Parallel Symbolic Computation on Shared Memory Multiprocessors

E. M. Clarke D. E. Long[1] S. Michaylov[2]
S. A. Schwab[3] J. P. Vidal S. Kimura[4]

October 1990
CMU-CS-90-182

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]This author was partially supported by an NSF graduate fellowship.

[2]This author was partially supported by an IBM graduate fellowship.

[3]This author was partially supported by an NSF graduate fellowship.

[4]This author's current address is Kobe University, Japan

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20 and in part by the National Science Foundation under grant CCR-87-226-33.

Abstract

We describe the implementation of three symbolic computation algorithms on shared memory multiprocessors. We also evaluate the performance of the implementations, point out some of their common characteristics, and describe why these algorithms should be able to take advantage of the large scale heterogeneous shared memory machines currently being developed.

1 Introduction

Symbolic computation algorithms have proven useful in a wide range of applications from formal verification to computer algebra. However, the time required by these algorithms restricts the size of the problems which can be solved. In light of this, it is somewhat surprising that there have been few parallel implementations of these algorithms. One possible reason is the inherent complexity of these algorithms; obtaining a good sequential implementation can require deep knowledge of the problem domain and extensive programming skills. In this paper, we consider typical symbolic algorithms from three different areas: theorem proving, computer aided design, and computer algebra. We show that on a shared memory multiprocessor each of the algorithms has a natural decomposition into parallel processes. We evaluate the performance of each algorithm on several different multiprocessors in order to gain an understanding of what architectural features facilitate the construction of such a program.

There are two obvious ways of obtaining faster symbolic algorithms. The first is to use new insight into the problem domain to improve the algorithm. This is certainly the most natural way of obtaining a faster algorithm, but it may be quite difficult in practice to gain such insight. Building a parallel implementation of the algorithm also has some drawbacks. One of the most serious is that a parallel implementation may not be able to take advantage of all the optimizations possible in a sequential algorithm. Nevertheless, in many cases it is possible to obtain near linear speedup using parallelism. This can have a dramatic effect on the usefulness of the algorithm. In this paper, we consider three implementations of symbolic algorithms. Each required a considerable amount of programming effort as well as study of the problem domain. We believe that our results are realistic and demonstrate what can be achieved by using parallelism. We briefly describe each of the three problems below.

Resolution theorem proving forms the basis for logic programming and is used in many automated reasoning systems. The OTTER [20] theorem proving system has been used to discover new results in combinatory logic. Stickel's Prolog Technology Theorem Prover [24] demonstrated that resolution systems could achieve high inference rates. Parallel logic programming languages are currently the focus of intense research [26]. We discuss a parallel resolution theorem prover called Parthenon [3] which uses or-parallel Prolog technology to achieve good performance.

Boolean decision diagrams are an efficient symbolic representation for boolean functions [6]. They are rapidly becoming widely used in circuit verification [15, 10] and simulation [5]. There have been a number of highly optimized sequential implementations [4]. Nevertheless, the enormous complexity of modern VLSI designs can strain the capabilities of these systems. We describe a parallel implementation of the boolean decision diagram manipulation routines [18].

The last example is a parallel implementation of the Gröbner basis algorithm. This algorithm has a large number of applications. In particular, it can be used to find exact solutions of systems of polynomial equations. The algorithm is a key component of most symbolic mathematics systems such as Mathematica, Scratchpad, and Maple. There have been several good sequential implementations such as that in the Macaulay system. An implementation on a Cray X-MP using a vectorized big number package is described by Neun and Melenk [22]. Vidal [25] was apparently the first to develop a parallel implementation of the algorithm for a shared memory multiprocessor. Schwab [23] has demonstrated additional performance benefits by utilizing two levels of parallelism.

We evaluated our implementations on three different architectures. We were primarily concerned with three issues: speedup, memory access patterns, and difficulty in programming and debugging. In the next four sections, we describe the machine architectures and the three parallel implementations in more detail. We sketch how each algorithm works and give performance statistics for the various machines. In the conclusion, we propose some general principles about symbolic computation on shared memory multiprocessors that we have learned from the examples we have considered.

2 Shared memory architectures

A wide variety of multiprocessor architectures have been described in the literature over the past few years. The characteristics of these architectures vary widely and are determined by the available technology, the class of applications addressed, the trade-offs between processor speed, memory system complexity and communication costs, and other design decisions. In this paper, we describe algorithms implemented on three shared memory architectures.

The Encore Multimax is a classic shared memory multiprocessor using a central bus for communication between processors and main memory. Each processor has a local cache and uses a snooping protocol to maintain cache coherency. The Multimax is suitable for medium and coarse grained parallel applications [13]. The particular machine used in these experiments had 16 National Semiconductor 32332 processors, each rated at roughly 2 MIPS, and 32 megabytes of shared memory. The local cache size is 64K bytes.

The RP3 is a large-scale research parallel processor developed at the T. J. Watson Research Center. The machine consists of a number of processor-memory elements connected by an Omega-interconnection network. Local memory references are handled immediately, while remote memory references must be resolved over the network. The architecture provides three types of memory pages: local pages, global pages, and replicated pages. The machine does not support automatic cache coherency; instead, cache management is the responsibility of the programmer or compiler [17]. The virtual memory interface allows each page to be marked as cacheable or non-cacheable, as well as allowing the cache to be flushed under user level control. The current version of the system has 64 ROMP processors. Each processor has 8 megabytes of storage, and a 64K byte local cache.

The Plus architecture [2] is a mesh of processor-memory elements connected by a deterministic routing network. Remote memory references are transparently routed to the appropriate destination. In addition several special features enhance this basic design. Local computation proceeds while remote writes are being completed, increasing overall throughput. When needed, a fence operation is used to stall the local processor until all remote writes have completed. Also, any page in memory may be replicated to other processing nodes. Reads from these replicated pages are handled locally, while writes are directed to the master copy of the page and transparently propagated to all replicated copies. Tuning an application to run on this architecture involves studying reference patterns of the algorithm and selecting a memory replication scheme that minimizes remote read references, while limiting remote writes and overall network traffic. This type of architecture is expected to perform better than a pure message-passing architecture because additional hardware supports the remote memory references, as opposed to a software layer processing these memory requests. The experiments described in the paper were run on the Plus simulator. A machine with 40 nodes is currently under construction.

These implementations were all written in C and use the C-Threads package [12], which allows parallel programming under the MACH operating system [1]. The programming model provided by C-Threads is one of many executing processes sharing a common global address space. Synchronization is provided through locks for mutual exclusion and conditions for waiting and signaling of events. The model is augmented on the non-uniform memory access machines to provide for the assignment of specific threads to specific processors, and for controlling the placement and replication policy of memory pages.

