

PARTHENON: A Parallel Theorem Prover for Non-Horn Clauses*

SOUMITRA BOSE, EDMUND M. CLARKE, DAVID E. LONG,
and SPIRO MICHAYLOV
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

(Received: 10 May 1989; accepted: 10 August 1990)

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 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 and IBM's 64-processor RP3. We have found that many theorem proving problems exhibit a great deal of potential parallelism. Parthenon has been able to exploit much of this parallelism, producing both good absolute run times and near-linear speedup curves in many cases.

Key words. Resolution, parallelism, model elimination, Prolog technology.

1. Introduction

Parthenon is a parallel resolution theorem prover that has been implemented on several shared memory multiprocessors, including a 16-processor Encore Multimax and the 64-processor IBM RP3. 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 often much higher than those of Prolog programs. On many examples Parthenon has exhibited near-linear speedup with respect to the number of processors used, and for one theorem, we have been able to reduce the time required to find a proof from several hours to less than ten minutes. 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

*This research was partially supported by NSF grant CCR-87-226-33. An earlier version of this paper appeared in the Fourth IEEE Symposium on Logic in Computer Science, Asilomar, CA, June 1989. D.E.L. was partially supported by an NSF graduate fellowship. S.M. was partially supported by an IBM graduate fellowship.

[8] 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 [11] 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 [13]. Although it was originally developed for Horn clause logic programming, we show that it can also be modified to handle general first order clauses with the model elimination procedure. In our adaptation of Warren's scheme, each processor performs an iterated 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 [6]. Parthenon is implemented in C and uses the C-Threads package [5], which allows parallel programming under the MACH operating system [1]. 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. We have also ported the system to IBM's RP3. The RP3 is a large-scale research parallel processor developed at the T. J. Watson Research Center. The current version of the system has 64 ROMP processors. Each processor has 8 megabytes of storage, half of which is local (this last percentage is configurable at boot time). The machine does not support automatic cache coherency; instead, cache management is the responsibility of the programmer or compiler [7]. Like the Multimax, the RP3 runs the MACH operating system, and provides support for parallel C applications via the C-Threads package.

We have tested Parthenon on a large number of examples used by Stickel [11]. 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

on the Multimax is almost 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 most 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 three sections describe key elements of the design in greater detail. Section 8 discusses the performance of Parthenon on some of the examples mentioned above. The paper concludes in Section 9 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 Loveland [8] in 1968. Our implementation is based on a simpler format for the procedure that was also developed by Loveland [9]. There have been several successful sequential implementations of the procedure, the most recent of which was the Prolog Technology Theorem Prover developed by Stickel [11]. 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 by Chang and Lee [4] or Loveland [10].

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

$$\begin{array}{l} P \vee N \\ N \vee \neg P \\ \hline P \vee \neg N \\ \hline P \wedge N \end{array}$$

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

*PTTP compiles clauses into Lisp and runs on a Symbolics Lisp machine; hence it is faster than Parthenon running with one processor.

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 P or N . Next, observe that resolving any such single literal clause with one of the original clauses will 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 clause 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 $\mathcal{C} = \{P(x) \vee P(y), \neg P(x) \vee \neg P(y)\}$. If conventional resolution is applied to \mathcal{C} , it is impossible to obtain a clause with fewer than two literals even though \mathcal{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, factoring can result in a significant increase in the branching factor of the search, although it can be needed to achieve a proof.

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, \dots, 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 centre 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 some literal of the centre chain and a matching literal of some side clause in \mathcal{S} . There are many possible strategies for selecting literals, but that of selecting the leftmost literal makes the presentation clearer. Instead of discarding the first literal in the centre chain, 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 leftmost literal in the centre 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 leftmost literal in C_1 and some literal of $C_2\eta$, 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 leftmost 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 left of $C_1\theta$. For example, if $C_1 = Q(f(x)) \vee P(f(x), g(y))$ and $C_2 = R(x, y) \vee \neg Q(y)$, then $C_2\eta = R(u, v) \vee \neg Q(v)$ and $C_3 = R(u, f(x)) \vee \boxed{Q(f(x))} \vee P(f(x), g(y))$.

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 leftmost literal of C_1 and some framed literal that occurs earlier in C_1 . C_2 is $C_1\theta$ with the leftmost literal deleted. For example, if $C_1 = \neg P(f(a), b) \vee R(x, y) \vee \boxed{P(f(x), y)}$, then $C_2 = R(a, b) \vee \boxed{P(f(a), b)}$.

The contraction rule is the simplest of the three rules; it takes as input a chain C_1 that ends on the left with at least one framed literal and produces a chain C_2 as output by deleting all of the framed literals to the left of the leftmost unframed literal. For example, if $C_1 = \boxed{R(z)} \vee \boxed{Q(y)} \vee P(x, f(x))$ 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. | $P \vee \boxed{\neg N} \vee \boxed{\neg P} \vee \neg N$ | extension by 1 |
| 7. | $\boxed{\neg N} \vee \boxed{\neg P} \vee \neg N$ | reduction |
| 8. | $\neg N$ | contraction |
| 9. | $P \vee \boxed{\neg N}$ | extension by 1 |
| 10. | $N \vee \boxed{P} \vee \boxed{\neg N}$ | extension by 2 |
| 11. | $\boxed{P} \vee \boxed{\neg N}$ | reduction |
| 12. | \square | contraction |

One final warning: mode 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 centre 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. $P(a) \vee \boxed{\neg P(b)}$ extending 1 by 2
with $x = b$
4. $\boxed{P(a)} \vee \boxed{\neg P(b)}$ extending 3 by 1
with $x = a$
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 the goal clause $\neg P(x)$ in the proof.

3. Or-Parallel Execution

Conceptually, each node in a model elimination search tree can be viewed as a 4-tuple (C, L, A, B) , where C is the centre chain for the node, L is the selected literal, A is a list of unexplored alternatives, and B is a set of variable bindings. The centre chain is the chain to be solved at this node, and the selected literal is the literal within the chain 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 descendants of each node can be explored simultaneously by a number of processes. The central issue in such a search is how the

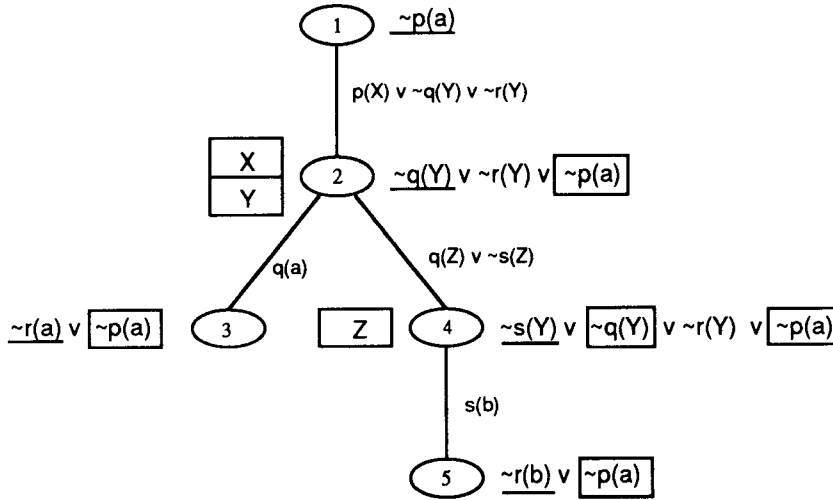


Fig. 1. Conditional and unconditional bindings.

descendants of a node can share bindings for variables. Several alternative schemes have been proposed for handling bindings [12]. Before discussing these, we review how bindings are handled in Prolog implementations. The root node of the search tree contains bindings for all the variables in the goal clause. A node representing a chain derived by extension contains bindings for all the variables in the side clause used for the extension. Hence, the variable bindings for a given chain are distributed along the path from the root node to the node representing that chain. We refer the reader to Bruynooghe [2] for more details.

