at *from* is searched first; if work is found in this subtree, no bindings will have to be undone. The first part of *Search_Once* handles the case of moving down. First, *from* is checked to see if it has any alternatives. If there are none, the process tries to move to the left child of *from*. If there is no left child, no further descent is possible, so the process begins moving up. Note that when a process begins moving up, there is no work at *from*. This property is invariant, so there is no need to check for alternatives at *from* in the case when the process is moving up. To perform an upward movement, the immediate parent is checked and returned if it has work. If there is no work at the parent, the process calls *Exit_Subtree* and tries to move to the node which is returned. This will be the right sibling of *from* unless *from* has already been unlinked from the tree (in which case it will be a sibling of *from*). If a valid node is returned, the process begins moving down starting at that node. Otherwise, the process moves to the parent of *from* and continues up. Throughout the procedure, a record is kept of the highest node reached in the tree. Whenever the process tries to move up past this node, *Reset_Bindings* is called to remove the appropriate conditional bindings.

If a single call to *Search_Once* does not yield a node with alternatives, the search is repeated beginning at the root of the tree. When an open choice point is found, the binding array is modified appropriately to reflect the environment as seen from this new node. Note that bindings are only added once a job has actually been found. As indicated before, this is done so that minimal changes to the binding array are made. When work is found, garbage collection is performed by removing dead nodes from the top of the choice point stack. The trail and environment stacks are also modified accordingly.

6.2 Correctness of the Algorithm

We conclude this section with an informal proof of correctness of the tree traversal algorithm. Program statements are referenced by the pair (Procedure number: line number in pro-

```
1   choice_point *Search_Once (from, ancestor, ba)
2   choice_point *from, **ancestor;
3   cell *ba;
4
5   boolean going_down := TRUE;
6   while (TRUE)
7         if (from - > alternatives)
8            return (from);
9         if (going_down)
10           if (from - > left_child ≠ NULL)
11              lock (from - > left_child);
12              unlock (from);
13              from := from - > left_child;
14              from - > workers := from - > workers + 1;
15        else
16           unlock (from);
17           going_down := FALSE;
18        if (NOT going_down)
19           previous := from - > prev;
20           if (previous ≠ NULL)
21              lock (previous);
22           lock (from);
23           if (from = *ancestor)
24              Reset_Bindings (from, previous);
25              *ancestor := previous;
26           next := Exit_Subtree (previous, from);
27           if (previous = NULL OR previous - > alternatives)
28              return (previous);
29           if (next ≠ NULL)
30              from := next;
31              lock (from);
32              from - > workers := from - > workers + 1;
33              unlock (previous);
34              going_down := TRUE;
35           else
36              unlock (previous);
37              from := previous;
```

Figure 3: Procedure 2, Search_Once

cedure).

Proposition For every node in the tree, the *workers* field at the node indicates the number of workers within the subtree rooted at the node.

Proof For an existing node, the *workers* field may be altered at lines 1:11, 2:14, 2:32 and in *Find_Work*. At each of these points, the node is locked, eliminating the possibility of interference from other processes. We now show that the proposition is an invariant of the program. Each node is created with this field initialized to zero. When 1:11 occurs, a process is leaving the subtree represented by the node. In the second and third cases, a process has just entered the subtree rooted at the node in question. In the last case, the count at the root is increased when a process enters the tree. Thus, in all cases, the invariant is maintained. □

Proposition Within a single worker process, locking and unlocking of a given node occur in strict alternation.

Proof The following is invariant at line 2:6:

> Either *going_down* is true and *from* was locked on the previous iteration, or *going_down* is false and no nodes are locked.

The above is true initially because the node *from* is locked in *Find_Work*. If the process is moving down, and *from* has a child, then 2:11 and 2:12 lock the child and unlock *from*, and *from* is set to the child. If there is no left child, 2:16 unlocks *from* and the process starts moving upwards. In the upward moving phase, 2:22 and 2:26 lock and unlock *from* respectively. Any previous node is locked at 2:21 and unlocked at either 2:33 or 2:36, one of which must occur. The direction is again changed at 2:34 at which point *from* has been locked. In all cases, we see that the invariant is maintained. The desired property follows immediately from the invariant. □

Corollary There are no deadlocks.

Proof Whenever two nodes are locked, the highest one is always locked first. Thus there is a natural total order followed by workers on the same branch of the tree when locking multiple nodes. □

Proposition The sibling pointers are manipulated correctly.

Proof Sibling pointers at a node and its siblings are changed only when the parent is also locked. The procedure *Unlink* is called in line 1:9 after both the node and its parent has been locked at lines 2:21 and 2:22. Furthermore, the process does not change the sibling and child pointers anywhere else in the program. □

Corollary When a node is freed, no other references are made to it.