3 Parthenon: a resolution theorem prover

An area of symbolic computation where parallelism is just beginning to be applied is automated theorem proving. Most procedures for theorem proving are highly nondeterministic, since at each step of the procedure there are typically a number of inference rules which may be applied, and each inference rule can usually be applied to a large number of axioms. This leads to a combinatorial explosion in the search for a proof, and while attempts to limit this explosion have been partially successful, large search spaces still result.

Parthenon is a parallel theorem prover which uses Loveland's model elimination procedure [19]. Stickel recognized that model elimination is in fact very similar to Prolog's SLD resolution and used this observation in his Prolog Technology Theorem Prover (PTTP) [24]. By using sequential Prolog implementation techniques, he was able to obtain a very fast system. We have taken these ideas further by applying or-parallel Prolog implementation technology to achieve a fast parallel model elimination implementation.

3.1 Model elimination

The model elimination procedure is a variant of resolution with two inference rules. Given a set of input clauses, it can be used to prove that the clauses are unsatisfiable. Each inference rule operates on a chain, which is essentially a clause with certain literals regarded as special. We will refer to the special literals as framed literals and indicate them by placing them in a box. The inference rules are:

Extension To perform an extension operation on a chain, we must find an input clause which contains a literal that unifies with the rightmost literal in the chain. We then turn the rightmost literal of the chain into a framed literal and add the other literals of the input clause to the right of the chain, applying the unifying substitution.

Reduction If the rightmost literal of a chain unifies with a framed literal in the chain, we may delete the rightmost literal. Again we must apply the unifying substitution.

Framed literals at the right of a chain are always deleted. The procedure starts from one of the input clauses (which is regarded as a chain with no framed literals.)

As an example, we prove $\exists x\, p(x)$ from the single assumption $\exists a\, \forall y\, (\neg p(a) \rightarrow p(y))$. Negating the desired conclusion and putting the statements into clause form gives the two clauses $\neg p(x)$ and $p(a) \vee p(y)$. We start the model elimination procedure from the chain $\neg p(x)$.

1.	$\neg p(x)$	initial chain
2.	$\boxed{\neg p(a)} \vee p(y)$	extension, $x = a$
3.	$\Box$ (the empty clause)	reduction, $y = a$

3.2 Searching for a proof

In an or-parallel search, multiple processes can attempt to apply different inference steps in parallel with the restriction that if a conjunction must be proved, the individual conjuncts must be proved sequentially. In the context of the model elimination procedure, or-parallel search corresponds to trying the different extension and reduction possibilities for a chain in parallel. Since it is possible to have infinite chains of inferences which do not lead to a proof, it must be possible to guarantee that no inference is postponed indefinitely. One possible way to insure this would be to perform a breadth first search of all possible proofs, but the space required for this quickly becomes prohibitive. A better alternative is to perform a depth first search to a specific depth bound, and if no proof is found, to increase the bound and try again. This technique is called depth first iterative deepening. It would seem that this technique is very wasteful, since some parts of the tree are searched many times, but it is possible to show that it is within a constant factor of optimal. Furthermore, this constant factor is the average branching factor of the search. Hence, we can expect the speedup from parallelism to dominate this cost.

Each process in Parthenon performs a bounded depth first backtracking search of a different part of the search space. When a process exhausts its part of the space, it can obtain a new subtree from one of the other processes and continue searching there. A distributed scheduling algorithm handles the distribution of work to processes. In the remainder of this section, we give details on the or-parallel search, details of the scheduling algorithm, and some performance figures for Parthenon.

3.3 Representing the search space

The goal of Parthenon's inference mechanism is to make each individual resolution step as fast as possible. Like Stickel's PTTP, it exploits the similarity between model elimination and Prolog's SLD resolution to make model elimination efficient. In a Prolog system, each derived clause is represented by the inferences used to derive the clause. The individual inference steps are activation records on a stack, each describing which input clause was used in the inference step and what variable bindings were made in the input clause. To perform an inference step, a new activation record is added to the current stack of activation records, and variables are bound during unification. This process will often involve making new variable bindings in the original activation record chain, i.e., specializing some of the variables in the original clause. If the search fails at some point, other input clauses must be tried, and hence we must be able to undo this specialization. In order to accomplish this, another stack called the trail is used. Whenever a variable is bound, we record this fact in the trail and record the height of the trail in the activation record we are building. When backtracking, we pop activation records from the stack and unbind any variables indicated by the trail.

The model elimination extension step is almost exactly the same as Prolog's resolution step, and can be implemented in the above manner quite easily. The reduction operation is only slightly more expensive; it requires searching back through the stack of activation records to find the framed literals in the current chain.

When multiple processes can search in parallel, it is necessary to represent several chains simultaneously. We do this by having multiple stacks of activation records. In addition, if two chains were derived from a common ancestor chain, they will also share the activation records corresponding to that ancestor. This is done for two reasons.

1. There is never any need to copy an entire chain from some other process when performing an inference.

2. In order to ensure that inferences are not duplicated or omitted, it is necessary to have information about the possible inferences stored in a central location. Since this information is associated with each chain, it is natural to store it in the activation records and use locks in the activation records to coordinate access.

Because of this sharing of activation records, it may be the case that two chains with a common ancestor have incompatible bindings for some variables. Therefore, each process must be able to have its own set of bindings. This is accomplished by giving each process a semi-private binding array. This binding array corresponds directly to the variable stack in a Prolog implementation. When a process moves to a different part of the search tree, it must update this binding array in some manner. This is done by augmenting the trail to record the bindings of variables as well as which variables were bound. When a process wants to start working at an activation record, it uses the portion of the trail indicated by that record to fill in its binding array.

One key point is that almost all references made by a process can be considered local. Each binding array is associated with a single process, and the activation records and trail are stored in a distributed fashion as well. In particular, activation records and trail entries which are created by a process are created locally. The only time a process references nonlocal memory is when moving to a part of the search tree which was created by a different process. Measurements show that because of the large amount of work available in theorem proving contexts, this is very rare. Thus, we expect that Parthenon would run well on the heterogeneous shared memory machines currently being developed. We are currently experimenting with running Parthenon on the Plus simulator to verify this claim.

3.4 Coordinating the search

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[1] in many problems. Hence, simple distributed scheduling algorithms which have the first two properties may also satisfy the last two constraints. A more complicated distributed scheduling algorithm (such as those used in or-parallel Prolog systems [11]) or a global scheduling algorithm might be better able to assign processes to jobs when work is scarce. However, because of the nature of the problem, any advantage is likely to be slight. In addition, complicated distributed algorithms typically have more overhead than simple schemes, and global scheduling algorithms do not scale well.

In Parthenon, when a process searches for a job, it essentially performs a depth first, left to right traversal of the search tree. There are a few refinements to this basic strategy.