We discuss 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. Figure 1 illustrates the difference between the two types of bindings. In this figure, variables are shown in boxes beside the nodes to which they belong. For example, node 2 results from an extension with the clause $p(X) \vee \neg q(Y) \vee \neg r(Y)$, and hence the node has variables X and Y . A variable is unconditionally bound when the binding will be the same for all the subtrees beneath the node to which the variable belongs. In the figure, X is unconditionally bound to a at the time node 2 is created, and Z is unconditionally bound to Y at the creation of node 4. A conditional binding, in contrast, results when different subtrees could possibly have different bindings for the same variable. As an example, consider the bindings of Y to a at the time node 3 is created and of Y to b when node 5 is created.

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 favoured and unfavoured 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 favoured bindings. The scheme did not perform well in our implementation; the overhead of maintaining the hash tables was simply too high.

In the SRI model, each process has access to the variable stacks of the other processes, but conditional bindings are placed in a private binding array belonging to the process that is making the binding. This is illustrated in Figure 2. The diagram shows two processes searching the tree of Figure 1. The binding arrays of the two processes are pictured at the very bottom of the figure, and their variable stacks are just above the binding arrays.

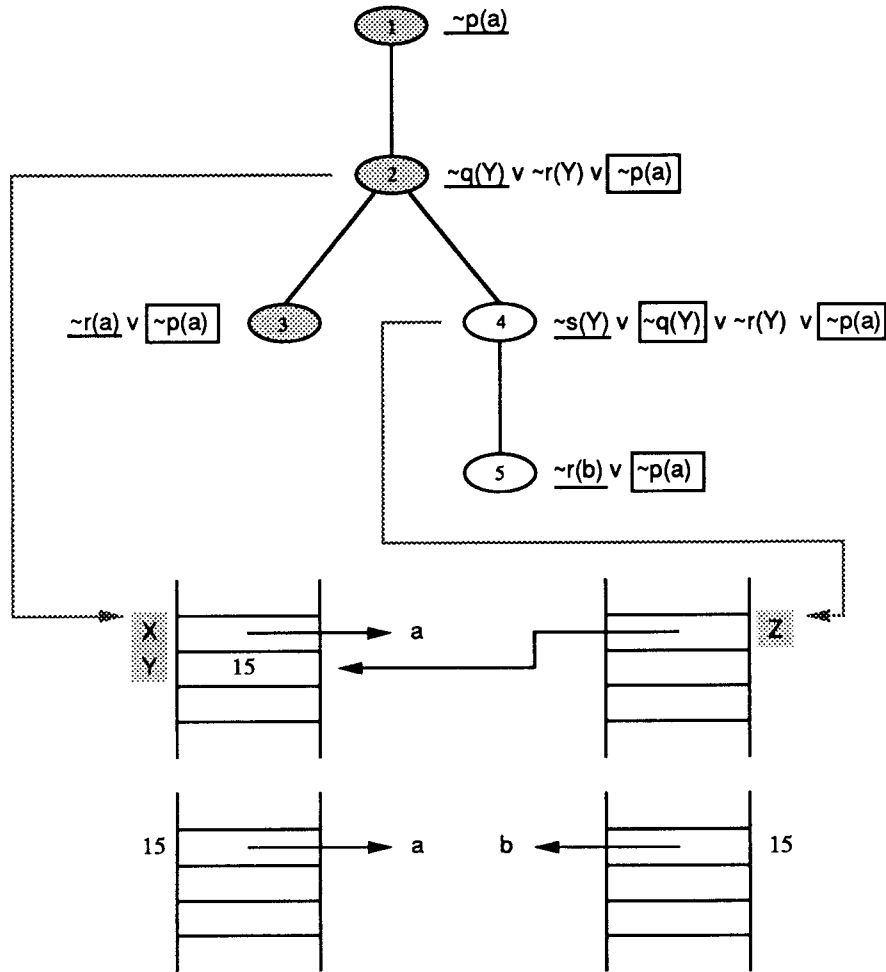


Fig. 2. Bindings in the SRI model.

process is at node 3 and its data structures are shown at left. The second process is at node 5. Associated with each node is a pointer (drawn as a gray arrow) to the variables of that node (only nodes 2 and 4 have variables). The unconditional bindings of X to a and Z to Y are made directly. For the conditional bindings of Y , an offset into the binding arrays is placed in Y , and the actual bindings to a and b are placed in the binding arrays at that offset. Variables which are conditionally bound are trailed (that is, the variables' addresses are placed on a stack so that their bindings may be undone on backtracking). The value of each binding is also trailed; this information is used to update the binding array when a process 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 in Section 6.

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 [5] and has been evaluated on the VAX-11/784, Encore Multimax and IBM RP3 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 is 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. Any given side clause may be used in several ways to solve one literal. A clause with n literals contributes n rules; each literal becomes the head of one of the rules. The head literal is the one which is considered when performing the extension operation. 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. The final preprocessing step involves generating the root of the search tree.

Nodes in the proof tree are represented with a structure which is a combination of Prolog's choicepoint and activation records. The structure contains such things as:

1. the clause used to derive the node (or none if the node was formed by reduction);
2. variables for the above clause (its environment) and their bindings;
3. a pointer to the node containing the environment for the currently selected literal (in the case of a node created by non-unit extension, this points to the node itself);
4. a list of extension and reduction alternatives;

5. an interlock and some bookkeeping information for synchronization and controlling the search; and
6. pointers to the parent, children, and siblings of the node (for maintaining the tree).

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 tried, 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. If there are no remaining alternatives at the node, 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 non-unit 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 unframed literals remaining in the centre chain exceeds the number of steps we have remaining in this iteration, we need not try the unification for this extension.

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 [11] in a sequential setting, and discuss their appropriateness in a parallel environment:

1. Whenever a reduction can be made without specializing any variables in the current centre chain, no alternative inferences need be considered. This is because no possible solutions for any remaining literals are eliminated. This is easy to implement and not very expensive. It is useful in a few cases.
2. Similarly, if an extension against a unit clause can be made without specializing any variables in the centre chain, 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 centre chain can be solved by some number of steps without specializing the other variables in the chain then no other means of solving that atom need be considered. Like Stickel, we have found this to be extremely helpful in certain cases.
3. 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.

It should be noted that the first and second 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 alternative branches that have not yet been attempted. 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 centre chain as an answer: it must also return those bindings made whenever the goal is used as a side clause. The binding to the centre chain variables, and each of the subsequent side clause bindings result in a substitution θ , such that *at least one* of these gives the resulting counter-example 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

Readers will find this section more accessible if they have a basic knowledge of the implementation of Prolog interpreters. Bruynooghe [2] is a good reference on the subject. Parthenon's inference mechanism is modeled after that of a standard Prolog interpreter. Aside from the use of the occur check, which is necessary for soundness, and some way to guarantee completeness, such as depth-first iterative deepening, the only real difference is the model elimination reduction step. The other inference rules are straightforward; the extension rule is analogous to a procedure call in Prolog, and the contraction rule is obtained for free (see below). For the reduction rule, what is required is a way to determine the framed literals in the current chain based on the current state of the proof tree. With this in mind, consider the following set of clauses.

