Using Hundreds of Workstations to Solve First-Order Logic Problems

Alberto Maria Segre & David B. Sturgill
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
{segre,sturgill}@cs.cornell.edu

Abstract

This paper describes a distributed, adaptive, first-order logic engine with exceptional performance characteristics. The system combines serial search reduction techniques such as bounded-overhead subgoal caching and intelligent backtracking with a novel parallelization strategy particularly well-suited to coarse-grained parallel execution on a network of workstations. We present empirical results that demonstrate our system's performance using 100 workstations on over 1400 first-order logic problems drawn from the "Thousands of Problems for Theorem Provers" collection.

Introduction

We have developed an distributed, adaptive, first-order logic engine as the core of a planning system intended to solve large logistics and transportation scheduling problems (Calistri-Yeh & Segre, 1993). This underlying inference engine, called DALI (*Distributed, Adaptive, Logical Inference*), is based on an extended version of the Warren Abstract Machine (WAM) architecture (Aït-Kaci, 1991) which also serves as the basis for many modern Prolog implementations. DALI takes a first-order specification of some application domain (the *domain theory*) and uses it to satisfy a series of queries via a model elimination inference procedure. Our approach is inspired by PTTP (Stickel, 1988), in that it is based on Prolog technology (*i.e.*, the WAM) but circumvents the inherent limitations thereof to provide an inference procedure that is complete relative to first-order logic. Unlike PTTP, however, DALI employs a number of serial search reduction techniques such as bounded-overhead subgoal caching (Segre & Scharstein, 1993) and intelligent backtracking (Kumar & Lin, 1987) to improve search efficiency. DALI also exploits a novel parallelization scheme called *nagging* (Sturgill & Segre, 1994) that supports the effective use of a large number of loosely-coupled processing elements.

The message of this paper is that efficient implementation technology, serial search reduction techniques, and parallel nagging can be successfully combined to produce a high-performance first-order logic engine. We support this claim with an extensive empirical performance evaluation.

The DALI System

The basis of our implementation is the WAM. The WAM supports efficient serial execution of Prolog: the core idea is that definite clauses may be compiled into a series of primitive instructions which are then interpreted by the underlying abstract machine. The efficiency advantage of the WAM comes from making compile-time decisions (thus reducing the amount of computation that must be repeated at run time), using carefully engineered data structures that provide an efficient scheme for unwinding variable bindings and restoring the search state upon backtracking, and taking several additional efficiency shortcuts, which, while acceptable for Prolog, are inappropriate for theorem proving in general.

In our implementation, the basic WAM architecture is extended in three ways. First, we provide completeness with respect to first-order logic. Next, we incorporate serial search reduction techniques to enhance performance. Finally, we employ a novel asynchronous parallelization scheme that effectively distributes the search across a network of loosely-coupled heterogeneous processing elements.

First-Order Completeness

As described in (Stickel, 1988), Prolog — and the underlying WAM — can be used as the basis for an efficient first-order logic engine by circumventing the following intrinsic limitations: (*i*) Prolog uses unsound unification, *i.e.*, it permits the construction of cyclic terms, (*ii*) Prolog's unbounded depth-first search strategy is incomplete, and (*iii*) Prolog is restricted to definite clauses. PTTP demonstrates how these three limitations can be overcome without sacrificing the high inference rate common to many Prolog implementations.

Like Stickel, we repair Prolog's unsound unification by performing the missing "occurs check." We also borrow a compile-time technique from (Plaisted, 1988) to "flatten" unification and perform circularity checking only when needed. In our implementation, the circularity check is handled efficiently by a new WAM instruction. We restore search completeness by using a depth-first iterative deepening search strategy (Korf, 1985) in the place of depth-first search.[1] Finally, the definite-clause restriction is lifted by adding the model elimination reduction operation to the familiar Prolog resolution step and by compiling in all contrapositive versions of each domain theory clause. As suggested in (Stickel, 1988), the use of the model elimination reduction operation enables the inclusion of cycle detection with little additional programming effort (cycle detection is a pruning technique that reduces redundant search).

Serial Search Reduction

Our second set of modifications to the WAM support a number of *adaptive inference techniques*, or serial search reduction mechanisms. In (Segre & Scharstein, 1993) we introduce the notion of a bounded-overhead subgoal cache for definite-clause theorem provers. Bounded-overhead caches contain only a fixed number of entries; as new entries are made, old entries are discarded according to some preestablished cache management policy, *e.g.*, *least-recently used*. Limiting the size of the cache helps to avoid thrashing, a typical consequence of unbounded-size caches operating within bounded physical memory. Cache entries consist of successfully-proven subgoals as well as subgoals which are known to be unprovable within a given resource limit; matching a cache entry reduces search by obviating the need to explore the same search space more than once. As a matter of policy, we do not allow cache hits to bind logical variables. In exchange for a reduction in the number of cache hits, this constraint avoids some situations where taking a cache hit may actually increase the search space. Cache entries are allowed to persist until the domain theory changes; thus, information acquired in the course of solving one query can help reduce search on subsequent queries.