1. A process begins searching at its current node. Because long distance moves correlate with extensive updates to the binding array, it is best if a new job can be found close to the old one.
2. If a process finishes traversing the tree without having found a job, it repeats the search starting from the root of the tree.
3. A pure left to right traversal tends to result in situations where many processes are working in the left part of the tree and few are working in the right part. In order to achieve a better distribution of work, subtrees where there are few other processes should be preferred to subtrees where there are many other processes. To implement this, each node in the tree contains a count of the number of processes working in the subtree rooted at that node. When a process moves deeper in the tree, it moves to the subtree with the smallest count. Ties are broken by favoring nodes to the left.

Some care must be taken during traversal and modification 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 the node. Another error which must be prevented is having two processes both take the same job. To avoid such problems, each node in the tree contains a lock. The rules for using these locks are as follows.

[1]Often considerably greater than 2, unlike typical Prolog programs.

1. When checking for alternatives, creating a child node, etc., the current node must be locked.

2. When moving from the current node to one of its children or to its parent, both the current node and the destination node must be locked.

To avoid deadlocks, whenever a process must acquire two locks the node closest to the root of the tree is always locked first.

3.5 Implementation, performance and evaluation

We have tested Parthenon on a large number of examples used by Stickel. For the problems requiring more extensive searches, the inference rates show an almost linear speedup with the number of processes. This is probably because most theorem proving problems have relatively high branching factors. The data in this section represent single runs, but the measurements are repeatable.

Figure 1 gives speedup curves for the Multimax, and figure 2 gives speedup curves for the RP3. Each figure consists of two graphs, representing two different problems. Each graph has two curves which show the speedup in inferences per second (solid curve) and the speedup in execution times (dotted curve). For the RP3, the speedup values were calculated relative to the 10 or 20 process case, i.e., a factor of 2 speedup going from 10 to 20 or 20 to 40 processes was considered perfect. For the problem which was not run with 10 processes, we calculated the speedups by assuming that the 20 process case had a speedup of exactly 2. We should point out that the RP3 does not support cache coherency in hardware, and that Parthenon was originally written assuming 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 process. However, the RP3 implementation certainly does not take full advantage of the machine.

4 Boolean decision diagrams

An ordered boolean decision diagram [6] is an acyclic graph representation for boolean functions. Because this representation provides a canonical form (two functions are logically equivalent if and only if they have the same form) and is quite succinct in many cases, it has become widely used in CAD applications. However, the construction of boolean decision diagrams for certain large or particularly complex boolean functions can be very time consuming. In this section, we present a parallel algorithm for this task and describe its implementation.

4.1 Boolean decision diagrams and finite automata

Our approach to boolean decision diagrams uses some simple ideas from finite automata theory. If we fix an ordering of the variables, then an n-argument boolean function can be identified with the set of boolean vectors that make it true. For example, the function denoted by the boolean expression $x_1 \wedge x_2 + \neg x_2 \wedge x_3$ with the variable ordering $x_1 < x_2 <$