1. $p(X) \vee q(a)$
2. $\neg p(a) \vee s(Z) \vee r(Y)$
3. $\neg r(a)$
4. $\neg s(Z) \vee \neg q(Z)$

Beginning with the first clause and extending with the others in turn gives the proof tree shown in Figure 3. Each node in the tree is labelled with the model elimination chain it represents. The selected literal at each step is overlined. In addition, each node is shown with a pointer (its 'calling pointer'). The target of this pointer contains the environment for the selected literal at the node and is referred to as the calling node.

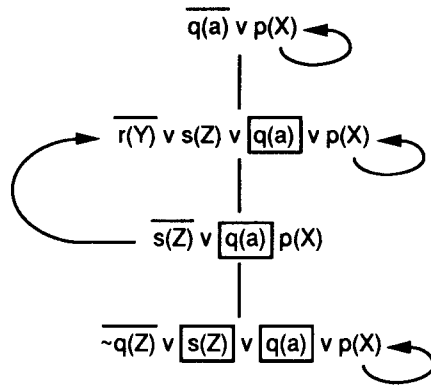


Fig. 3. Finding reduction alternatives.

For example, in Figure 3, the calling pointer for the second node in the tree points to the node itself since the second node was created by non-unit extension. The third node arose from unit extension, and the environment for the selected literal at that node, $s(Z)$, resides at the second node. To perform an extension operation with some clause, a new environment is created to hold the variables of the clause. Then we must unify with the selected literal in the current chain. This literal will in general contain variables, and the environment for these variables is the environment in the calling node. The important point for our purposes here is that the calling pointer at a node always points to the node representing the most recent active non-unit extension above that node (a non-unit extension is active if not all its subgoals have been solved). This is because unit extensions or reductions can only remove literals; they cannot create a new one which might then become the selected literal. Now, framed literals arise exactly when we perform a non-unit extension on some chain. When every other literal in the clause used for extension is solved, the framed literal is contracted away. Thus, to find all the framed literals, we must search for non-unit extensions which have not been completely solved. In Prolog, this corresponds to finding the selected atoms of all procedure calls which have not yet returned.

The observations above have several implications. First, we obtain the following algorithm for finding framed literals. The current node's calling pointer tells us the node n for the most recent active non-unit extension. Then the selected literal at n 's parent node is exactly the leftmost framed literal in the current chain. Now n 's calling pointer indicates the next earlier active non-unit extension, and the indicated node's parent gives the next framed literal. We continue moving up the tree in this fashion until we find a framed literal to use in a reduction step, or until we reach the top of the tree. Note that the nodes which correspond to extensions which are no longer active are still kept for backtracking purposes, but they are not on the chain of calling pointers beginning at the current node. The second implication is that the contraction operation of the model elimination procedure is automatically done at node creation time when the calling pointer of the new node is set. The only remaining detail is how

to tell which reductions have been (or are being) attempted in order to allow reductions to happen in parallel. This problem is solved by having a reduction pointer in each node which indicates where to start the search for candidate framed literals. When a process looks for a reduction to perform, it moves this pointer back up the tree in the manner described above. If it reaches the top of the tree, it sets the reduction pointer to null to indicate that all reductions have been tried and then looks for an extension. If a reduction alternative is found, it points the reduction pointer another step up the tree and begins working on the reduction.

6. The Variable Binding Model in Parthenon

As mentioned previously, the central issue in performing an or-parallel search is how to share bindings among the processes. Parthenon was originally based on the SRI model. Experimentation with this version showed three things.

1. There was enough available parallelism in most problems that processes rarely had to move very far to find work (see Figure 25).
2. Parthenon was spending large amounts of time dereferencing variables.
3. Many of the variable bindings were conditional.

The first observation above indicated that the major criticism of the SRI model, the relatively large amount of time required to update the binding array when switching jobs, was not significant. The second point meant that enough time was being spent following pointers in the unification routine to try to optimize the dereferencing. With this in mind, the last point is what led us to modify the SRI model. In the SRI model, a single step in dereferencing a variable involves a check to see whether the binding is conditional or unconditional and, if the binding is conditional, an extra indirection through the private binding array. The idea behind the scheme currently used in Parthenon is to eliminate the check and possible extra indirection by increasing the time required to switch tasks.

In the current version of Parthenon, each process has a variable stack, a choicepoint stack, and a trail. All of these structures are very similar to the corresponding ones in a Prolog interpreter. The variable stack is used to record bindings to variables; it replaces the binding array of the SRI model. The choicepoint stack is used to store information about the nodes in the search tree. The trail records variable bindings so that they may be undone upon backtracking. Conceptually, the choicepoint and trail stacks are shared to form the search tree, while the variable stacks are local to each process. One way to understand the binding model used in Parthenon is to imagine applying the following transformations to the SRI model.

1. Make all the bindings conditional. Once this is done, there is no longer a need to check the type of a binding.
2. If all the bindings are conditional, then every dereferencing operation will involve repetitions of the following procedure.

- (a) Look at the variable being dereferenced. If the variable is unbound, stop. Otherwise, since the binding is conditional, it contains an index into the private binding arrays.
 - (b) Get this index, and look in the appropriate binding array. If the binding array contains a pointer to a term, stop. Otherwise it contains a pointer to another variable. Follow the pointer to this variable and repeat the process.
3. Move the information about whether a variable is unbound or bound into the binding arrays. The procedure above then becomes the following.
- (a) Look at the variable to get an offset into the private binding arrays.
 - (b) Look in the appropriate binding array. If the variable is unbound or bound to a term, stop. Otherwise, we have a pointer back to another variable. Follow this pointer and repeat the process.
4. Now we see that the variables (and a level of indirection) can be eliminated, since all the important information is in the binding arrays. Doing this gives the following. Look in the binding array. If the variable is unbound or bound to a term, stop. Otherwise, we have another index into the binding array. Follow this index and repeat the process.
5. Now convert the indices into true pointers. Since variables are allocated and deallocated in a last-in-first-out manner as the tree is explored, we can keep the bindings in a simple stack. This completes the transformation.
- Look in the variable stack. If the variable is unbound or bound to a term, stop. Otherwise, we have another pointer into the variable stack. Follow this pointer and repeat the process.

In the above model, the operations of dereferencing, unbinding, etc. are performed exactly as in Prolog, with one exception. Whenever any variable binding is made, the address of the variable is trailed, while in Prolog (and in the SRI model), only conditional bindings are trailed. This additional trailing results in two effects.

1. Variable binding may be slightly slower, though this is actually not as clear-cut as it might first appear. While more bindings are certainly being trailed, there is no longer a need to check whether each binding will be conditional or unconditional.
2. Task switching is slower. When a process switches tasks, it must install more bindings in its variable stack, since there is no longer any sharing of bindings.

There is one additional problem which must be addressed if the above scheme is used: exactly what must be done for a process to switch contexts? Somehow, when a process moves to a new part of the search tree, it must update its variable stack appropriately. In Parthenon, as in the SRI model, the solution is to use the trail to record the values of bindings as well as information about which variables were bound. One possibility would be to store the values in the trail as the binding is made, but this slows down binding. Instead, we add an additional field to each choicepoint that indicates which process created the choicepoint and whether it has been shared

with any other processes. When a process 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 the choicepoint and in its own variable stack. The choicepoint is then marked as shared.

Note that the process which created the choicepoint originally may leave the choicepoint even though the node is not dead. As an example of how this can happen, consider the following scenario.