Proof A node is freed only at line 1:13. Both the *parent* and *child* nodes are locked at this stage. At this point, the node has been unlinked from the tree. Since the *workers* field is zero at this point, there can be no other processes in the subtree rooted at the node. Thus if some other process can ever refer to the node, the other process must already have a pointer to the node. Such a pointer could only be obtained at 2:13 or 2:26, but at both of those places, the other process must have either the *parent* or the *child* node locked, which is impossible. Hence no other process can try to refer to the node again. The process which frees the node does not refer to the node again before moving to some different node, after which it can never get a pointer to the node again. □

Proposition The descent to a left child and move to a sibling are safe, i.e., the nodes cannot be deleted before the *workers* field is increased.

Proof These situations occur at lines 2:11 to 2:14 and 2:30 to 2:32. For the first situation to be unsafe, the child must be deleted as the process moves to the child, but this cannot happen because a deleting process must lock the child and its parent first. For second case, the process moving right must have locked the previous node, and a deleting process must lock the same node in order to remove the right sibling. □

Eventual termination of the search is guaranteed by unlinking each node from the tree as soon as no more work is possible in the subtree rooted at that node.

7 Performance Analysis

We have tested Parthenon on a large number of examples collected by Stickel [8]. This section presents speedup figures for some of the problems, and analyzes the performance of the scheduling algorithm outlined in Section 6. These examples were chosen to illustrate the effects of branching factor and problem size, and do not necessarily show the best speedups obtained. Even so, for the larger problems, the execution times show an almost linear speedup with the number of processors. Furthermore, as far as we have been able to determine, our scheduling algorithm does not deteriorate as the number of processors is increased. Whether this will remain true with very large numbers of processes remains to be seen. At this point, we feel that the results justify our choice of scheduling algorithm.

The first table presents some statistics on the nature of the problems, including measurements of the average branching factor in the proof tree, the average number of attempted unifications at a choice point, and the percentage of bindings which would be conditional in other or-parallel schemes. Though these figures vary between different executions of the same problem, the differences are minor.

Problem	% conditional bindings	Mean actual branching	Mean attempted branching
apabhp	25.40	2.438	7.732
ls36	38.08	2.180	8.432
has-parts2	28.21	1.600	4.098
wos1	48.21	1.970	6.374
wos4	34.43	3.214	8.713
wos10	35.44	2.104	7.281
wos21	31.42	2.098	8.183

Figure 4: Problem characteristics

To measure the speedup figures, we varied the number of processors from 1 to 15. The results are summarized in Tables 2 and 3. It is apparent from the tables that there is an almost linear speedup in the number of attempted inferences per second. Note that the problems with the lowest speedup, *has-parts2* and *wos1*, also have the lowest branching factors (refer to table 1). Table 3 shows that the execution times for some of the problems represent superlinear speedups. This results from a proof being found after only a small part of the tree has been searched (relative to the single processor case).

Table 4 gives percentages of times the processes have to move a certain number of nodes in the proof tree to find work. A move of distance 0 means that a process is able to find work at the choice point where it starts its search for open alternatives. The table indicates that these percentages are generally insensitive to the number of processors used. A noticeable exception to the above observation is *has-parts2*, which has a low branching factor. For this example, the average proportion of longer moves increases quite significantly as the number of processors is increased from 1 to 15 (though the total percentage of long moves is still quite low).

Problem	Attempted inferences per second, speedup Number of processors			
	1	5	10	15
apabhp	1493	7159, 4.8	13256, 8.9	17382, 11.6
ls36	2615	13365, 5.1	24498, 9.4	30936, 11.8
has-parts2	2314	10129, 4.4	14395, 6.2	11242, 4.9
wos1	1752	7919, 4.5	14099, 8.1	14163, 8.1
wos4	1537	7452, 4.8	14147, 9.2	18467, 12.0
wos10	2730	11896, 4.4	23645, 8.7	29940, 11.0
wos21	1708	8453, 5.0	16551, 9.7	21576, 12.7

Figure 5: Inference rates

Problem	Execution time in seconds, speedup Number of processors			
	1	5	10	15
apabhp	875.9	213.8, 4.1	154.1, 5.7	93.3, 9.4
ls36	1735.3	268.7, 6.5	238.0, 7.3	143.4, 12.1
has-parts2	13.0	1.8, 7.1	1.4, 9.3	1.6, 8.1
wos1	47.7	14.4, 3.3	3.4, 14.0	3.7, 12.9
wos4	8692.1	237.4, 36.6	163.3, 53.2	155.4, 55.9
wos10	129.3	33.1, 3.9	10.9, 11.8	8.2, 15.8
wos21	2164.6	371.2, 5.8	332.2, 6.5	240.5, 9.0

Figure 6: Execution times

8 Conclusions

Parthenon has been successful in using data structures developed for parallel implementations of logic programming languages like Prolog. Since our present implementation is an interpreter, it is unable to exploit the structure of terms to minimize the cost of unification, even though this information is known *a priori*. Although generating code for a parallel version of the Warren Abstract Machine is likely to be more complicated in the full first order case than in the Horn clause case, we hope to have a compiler running by the end of the year. We think this should speed up the system significantly, perhaps even by another order of magnitude.