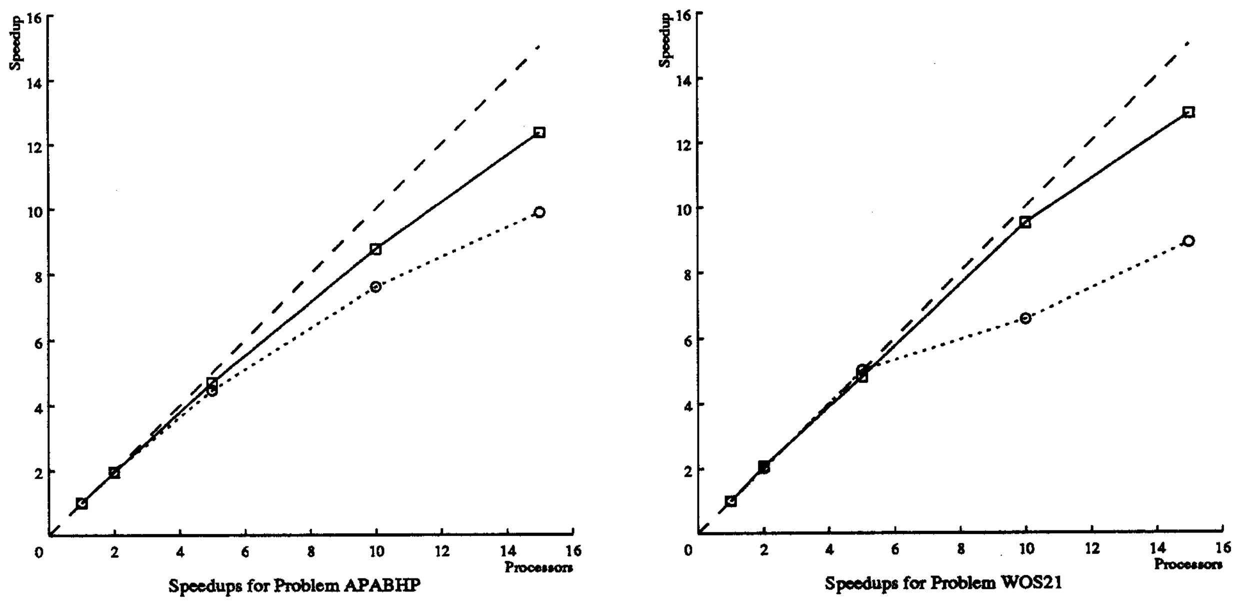

Figure 1: Multimax speedup curves for Parthenon

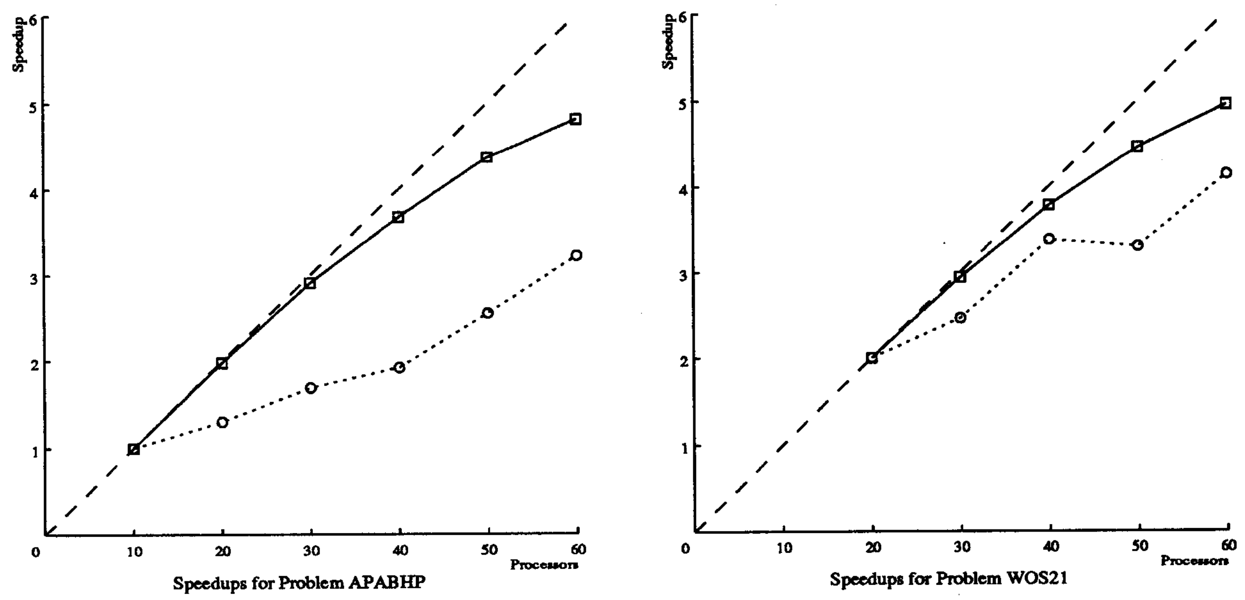

Figure 2: RP3 speedup curves for Parthenon

x_3 is uniquely represented by the set of vectors $\{(1,1,0),(1,1,1),(0,0,1),(1,0,1)\}$. The corresponding set of strings $\{110, 111, 001, 101\}$ is a finite language. Since all finite languages are regular, there is a minimal finite automaton that accepts this set. This automaton provides a canonical representation for the original boolean function. Since each node in the state-transition graph for a boolean function will have at most two successors (one for 0 and one for 1), we can view this graph as a boolean decision diagram for the function.

Logical operations on boolean functions can be implemented by set operations on the languages accepted by the finite automata: logical *and* corresponds to set intersection, logical *or* corresponds to set union, and logical *not* corresponds to set complement (with respect to the set of all boolean strings of length n.) Standard constructions from elementary automata theory are used to perform these set operations. For example, let $M_1 = (Q_1, \{0, 1\}, \delta_1, q_0^1, F_1)$ and $M_2 = (Q_2, \{0, 1\}, \delta_2, q_0^2, F_2)$ be the boolean decision diagrams for two n-variable boolean functions f_1 and f_2. Then an automaton for $f_1 \wedge f_2$ is given by $M = ((Q_1 \times Q_2) \cup \{\perp\}, \{0, 1\}, \delta_\wedge, (q_0^1, q_0^2), F_1 \times F_2)$, where $\perp$ denotes the sink state ($\delta_\wedge(\perp, 0) = \delta_\wedge(\perp, 1) = \perp$) for the product automaton. $\delta_\wedge$ is defined as

$$\delta_\wedge((q_1, q_2), a) = \begin{cases} (\delta_1(q_1, a), \delta_2(q_2, a)) & \\ \qquad \text{if } \delta_1(q_1, a) \neq \perp_1 \text{ and } \delta_2(q_2, a) \neq \perp_2 & \\ \perp \quad \text{otherwise} & \end{cases}$$

$\perp_1$ and $\perp_2$ are sink states of M_1 and M_2 respectively. An example of this procedure is shown in figure 3. In this figure, M_1 corresponds to $(\neg x_1 \wedge \neg x_3 \vee x_1)$, and M_2 corresponds to $(\neg x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$. The result of the logical *and* operation is $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2)$.

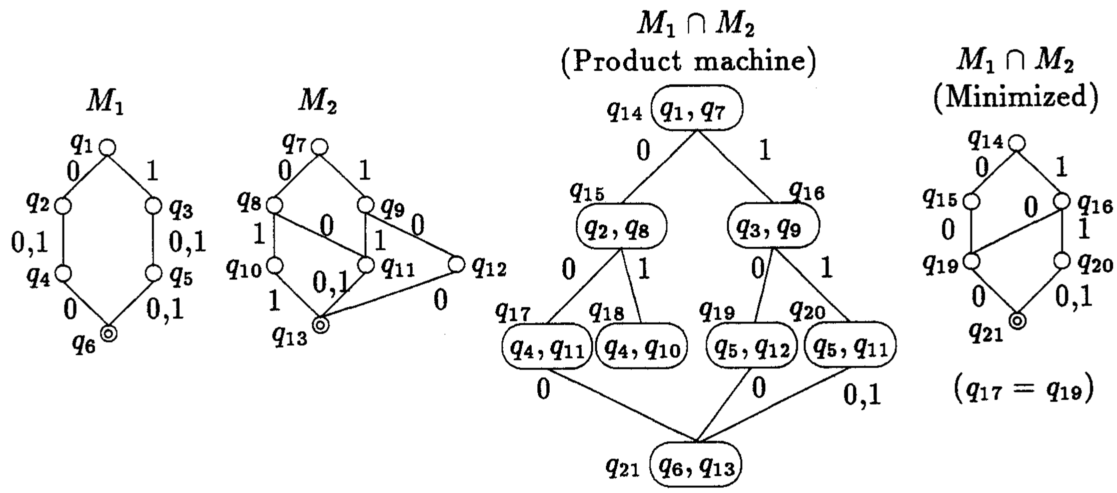

Figure 3: Product automaton generation for boolean decision diagrams

Automata for the other logical operations can be constructed in a similar manner. In general, determining the state set of the finite automaton for each of these operator involves a product construction. Also, in each case the resulting automaton may not be minimal. For this reason, a final minimization stage is needed after the product construction.

4.2 Implementation

We first describe the implementation of a single operation. Each operation consists of a product construction phase followed by a minimization phase. The construction of a product automaton begins with its initial state, i.e., the pair consisting of the initial states of the two argument automata. The successors of this state are determined for the inputs 0 and 1. As each pair of states is generated, it is entered into a queue of states to expand and into a hash table of generated pairs. When a pair is about to be produced, we check the hash table to see if it already exists. Pairs are removed from the queue and expanded in this manner until the queue becomes empty. In the minimization phase, states are processed starting at the lowest level and working upward. For each state, we hash its two successors and check a global hash table to determine if a state with those successors already exists. If not, the pair of successors is entered into the hash table along with a pointer to the state, and the state is returned. If such a state does exist, that state is returned. This procedure guarantees both that the result of an operation is a minimal automaton and that there are no equivalent states generated by operations which proceed in parallel. This last property is essential to control the storage requirements of the algorithm.

The above process of machine composition and state minimization is repeated once for each operation in the boolean formula. Some of these operations will be independent and may be performed in parallel. For example, to construct the boolean decision diagram for $f_1 \wedge f_2$, we may construct the decision diagrams for f_1 and f_2 in parallel before performing the final conjunction. More precisely, given a formula for which we wish to build the boolean decision diagram, the first step is to determine the level of each node in the parse tree for the formula. The leaf nodes of the tree are input variables; the non-leaf nodes correspond to the boolean operators that occur in the formula. The level of each node is determined by the rule:

1. The level of an input variable is 0.

2. The level of a non-leaf node is $max(l_1, l_2) + 1$, where l_1 and l_2 are levels of its operands.

The construction of the boolean decision diagram begins with level 0 nodes, which can be constructed immediately. In general, we can process a level i node as soon as all the level 1, 2, ..., $i - 1$ nodes beneath it have been processed. Operations at the same level in the tree can be performed in parallel, since they do not conflict.