1. The node is created by process A.
2. Process B moves into the node, selects the last alternative for extension/reduction, and begins working on it.
3. Process A finishes the alternative it is working on. It looks for another alternative, finds none, and exits the node.
4. Process B successfully completes its extension/reduction step. The new child node has many alternatives.

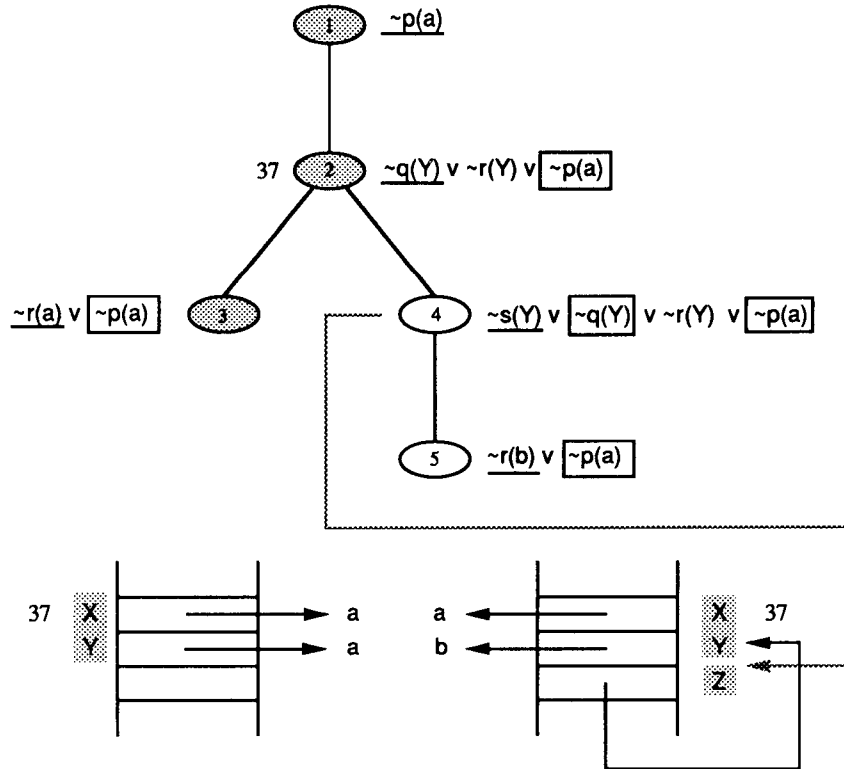


Fig. 4. Bindings in Parthenon.

If another process were to enter this same choicepoint, it would have to use the bindings from the trail to fill in the bindings in its own variable stack. Our measurements have indicated that only a few choicepoints near the top of the search tree ever become shared, so the idea of not filling in the trail completely until the node becomes shared saves a large amount of effort.

The above scheme is illustrated in Figure 4. The example from Figure 1 is used again. As in Figure 2, there are two processes, the first of which has created nodes 1, 2 and 3 and whose variable stack is shown on the left. Here, nodes 1 and 2 are shared, while nodes 3, 4 and 5 are unshared. Note that the second process has obtained the binding for X from the variable stack of the first process. It would also convert the address of X , stored in the trail, from a pointer to an offset into the variable stacks. The trail would then be augmented with the value of the binding, in this case a pointer to the term a . Thus, the trail at node 2 would then have an offset into the variable stacks rather than a pointer directly into the stack of the first process as the node had when it was created.

We feel that this binding model is especially appropriate in a theorem proving context because of the large amount of available parallelism and the relatively large percentage of conditional bindings. Overall, we have observed a speedup of from twenty to fifty percent for a given number of processors over the version of Parthenon based on the SRI model.

7. The Scheduling Algorithm

There are a number of desirable, but sometimes mutually exclusive, properties for a scheduling algorithm in a system like Parthenon.

1. It should have low overhead in terms of time and space.
2. It should be scalable to large numbers of processes.
3. It should keep all the processes busy.
4. It should assign large jobs to processes so as to minimize context switch overhead.

Fortunately, in a theorem proving context, there is a large amount of available parallelism due to the high branching factors in many problems. Hence, simple distributed scheduling algorithms which have the first two properties may also satisfy the last two constraints.

The scheduler in Parthenon is called when a process finds no alternatives at the current node. During the search for alternatives, 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 into one of the child nodes and repeats the process; if there are no children, the process starts moving up. When a process is moving up, it looks for alternatives at the ancestor of the current node. If none are found and the current node has some sibling nodes to the right, the process moves to one of the siblings and begins moving down. If there are no siblings

to the right, the ancestor becomes the current node, and the process continues moving up. A process may also leave the tree without having found an alternative. In this case, the process begins the search again, starting at the root.

In the description above, no mention is made of how children are selected when moving down, or how siblings are selected when moving up. These issues do not appear to be that crucial, but we have found that a left-right traversal with some simple load balancing gives good results. In order to efficiently implement such a strategy, each node contains a count of the number of processes working in the subtree rooted at that node. Then the scheduler uses the following rules.

1. When moving down, move to the child with the fewest processes working at or below it. Break ties by favouring children at the left.
2. When moving up and travelling to a sibling, move to the sibling with the fewest processes working at or below it. Break ties by moving to the leftmost candidate sibling.

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 remaining alternatives has been found. At this point, the trail entries for the nodes between the highest node reached and the node with an alternative are used to install bindings into the variable stack of the process.

Some care must be taken during traversal of the tree to ensure mutual exclusion at various points. For example, when a node is being removed from the tree, we must make sure that other processes do not try to enter it. Another problem which must be avoided is having two processes both try to take the same alternative. To avoid such problems, each node in the tree contains a lock. The rules for using these locks are as follows.

1. When checking for alternatives, updating the count of processes, creating a child, etc. at a node, the node must be locked.
2. When moving from a node to one of its children, both the parent and the child must be locked.
3. When moving from a child to its parent, both must be locked. The case of moving from a child to a sibling is treated as a move from the child to the parent followed by a move from the parent to the sibling.

To avoid deadlocks when a process must acquire two locks, the highest node in the tree is always locked first.

Figure 5 illustrates the movements of a process in search of alternatives. In the diagram, the process to the extreme left has started looking for a job.