In the next few months we also hope to experiment with alternative scheduling algorithms. Since the scheduling algorithm determines which node of the proof tree to expand next, it is crucial for obtaining a high inference rate. There are several different ways of implementing this procedure including the one we describe in Section 6 and the two mentioned by Warren in his paper on the SRI model. While we believe that our present scheduling algorithm is the most appropriate for large numbers of processors, we plan to study several different schemes and verify this. We consider the use of distributed scheduling to be a crucial point and note that other researchers have begun moving in this direction.

We feel that our deviation from the SRI model of or-parallelism was a significant factor in improving the performance of Parthenon. By treating variables bindings in a uniform fashion, we were able to achieve extremely rapid variable dereferencing and term lookup at only a small increased cost in binding. Because of the high percentage of conditional bind-

Problem	#p	Percentage of moves Move length					
		0	1	2	3	4	5+
apabhp	1	91.030	6.864	1.553	0.406	0.108	0.039
	5	90.169	6.830	1.542	0.394	1.029	0.037
	10	91.162	6.828	1.489	0.383	0.100	0.039
	15	91.169	6.781	1.512	0.390	0.010	0.048
ls36	1	89.125	10.303	0.503	0.065	0.004	0.001
	5	88.774	10.620	0.529	0.072	0.004	0.001
	10	89.117	10.274	0.535	0.068	0.005	0.002
	15	89.089	10.324	0.502	0.066	0.008	0.012
has-parts2	1	86.837	9.053	2.778	1.129	0.191	0.013
	5	85.896	9.606	2.892	0.998	0.149	0.458
	10	85.153	9.247	2.696	1.007	0.320	1.577
	15	83.546	8.704	2.788	1.140	0.516	3.307
wos1	1	82.316	16.585	0.956	0.121	0.019	0.004
	5	82.881	15.941	1.002	0.146	0.023	0.007
	10	82.859	15.910	1.005	0.152	0.028	0.046
	15	82.792	16.003	0.994	0.144	0.025	0.041
wos4	5	90.756	9.059	0.167	0.013	0.003	0.001
	10	89.936	9.849	0.197	0.015	0.003	0.001
	15	89.566	10.201	0.207	0.019	0.005	0.003
wos10	1	90.009	9.651	0.284	0.057	0.001	0.000
	5	88.611	10.944	0.350	0.087	0.005	0.003
	10	89.184	10.441	0.289	0.067	0.005	0.007
	15	89.172	10.462	0.295	0.059	0.004	0.008
wos21	1	88.886	9.832	1.067	0.185	0.026	0.004
	5	89.066	9.529	1.167	0.203	0.031	0.005
	10	89.129	9.452	1.176	0.206	0.032	0.005
	15	89.154	9.421	1.178	0.208	0.033	0.006

Figure 7: Performance of the scheduling algorithm

ings present in theorem proving problems relative to Prolog programs, or-parallel Prolog systems would benefit much less, if at all, from this scheme.

Finally, an important contribution of our project is our demonstration that the proof trees for a large number of well known examples from resolution theorem proving have high branching factors which can be exploited by a parallel theorem prover. We conjecture that this is true in general. From the results that we have obtained so far, it appears that typical theorems will have even higher branching factors than is common in logic programs. One possible explanation for this is that computational problems which are essentially deterministic and therefore have a low branching factor tend to be solved algorithmically and are not usually formulated as theorem proving problems. If our conjecture is correct, then parallelism is likely to revolutionize the field of automatic theorem proving.

Acknowledgements

We owe a great deal to Mark Stickel who communicated with us throughout the project and suggested many improvements. Paul Allen wrote a major portion of the first version of Parthenon. Sean Engelson also contributed to the first version during the summer of 1987. We are grateful to Nevin Heintze, Sunil Issar, and Milind Tambe for their careful reading of this paper.

References

[1] R. V. Baron, R. F. Rashid, E. Siegel, A. Tevanian, and M. W. Young. MACH-1: a multiprocessor oriented operating system and environment. In *New Computing Environments: Parallel, Vector and Symbolic*, SIAM, 1986.

[2] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1970.

[3] E. C. Cooper. C threads. 1987. unpublished manuscript.

[4] D. W. Loveland. *Automated Theorem Proving: A Logical Synthesis*. North Holland, 1978.

[5] D. W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Association of Computing Machinery*, 15:236–251, 1968.

[6] D. W. Loveland. A simplified format for the model elimination theorem-proving procedure. *Journal of the Association of Computing Machinery*, 16:349–363, 1969.

[7] *Multimax Technical Summary*. Encore Computer Corporation, 1986.

[8] M. E. Stickel. A prolog technology theorem prover. In *New Generation Computing 2, 4*, pages 371–383, 1984.

[9] D. H. D. Warren. *Or-Parallel Execution Models of Prolog*. Technical Report, Department of Computer Science, University of Manchester, 1987.

[10] D. H. D. Warren. The SRI model for or-parallel execution of prolog—abstract design and implementation issues. In *International Symposium on Logic Programming*, pages 92–102, 1987.