The levels at the top of the parse tree have only a few operations that can be performed in parallel. In order to extract more parallelism, we divide operations on such levels into several sub-operations. This can be done using Shanon's expansion:

$$f(x_1, \ldots, x_n) = \neg x_1 \wedge f(0, x_2, \ldots, x_n) \vee x_1 \wedge f(1, x_2, \ldots, x_n).$$

In the product and minimization phases, the 0 and 1 successors of the initial pair are generated. This corresponds to producing the initial states for $f(0, x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n)$. Then the product and minimization phases are performed (in parallel) for these two functions. We need one last merge step for the root state to complete the operation. Note that an operation can be decomposed further if needed using the above expansion.

# of processors	1	2	3	4	5	8	11	14	15
time (seconds)	1465.8	732.1	499.9	384.1	311.4	211.8	171.6	148.5	140.6

Table 1: BDD construction times for a 10 bit multiplier on the Multimax

An example of this procedure is shown in figure 4. This example is the same as that depicted in figure 3. To perform the operation, processor P_1 expands the 0 and 1 successors of the initial pair. Processor P_2 takes the 0 successor (q_2, q_8), generates the product automaton and minimizes it. Processor P_3 takes the 1 successor and does the same thing. After P_2 and P_3 have completed the minimization phase for their product automata, processor P_1 generates q_{14}.

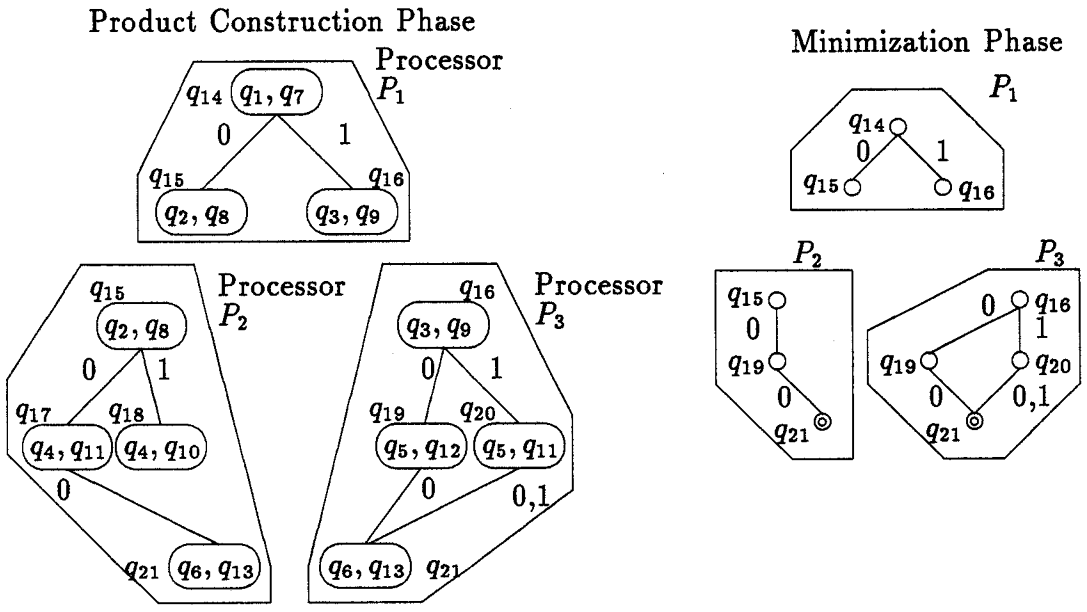

Figure 4: Decomposition of an operation for boolean decision diagrams

4.3 Performance evaluation

We experimented with building the boolean decision diagrams for boolean multipliers to evaluate the program. The decision diagrams for multipliers are known to grow quite rapidly (exponentially in the size of the operands, in fact.) Table 1 shows the execution time on the Multimax to construct decision diagrams for a multiplier with 10 bits (20 boolean variables). In the construction, there are 1726 operations (*and*: 527, *or*: 562, *not*: 0, *xor*: 307, *merge*: 330).

The execution time for a single processor is roughly the same as for the (sequential) program for constructing boolean decision diagrams described by Fisher and Bryant [14].

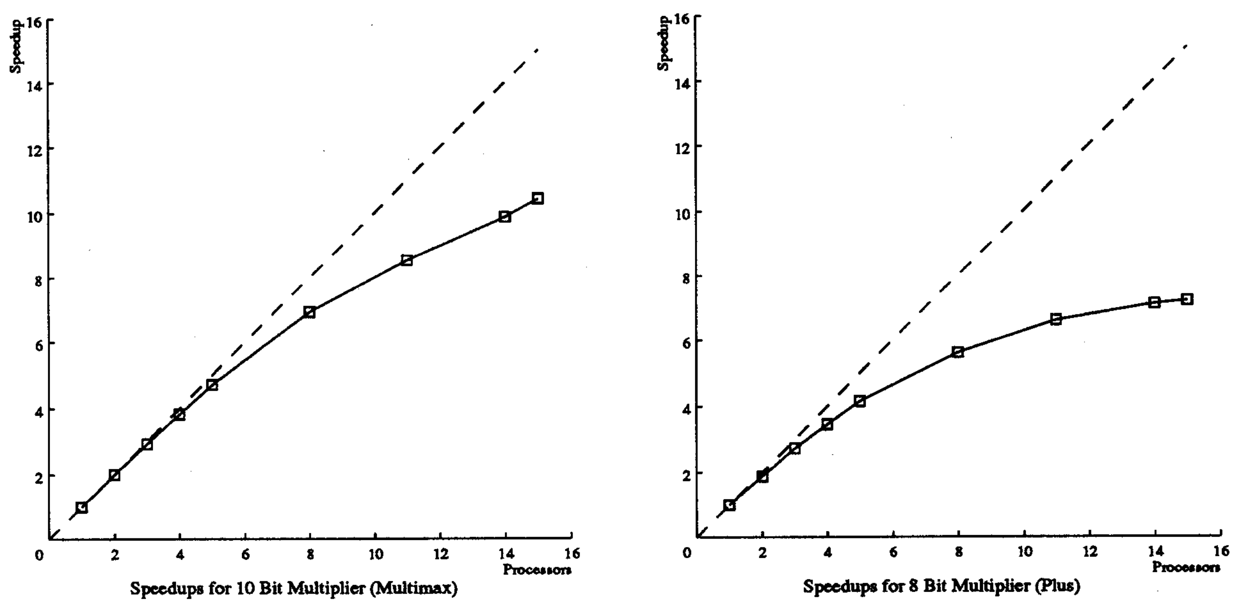

Figure 5: Speedup curve for 8-bit and 10-bit multiplier BDD construction

The graph in figure 5 shows the execution time speedup curve on the Multimax and Plus machines.