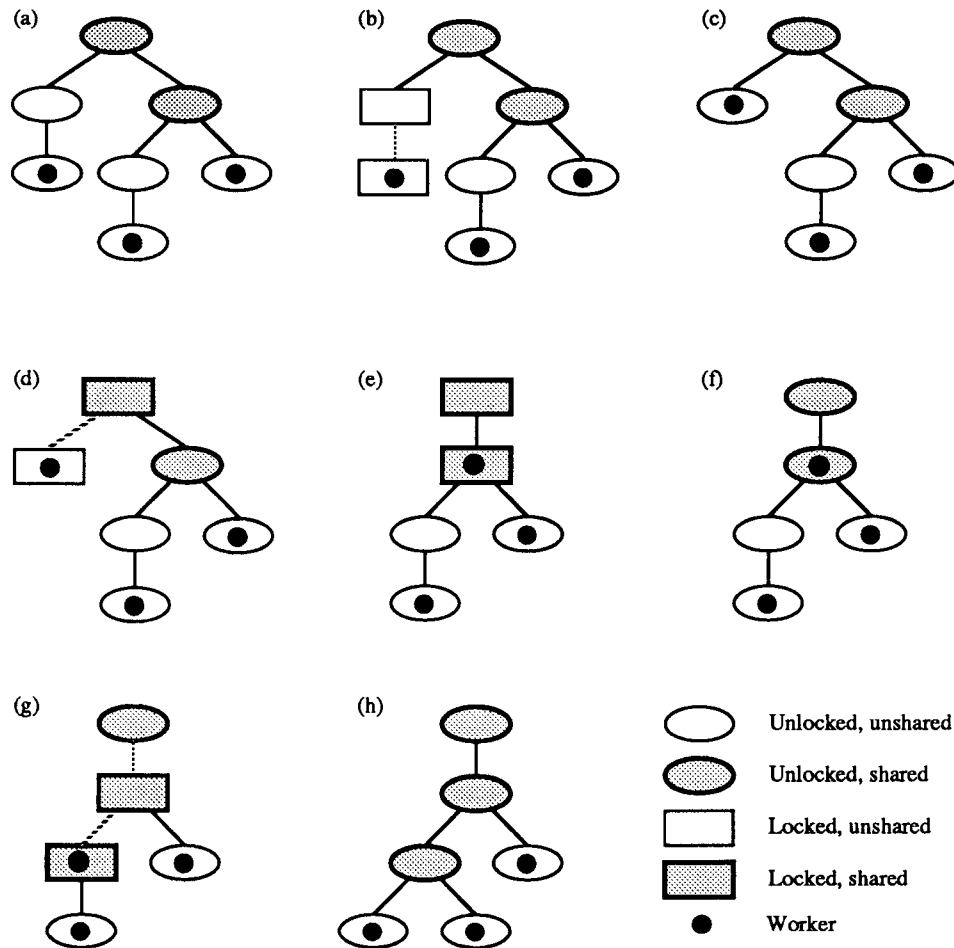


Fig. 5. Process movement during tree search.

- (a) The process is preparing to move.
 (b) and (c) The process locks the current node's parent and current node. Bindings are removed from the variable stack as the process moves up.
 (d) through (f) The process moves up again and starts down the remaining subtree of its parent. Bindings are again removed.
 (g) The process continues down and changes a formerly unshared node to shared. An alternative is found and bindings are installed.
 (h) The process creates a new node and moves into it.

More complex schemes are possible. For example, in his original paper on the SRI model, Warren proposes two basic scheduling strategies [13]. 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. More recently, other researchers have also recognized the desirability of distributed scheduling, and much work on or-parallel Prolog systems has been focused on efficient distributed scheduling algorithms [3]. However, we feel that these more complex procedures are unnecessary in a context where there is plenty of available parallelism such as theorem proving.

8. Performance Analysis

We have tested Parthenon on a large number of examples used by Stickel [11]. This section presents speedup figures for some of the problems, and analyzes the performance of the scheduling algorithm outlined in the previous section. For the problems requiring more extensive searches, the inference rates 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 significantly as the number of processors is increased. The data in this section represent single runs, but the inference rates and most of the run times are repeatable.

The table in Figure 6 presents some statistics on the nature of the problems, including measurements of the average branching factor in the proof tree, the average attempted branching at a choice point, and the percentage of bindings which would be conditional in other or-parallel schemes. Attempted branching measures the number of literals in side clauses and framed literals which have the appropriate sign and predicate symbol. Though these figures vary between different executions of the same problem, the differences are minor. These statistics were collected on the Multimax.

Figures 7 through 13 give speedup curves for the Multimax.* Each figure consists of two graphs, one showing the speedup in actual (solid curve) and attempted

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

Fig. 6. Problem characteristics.

* The performance figures given here differ from those presented in an earlier version of the paper. The earlier figures were obtained using an iterative deepening constant which was chosen based on the problem. The current figures are based on using a deepening constant of one for all the examples. We feel this is more appropriate since in practice it is impossible to predict the constant which would give best performance.

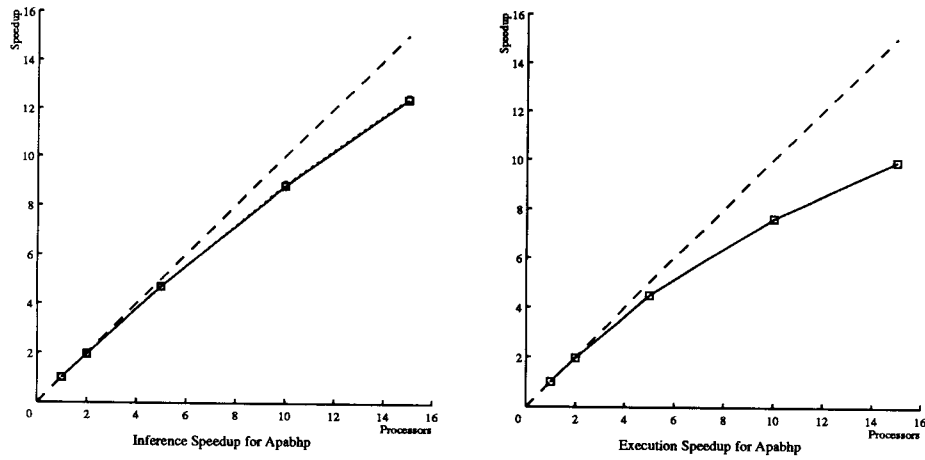


Fig. 7. Multimax speedup curves for apabhp.

(dotted curve) inferences per second and one showing the speedup in execution times. Figures 14 through 18 give speedup curves for the RP3. For the RP3, the speedup values were calculated relative to the 10 processor case, i.e., a factor of 2 speedup going from 10 to 20 processors was considered perfect.* For those problems which were not run with 10 processors, we calculated the speedups by assuming that the 20 processor case had a speedup of exactly 2. The data used to produce the graphs is given in Figure 19 through 24. We should point out that the RP3 does not support cache coherency in hardware, and that Parthenon was originally written assuming

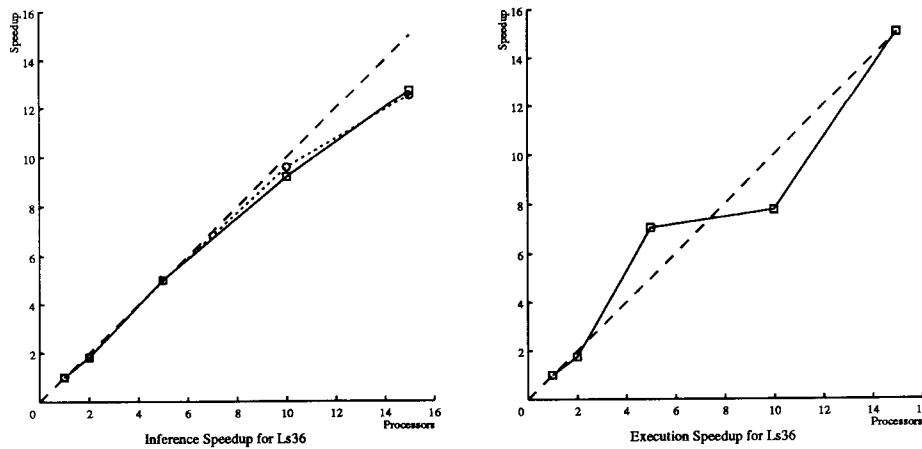


Fig. 8. Multimax speedup curves for ls36.

* The RP3 is an experimental machine located at the IBM T. J. Watson Research Laboratory. Because the machine is used by many researchers and because the individual processors are fairly slow, we were unable to obtain timings for smaller numbers of processors. We feel that it is important to report the RP3 measurements because the RP3 is the largest multiprocessor available to us for which Parthenon was suited.