4.4 Manipulation of large decision diagrams

Even on a large machine, in some examples the number of states required is too large to fit in available memory. For example, we were originally unable to construct a boolean decision diagram for a 13-bit multiplier. To overcome this problem, we use the above decomposition to split the task into manageable pieces. Since the original parallel algorithm guarantees high-speed execution, each part of the boolean formula can be processed in reasonable time, and the total execution time is also reasonable. We were able to use this method to avoid running out of memory when it comparing the decision diagrams for two 16-bit multipliers. We divided the problem into 2048 parts. Each part required about 800,000 states for the decision diagrams took about 220 seconds to complete. The total execution time was about 100 hours.

5 Gröbner bases

Gröbner bases are one of the basic tools of computational algebraic geometry, the branch of mathematics which deals with the solution of sets of algebraic equations. Gröbner bases give a normal form to ideals in polynomials rings. Once a Gröbner basis is found for the corresponding ideals, it becomes easy to test if a polynomial belongs to an ideal, if two ideals are equal, if an ideal is contained in another, and so on. The interested reader is referred to Buchberger [7, 9], where one will find a complete introduction to Gröbner bases, a description of their computation, as well as a list of many applications. A review of the various attempts to parallelize the algorithm can be found in [25].

The algorithm transforms a set of polynomials into another set of polynomials, a Gröbner basis generating the same ideal. These polynomials are rational polynomials over some finite set of variables. While the theory behind the algorithm is quite complex, the actual computation is relatively simple to describe. This implementation is based on an algorithm given by Gebauer and Möller [16].

5.1 The basic algorithm

The primitive step of the computation, called reduction, involves deleting the first term of the current working polynomial, called the S-polynomial, by subtraction of an appropriate multiple of a polynomial already in the set. The result of this reduction is of lower order, but not necessarily smaller. A polynomial is totally reduced with respect to a set of polynomials when no further reduction is possible.

The S-polynomial of two polynomials already in the set is computed by finding the least common multiple of the leading terms, multiplying by appropriate monomials to make the two leading terms equal, and then finding the difference of these polynomials. Intuitively, each polynomial is scaled by the smallest factor such that the lead terms become equal; subtracting them cancels out the lead terms.

```
$Spol(P, Q) := \{$
    $G := LCM(LeadingTerm(P), LeadingTerm(Q))$
    $M_p := G / LeadingTerm(P)$
    $M_q := G / LeadingTerm(Q)$
    $Spol := M_p \cdot P - M_q \cdot Q$
$\}$
```

```
$Spol(2x^2y + 3xy + 1, 3xy^2 + xy + 2y + 2) :=$
    $G = LCM(2x^2y, 3xy^2) = 6x^2y^2$
    $M_p = 6x^2y^2/2x^2y = 3y,\ M_q = 6x^2y^2/3xy^2 = 2x$
    $Spol = 3y \cdot (2x^2y + 3xy + 1) - 2x \cdot (3xy^2 + xy + 2y + 2)$
        $= -2x^2y + 9xy^2 - 4xy - 4x + 3y$
```

The simplest form of the algorithm can now be stated. Termination is guaranteed by restrictions on the ordering of terms in each polynomial. For our discussion, it is enough that such orderings exist.

The algorithm begins with the input set B, and constructs the set P of all pairs of distinct polynomials in B. Then, while pairs remain, the algorithm processes a pair from P. This processing consists of computing the S-polynomial of the pair, and then reducing it with the polynomials in B until no further reductions are possible. Sometimes, the S-polynomial reduces all the way to 0, in which case the algorithm proceeds to the next iteration of the loop with no additional work. However, if the S-polynomial reduces to a non-zero polynomial, then this new polynomial is added to the set B. In addition, the new polynomial is paired with each polynomial already in B, and these additional pairs are added to the set P. Eventually, the set of pairs is empty, and the algorithm terminates. It should be noted that while B is a Gröbner basis upon termination, a follow-up step to transform B into a reduced Gröbner basis is often needed. We have not addressed that step in this paper, although some of the techniques presented here can also be applied.

Of course, the order in which the pairs are selected is very important to the practical efficiency of the algorithm. One heuristic is to select the pair whose S-polynomial has the smallest leading term under the selected order. In addition, several tests have been developed to delete pairs whose S-polynomial will reduce to zero. These criteria can be found in [8, 16] and involve testing for certain constraints on the leading terms. The order in which reductions are performed is less well defined; in this implementation we use the order in which the polynomials first appear when searching for a reducing polynomial. Most of the computation time is spent in the S-polynomial and Reduction steps.

The typical computation described below consists of tens to hundreds of S-polynomial computations and hundreds to thousands of reduction steps. Because of the very rapid growth in the actual complexity of the problem, larger inputs will tend to produce polynomials with more terms and larger coefficients. In our sample executions, we observed that the total number of S-polynomials actually reduced does not grow as $|B|^2$, but at a somewhat greater than linear rate.

5.2 Coarse grain parallelism

The coarse grain parallelism in this algorithm consists of expanding and reducing several S-polynomials in parallel. A lock is used to ensure mutually exclusive access to the set of pairs P and the set of polynomials B. One potential bottleneck is access to these shared structures, which is minimal in the actual implementation. A greater problem, for small examples, is that processors may be left idle. This occurs because only a few pairs are produced at each stage in the algorithm. Also, in the parallel case, some pairs are expanded that would have been deleted in the sequential case. Typical data runs vary over a wide range, with small inputs taking milliseconds and modest ones hundreds or thousands of seconds on a sequential 2-MIPS processor. Other examples show greater than linear speedup, due to early reduction of pairs that prune part of the search space. As expected, larger inputs benefit the most from larger numbers of processors. However, inputs with very few large polynomials, or which produce only a limited number of pairs at any stage in the computation, benefit less from additional processors. Figure 6 presents the speedup versus number of processors for two examples from the research literature. (Box indicates Encore Multimax, Circle indicates IBM RP3, Triangle indicates Plus Simulator). On a single Multimax processor, the Rose example requires about 35 seconds, while the Trinksl example requires only 10 seconds.

5.3 Fine grain parallelism

At the heart of the Gröbner basis computation is the operation of reduction, in which one polynomial is used to eliminate the most significant term from an initial polynomial. Speeding up this step is critical to improving the performance of the algorithm.

The source of fine-grain parallelism in the algorithm occurs in the reduction loop. The key observation of Melenk and Neun [21] is that the algorithm only branches based on the

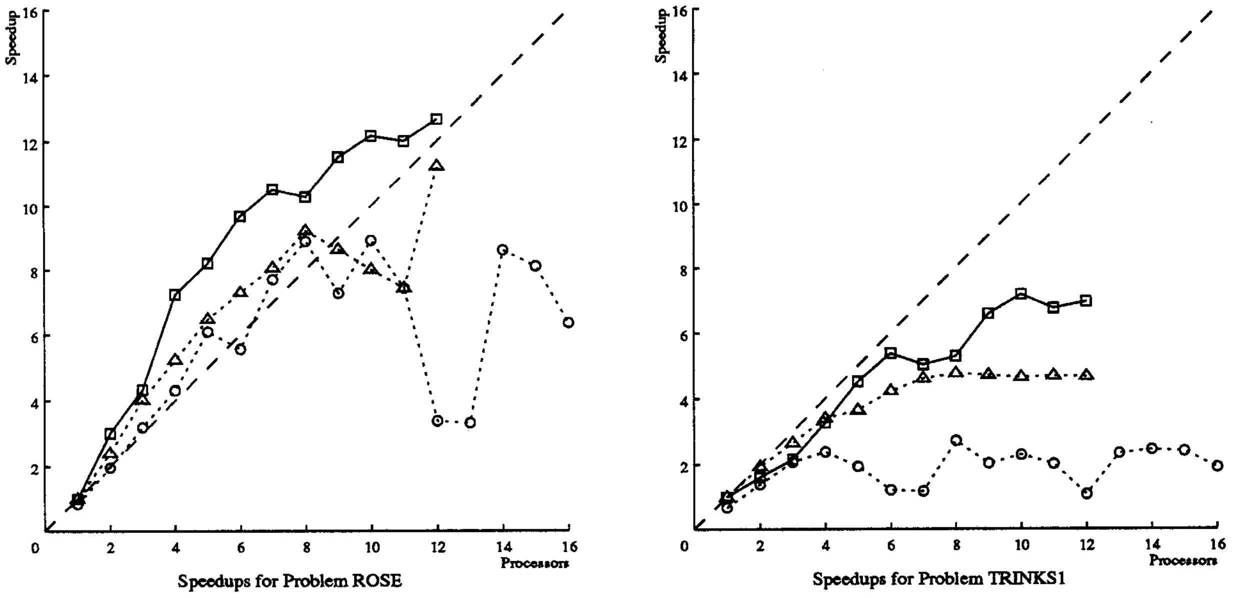

Figure 6: Gröbner Bases speedup curves

leading power product of the polynomials. This means that as soon as the first term of any polynomial is computed, the next step of the algorithm may begin concurrently with the computation of the remainder of the polynomial. We have implemented this approach explicitly in the program, as opposed to implicitly in a dataflow style as Melenk and Neun described.

The reduction loop consists of searching the list of polynomials for a reducing polynomial, then setting up the reduction, and finally performing it term by term. The critical observation here is that the next iteration of the loop will examine the terms in the same order as they are produced.

In order to exploit this parallelism, this implementation stores two synchronization fields with the polynomial. A size field keeps track of the number of terms and a done flag indicates that this polynomial is complete or still under construction. The reduction process begins as in the sequential case with an examination of the leading term and a search for a reducing polynomial. Then before actually computing the reduction, a second processor is assigned the task of performing the next loop iteration. The second processor busy waits, checking for either a new term or a done indication, while the first processor computes the polynomial term by term. When the first processor finishes, it will wait once more in a queue to be assigned additional work. The second processor, meanwhile, waits for a term to be produced. Once a term is produced, the polynomial is definitely known to be non-zero, and the second processor will search for another reducing polynomial. If the search is successful, the second processor will assign a third processor to consume the next intermediate result, and the process repeats. If no terms are produced, and the done flag is set, then the second processor has detected a zero polynomial. There is no need to update the pairs set, and a new pair may be expanded. If the search for another reduction fails the second processor begins the update-pairs routine, in which useless pairs are deleted and the new pairs are added. This overlaps with the previous computation. This processor then begins expansion of the next pair. The S-polynomial expansion may be forced to wait because one of the polynomials in the pair may still be incomplete. The code to compute the S-polynomials uses the same synchronization technique to consume terms as they are produced, overlapping additional computation and providing better speedup.

There are several issues to point out about this pipelined reduction process. (1) The busy waiting loops all spin on two cached variables. This means that the busy waiting does not saturate the bus and slow other processors down. (2) Once the reduction begins, each processor in the pipeline consumes terms at approximately the same speed as the previous processors produces them; therefore very little time is spent busy-waiting in the middle of a reduction pipeline. (3) Because the synchronization constraints flow in one direction only, no synchronization primitives are used beyond strong ordering of reads and writes. (4) There is no speculative parallelism in this part of the computation. Each reduction must be performed, and this approach simply overlaps the computation performed by the inner loop as much as possible.

5.4 Combining the two techniques

The two forms of parallelism described above combine in a complementary fashion. The coarse-grain method performs many reductions in parallel; the fine-grain method parallelizes the reduction chains. We experimented with a number of processor scheduling mechanisms before choosing the one used in the implementation. One version used a single work queue, another separate queues with locks, and the final version uses a fixed ordering among processors. We expect that the two forms of parallelism are independent, since performing the pair expansions in parallel, only faster, reduces the average time between pair expansion and updating the set of pairs. Because of this speedup, the fraction of time spent in the critical section updating the shared data increases. We believe that this will not become a bottleneck until hundreds of processors are used, but varying ratios of processor to synchronization speed across different architectures may make this an important consideration.

In figure 7 below, we present performance results for the full algorithm with both levels of parallelism. (Box indicates Encore Multimax, Triangle indicates Plus Simulator). The Rose example is large enough to benefit from the fine-grain parallelism, while the Trinks1 example is too small to speed up using this approach. The difference between these two problems is that the polynomials that appear in the first example tend to have 20 or more terms, while the second example has polynomials with 10 or fewer terms. The fine-grain parallelism only works when there are many terms to process in each polynomial. The Rose example seems to be more typical of the type of problems which arise in practice. A large multiprocessor will be able to operate on polynomials with hundreds of terms, allowing even better speed up from the fine-grain parallelism.