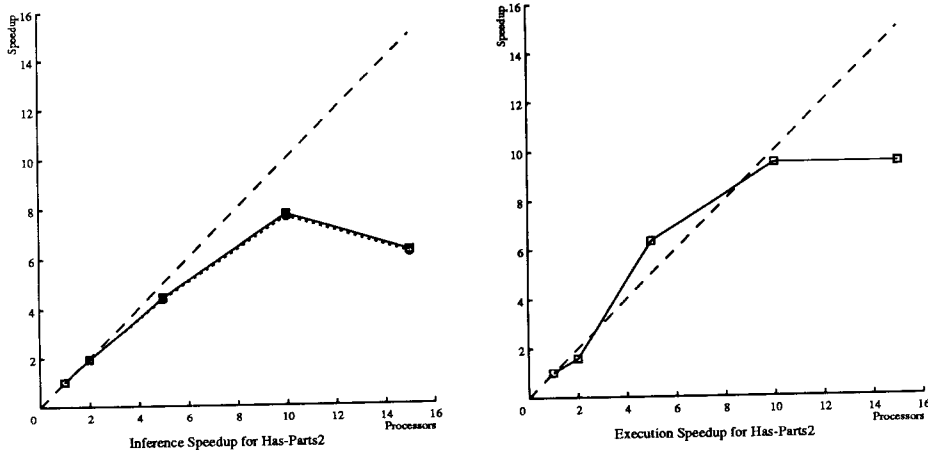


Fig. 9. Multimax speedup curves for has-parts2.

that such support would be available. We modified Parthenon slightly for the RP3; for example, the variable stacks were made to reside in the local memory of the appropriate processor. However, the RP3 implementation certainly does not take full advantage of the machine.

Note that the inference speedup curves for all but two of the examples are very close to linear; the exceptions are *has-parts2* on the Multimax and *wos10* on the RP3. Both of these problems are so small that, beyond a certain point, adding processors does not improve performance. It is also worth noting the disparity between the speedup curves for inference rates and execution times; the curves for the execution times are not as predictable as those for the inference rates. In particular, the execution speedups vary from being substantially sublinear to substantially superlinear (sometimes even within the same problem). This is caused by the interaction between

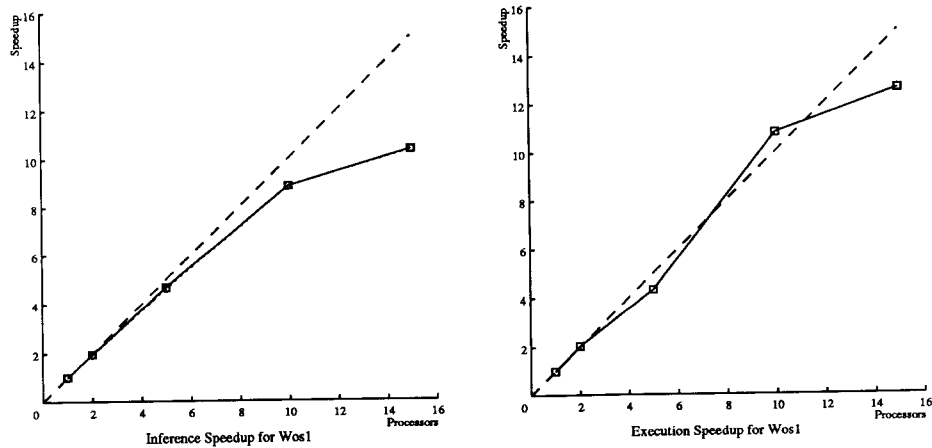


Fig. 10. Multimax speedup curves for wos1.

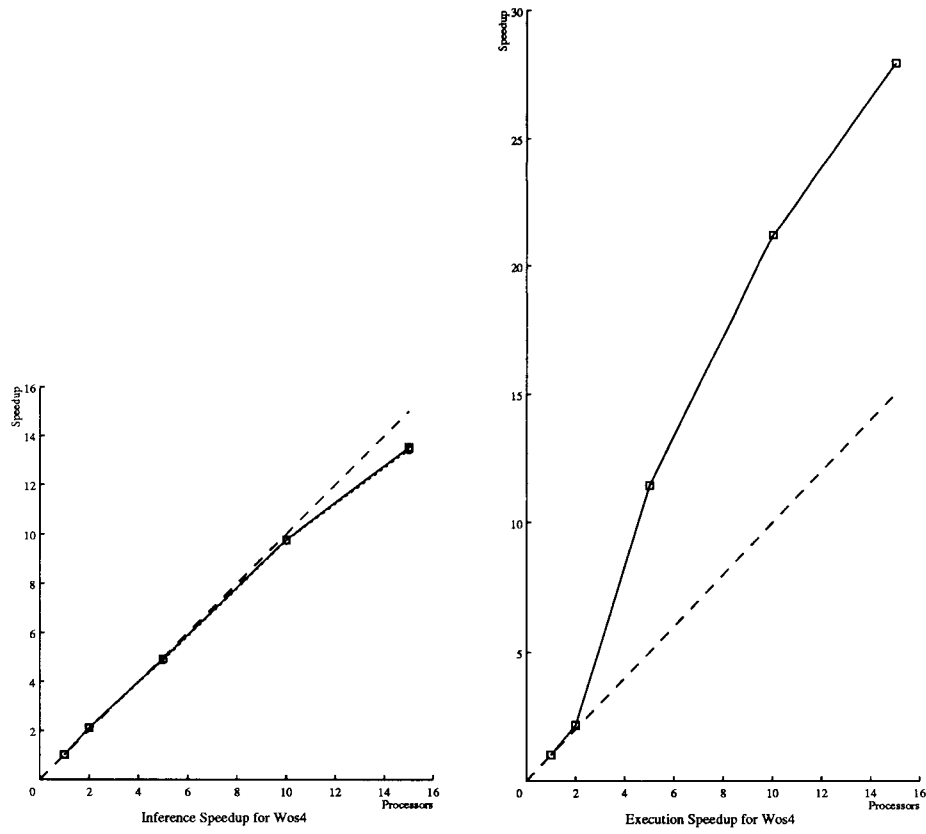
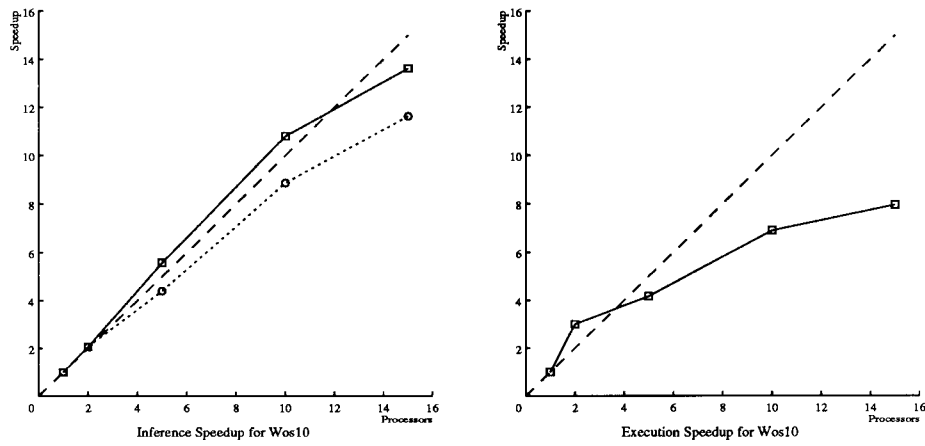


Fig. 11. Multimax speedup curves for wos4.