6 Conclusion

We observed essentially linear speedup for all three algorithms on many examples. There were also small examples where virtually no speedup was obtained regardless of the number

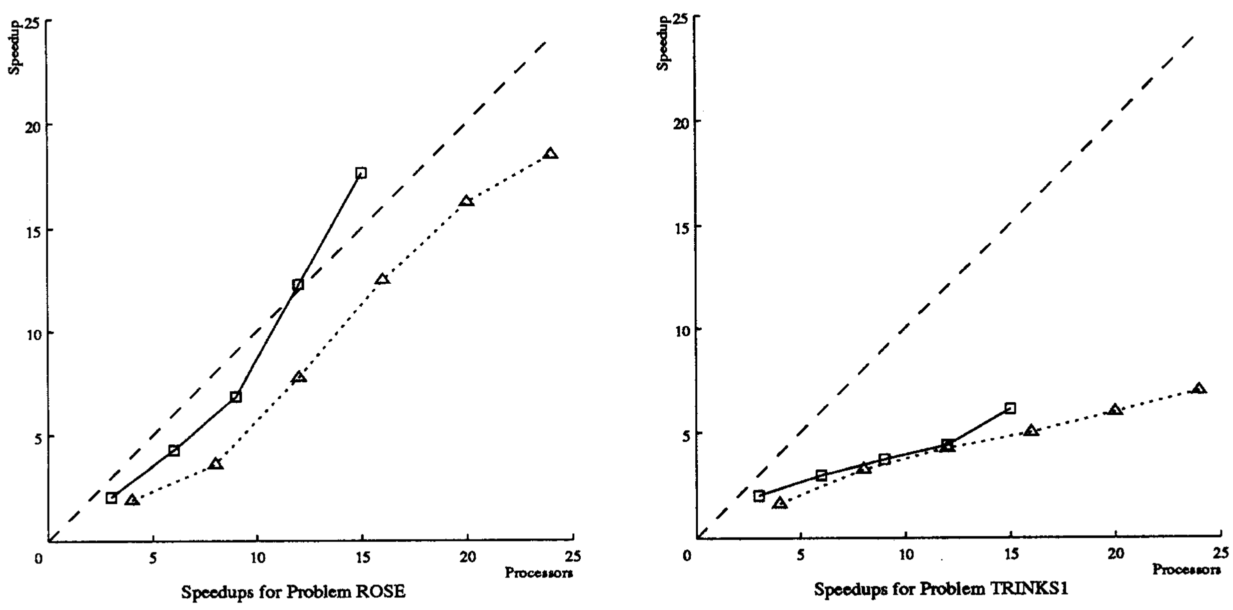

Figure 7: Gröbner Bases speedup curves

of processors used. Furthermore, we occasionally observed superlinear speedup in Parthenon and the Gröbner bases algorithm. This is a well-known artifact of parallel tree searches.

There are two main difficulties which arise in parallel implementations of symbolic algorithms. The first is that it is essentially impossible to implement any type of static scheduling. Given an instance of the problem, there is no way to predict the number of steps that the algorithm will require, the types of steps that will be needed, or the amount of work which will be available at any point in time. In order to get good speedup, it is often necessary to resort to speculative parallelism; trying to interrupt processors which are found to be doing useless work can further increase the complexity of the scheduling problem.

The second problem is the difficulty of debugging a parallel implementation. The scheduling problem contributes greatly to this. In addition, symbolic algorithms often must deal with numerous cases. When many processors are operating in parallel, the number of combinations of cases which can be active at a given time, and hence the number of possible interactions, is large. Because of this and the nondeterminism associated with the scheduling, it is not unusual to encounter bugs which only occur once every hundred runs and only in large problems.

We feel shared memory machines have some advantages over message passing machines as platforms for implementing symbolic algorithms. First, the implementation on a shared memory machine should be easier. On a message passing machine, references to remote data must either be handled by a separate thread running at the remote node or by the application itself, typically at the beginning of a main loop. In the first case, the code for synchronization is likely to be just as complex as on a shared memory machine, and the message handling itself is usually nontrivial. In the second case, there is a loss of parallelism, and the application becomes more monolithic. In addition, having to check explicitly whether references are local or remote increases the complexity of the implementation.

Second, the software overhead of handling messages would probably make it difficult to obtain good performance. Typical symbolic algorithms access many small objects scattered throughout memory as they work, so there is little opportunity to use large block transfers; instead, it is message latency which is critical. Experiments with Parthenon and the Gröbner basis algorithm on the Plus simulator indicate that about one percent of references are remote. On the Plus, this would mean about twenty percent of the execution time would be spent waiting for the network. Another order of magnitude of overhead for remote references would seriously degrade the performance of these applications.

Our experiences lead us to believe that it is possible to develop useful parallel implementations of symbolic algorithms. We also feel that these implementations can usually be structured so that almost all their memory references can be considered local. Thus, they should map extremely well onto the non-uniform shared memory machines currently being developed. We hope to demonstrate this in the near future on the Plus machine which is currently being built.

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] R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual Symposium on Computer Architecture.* IEEE Computer Society Press, June 1990.

[3] S. Bose, E. M. Clarke, D. E. Long, and S. Michaylov. Parthenon: A parallel theorem prover for non-horn clauses. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, 1989.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *24th Design Automation Conference*, 1987.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] Bruno Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal (German).* PhD thesis, University of Innsbruck, 1965.

[8] Bruno Buchberger. A criterion for detecting unnecessary reductions in the construction of Gröbner bases. In *Proceedings of EUROSAM 79, Lectures Notes in Computer Science 72*, pages 3–21, 1979.

[9] Bruno Buchberger. Gröbner bases : An algorithmic method in polynomial ideal theory. In N.K.Bose, editor, *Recent Trends In Multidimensional Systems Theory*, pages 184–232. D.Reidel Publishing Company, 1985.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of 5th Symposium on Logic in Computer Science*, 1990.

[11] R. Butler, T. Disz, R. Overbeek, and R. Stevens. Scheduling or-parallelism: An Argonne perspective. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605, Cambridge, 1988. MIT Press.

[12] E. C. Cooper. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, Pittsburgh, PA 15213, June 1988.

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

[14] Allan L. Fisher and Randal E. Bryant. Performance of COSMOS on the IFIP workshop benchmarkes. In *Proceedings of IMEC Conference*, 1989.

[15] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvement of boolean comparison method based on binary decision diagrams. In *1989 IEEE Conference on Computer-Aided Design*. IEEE Computer Society Press, 1989.

[16] Rüdiger Gebauer and Michael Möller. On an installation of Buchberger's algorithm. *Journal of Symbolic Computation*, 6(2 & 3):275–286, 1988.

[17] IBM. *Research Parallel Processor Prototype Principle of Operations.*

[18] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. Submitted for Publication, February 1990.

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

[20] W. McCune. *OTTER: Other Techniques in Theorem Proving.* Argonne National Laboratory.

[21] Herbert Melenk and Winfried Neun. Parallel polynomial operations in the large Buchberger algorithm. In *Computer Algebra and Parallelism, Workshop at the TIM3 Laboratory, University of Grenoble, France.* Academic Press, London, June 1988.

[22] Winfried Neun and Herbert Melenk. Implmentation of the LISP–arbitrary precision arithmetic for a vector processor. In *Computer Algebra and Parallelism, Workshop at the TIM3 Laboratory, University of Grenoble, France.* Academic Press, London, June 1988.

[23] S. A. Schwab. Extended parallelism in the Gröbner basis algorithm. Technical report, Carnegie Mellon University, 1991. In preparation.

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

[25] Jean Philippe Vidal. The computation of Gröbner bases on a shared memory multiprocessor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, August 1990.

[26] D. H. D. Warren. Or-parallel execution models of Prolog. Technical report, Department of Computer Science, University of Manchester, 1987.