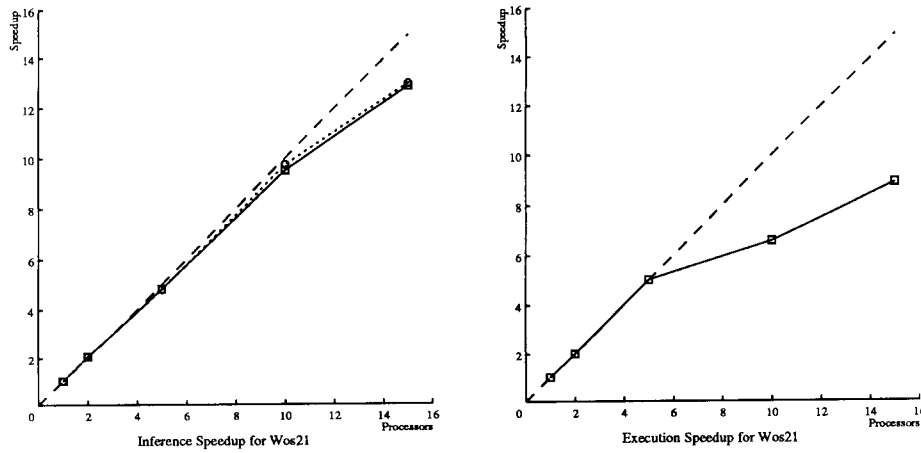


Fig. 13. Multimax speedup curves for wos21.

the parallel search strategy and the distribution of solutions in the search tree. For example, if the third processor in a search with three processors happens to go directly to a solution, the speedup from one to three processors will be highly superlinear, while adding additional processors may result in no further improvement (i.e., sub-linear speedup beyond three processors). Although the speedup in terms of elapsed time is the ultimate measure of the success of a parallel theorem prover, it is also important to examine the overhead in scheduling and communication. Since the cost of an individual inferences is essentially independent of the number of processors, the overhead can be easily inferred by looking at the increase in the inference rate.

We have also measured the performance of the scheduling algorithm directly for the Multimax. Figure 25 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

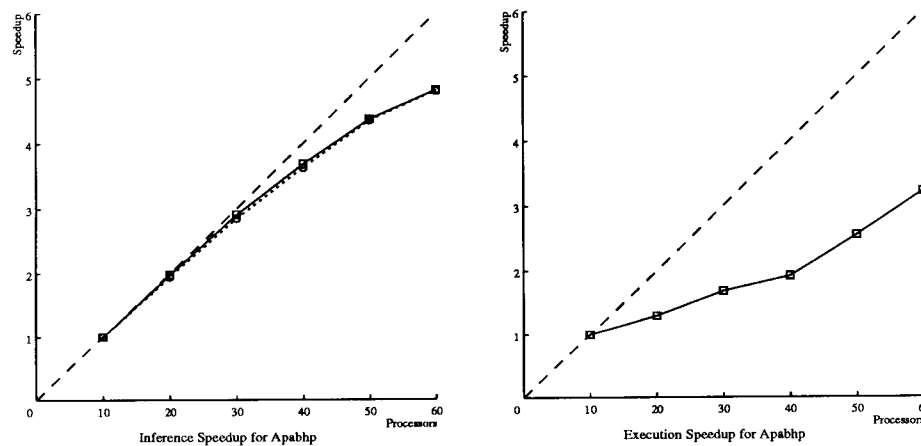


Fig. 14. RP3 speedup curves for apabhp.

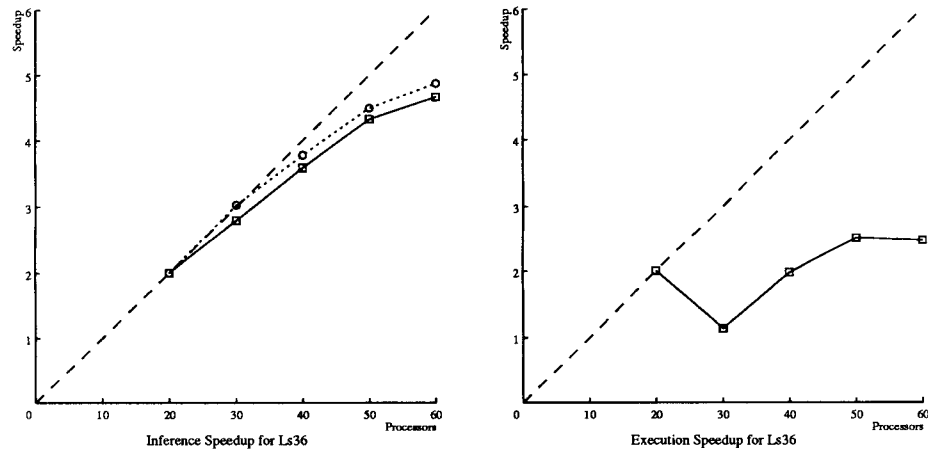


Fig. 15. RP3 speedup curves for ls36.

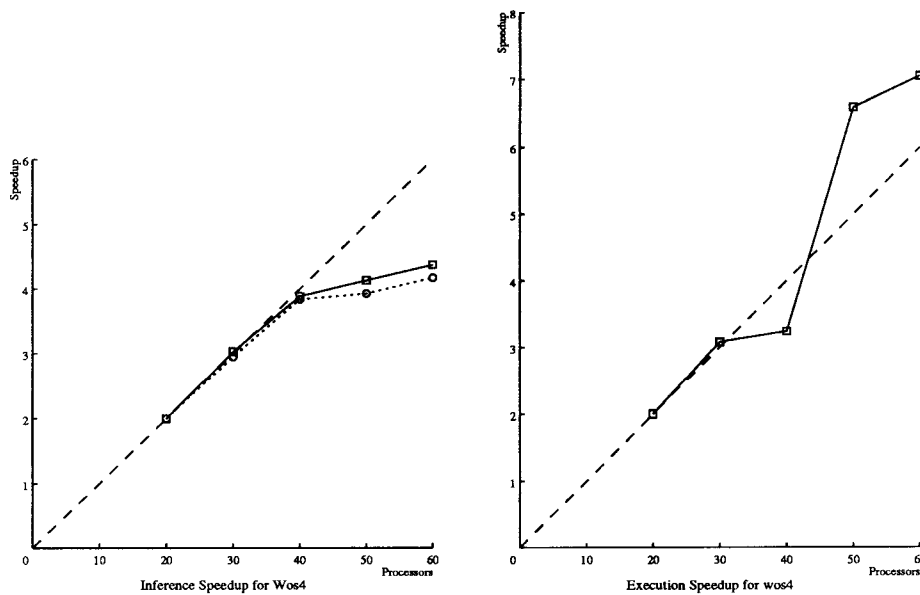


Fig. 16. RP3 speedup curves for wos4.

a process is able to find work at the choice point where it starts its search for 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).

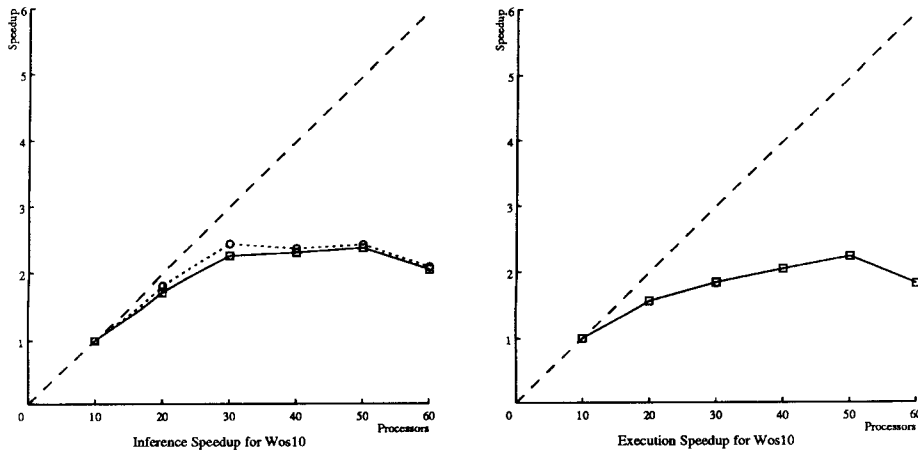


Fig. 17. RP3 speedup curves for wos10.

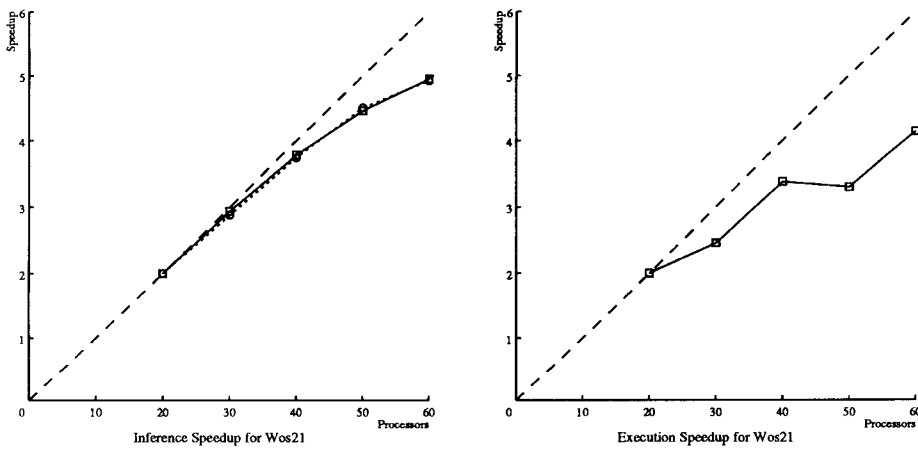


Fig. 18. RP3 speedup curves for wos21.

Problem	Successful inferences per second				
	Number of processors				
	1	2	5	10	15
apabhp	605	1179	2847	5304	7478
has-parts2	374	721	1652	2901	2339
ls36	644	1175	3246	5931	8197
wos1	914	1794	4273	8100	9465
wos4	705	1489	3481	6891	9532
wos10	619	1274	3455	6685	8419
wos21	778	1612	3753	7388	10023

Fig. 19. Multimax successful inference rates.

Problem	Attempted inferences per second Number of processors				
	1	2	5	10	15
apabhp	1660	3230	7799	14644	20606
has-parts2	1466	2855	6386	11207	9028
ls36	2793	5166	13976	26805	34956
wos1	1869	3652	8649	16557	19323
wos4	1568	3321	7675	15280	21059
wos10	2822	5802	12418	25024	32780
wos21	1863	3876	8938	18134	24210

Fig. 20. Multimax attempted inference rates.

Problem	Multimax execution time in seconds Number of processors				
	1	2	5	10	15
apabhp	2886	1471	645	379	292
ls36	2273	1300	323	292	151
has-parts2	38	24	6	4	4
wos1	151	74	35	14	12
wos4	14000	6476	1225	660	501
wos10	159	53	38	23	20
wos21	3315	1653	661	504	373

Fig. 21. Multimax execution times.

Problem	Successful inferences per second Number of processors					
	10	20	30	40	50	60
apabhp	626	1237	1822	2303	2735	3011
ls36	—	1468	2056	2633	3170	3423
wos4	—	1752	2658	3411	3622	3831
wos10	791	1374	1789	1832	1886	1622
wos21	—	1876	2759	3547	4185	4645

Fig. 22. RP3 successful inference rates.

Problem	Attempted inferences per second Number of processors					
	10	20	30	40	50	60
apabhp	1800	3511	5135	6507	7828	8635
ls36	—	6661	10102	12597	14944	16214
wos4	—	3878	5713	7454	7629	8076
wos10	2847	5190	6952	6734	6910	5963
wos21	—	4327	6233	8088	9762	10635

Fig. 23. RP3 attempted inference rates.

Problem	RP3 execution time in seconds					
	Number of processors					
	10	20	30	40	50	60
apabhp	1093	842	647	569	429	339
ls36	—	645	1122	660	514	526
wos4	—	935	606	576	283	265
wos10	152	96	82	74	68	83
wos21	—	1446	1180	855	874	697

Fig. 24. RP3 execution times.

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

Fig. 25. Performance of the scheduling algorithm.

9. 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*. Compiling clauses should speed up the system significantly.

We believe that our present scheduling algorithm is appropriate for large numbers of processors. However, it might be interesting to experiment with alternative scheduling algorithms to try to 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 variable 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 bindings present in theorem proving problems relative to Prolog programs, we suspect that this scheme would be inappropriate for or-parallel Prolog systems.

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 many 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 have a significant role in the implementation of future automatic theorem provers.

Acknowledgements

We owe a great deal to Mark Stickel, who communicated with us throughout the project and suggested many improvements, and to Donald Loveland, Ross Overbeek, and David Plaisted for many helpful discussions. 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, and to David Black for assisting us in collecting statistics on the Encore. George Paul and Bryan Rosenburg at IBM helped us get Parthenon running on the RP3.

References

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

2. Bruynooghe, M., 'The memory management of Prolog implementations'. In K. L. Clarke and S.-A. Tarnlund (Eds.), *Logic Programming*, pp. 83–98. Academic Press, London (1982).
3. Butler, R., Disz, T., Overbeek, R., and Stevens, R., 'Scheduling or-parallelism: An Argonne perspective'. In R. A. Kowalski and K. A. Bowen (Eds.), *Proceedings of the Fifth International Conference on Logic Programming*, pp. 1590–1605, Cambridge (1988) MIT Press.
4. Chang, C. and Lee, R., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press (1970).
5. Cooper, E. C., 'C threads', Technical Report CMU-CS-88-154, Carnegie Mellon University, Pittsburgh, PA 15213 (June 1988).
6. Encore Computer Coporation, *Multimax Technical Summary* (1986).
7. IBM, *Research Parallel Processor Prototype Principle of Operations*.
8. Loveland, D. W., 'Mechanical theorem proving by model elimination', *Journal of the ACM* **15**, 236–251 (1968).
9. Loveland, D. W., 'A simplified format for the model elimination theorem-proving procedure', *Journal of the ACM* **16**, 349–363 (1969).
10. Loveland, D. W., *Automated Theorem Proving: A Logical Synthesis*, North-Holland (1978).
11. Stickel, M. E., 'A Prolog technology theorem prover'. In *New Generation Computing* **2**, *4*, pp. 371–383 (1984).
12. Warren, D. H. D., 'Or-parallel execution models of Prolog'. Technical report, Department of Computer Science, University of Manchester (1987).
13. Warren, D. H. D., 'The SRI model for or-parallel execution of Prolog – abstract design and implementation issues'. In *International Symposium on Logic Programming*, pp. 92–102 (1987).