In a definite-clause theory, the satisfiability of a subgoal depends only on the form of the subgoal itself. However, when the model elimination reduction operation is used, a subgoal's satisfiability may also depend on the ancestor goals from which it was derived. Accordingly, a subgoal may fail in one situation while an identical goal may succeed (via the reduction operation) elsewhere in the search. DALI extends the definite-clause caching scheme of (Segre & Scharstein, 1993) to accommodate the context sensitivity of cached successes. The DALI implementation presented here simply disables the caching of failures in theories not composed solely of definite clauses, although this is unnecessarily extreme.

In addition to subgoal caching, we employ a form of intelligent backtracking similar to that of (Kumar & Lin, 1987). Normally, the WAM performs chronological backtracking, resuming search from the most recent OR choicepoint after a failure. Naturally, this new search path may also fail for the same underlying reason as the previous path. Intelligent backtracking attempts to identify the reasons for a failure and backtrack to the most recent choicepoint that is not doomed to repeat it. Our intelligent backtracking scheme requires minimal change to the WAM for the definite-clause case. Briefly, choicepoints along the current search path are marked at failure time depending on the variables they bind; unmarked choicepoints are skipped when backtracking. As with subgoal caching, the marking procedure must also take ancestor goals into account when deciding which choicepoints to mark.

Nagging

In (Sturgill & Segre, 1994) we introduce a parallel asynchronous search pruning strategy called *nagging*. Nagging employs two types of processes; a *master* process which attempts to satisfy the user's query through a sequential search procedure and one or more *nagging* processes which perform speculative search in an effort to prune the master's current search branch. When a nagging process becomes idle it requests a snapshot of its master's state as characterized by the variable bindings and the stack of open goals. The nagging process then attempts to prove a permuted version of this goal stack under these variable bindings using the same resource limit in effect on the master process. If the nagging process fails to find a proof, it guarantees that the master process will be unable to satisfy all goals on its goal stack under current variable bindings. The master process is then forced to backtrack far enough to retract a goal or variable binding that was rejected by the nagger. If, however, the nagging process does find a proof, then it has satisfied a permuted ordering of all the master's open goals, thereby solving the original query. This solution is then reported.

[1] Unlike PTTP's depth metric which is based on the number of nodes in the proof, our depth-first iterative deepening scheme measures depth as the height of the proof tree. Although each depth measure has its advantages and neither leads to uniformly superior performance, our choice is motivated by concerns for compatibility with both our intelligent backtracking and caching schemes.

Nagging has many desirable characteristics. In particular, it affords some opportunity to control the granularity of nagged subproblems and is also intrinsically fault tolerant. As a result, nagging is appropriate for loosely-coupled hardware. Additionally, nagging is not restricted to definite-clause theories and requires no extra-logical annotation of the theory to indicate opportunities for parallel execution. Finally, nagging may be cleanly combined with other parallelism schemes such as OR and AND parallelism. Readers interested in a more complete and general treatment of nagging are referred to (Sturgill & Segre, 1994).

Evaluation

We wish to show that (*i*) subgoal caching, intelligent backtracking, and nagging combine to produce superior performance, and (*ii*) our approach scales exceptionally well to a large numbers of processors. In order for our results to be meaningful, they should be obtained across a broad spectrum of problems from the theorem proving literature. To this end, we use a 1457-element subset of the 2295 problems contained in the *Thousands of Problems for Theorem Provers* (TPTP) collection, release 1.0.0 (Suttner *et al.*, 1993). The TPTP problems are expressed in first-order logic: 37% are definite-clause domain theories, 5% are propositional (half of these are definite-clause domain theories), and 79% require equality. The largest problem contains 6404 clauses, and the number of logic variables used ranges from 0 to 32000. For our test, we exclude 838 problems either because they contained more than one designated query clause, or, in one instance, due to a minor flaw in the problem specification.

Four different configurations of the DALI system are applied to this test suite; three are serial configurations, while the fourth employs nagging. Each configuration operates on identical hardware and differs only in which serial search reduction techniques are applied and in whether or not additional nagging processors are used. We use a single, dedicated, Sun Sparc 670MP "Cypress" system with 128MB of real memory as the main processor for each tested configuration. Nagging processors, when used, are drawn from a pool of 110 additional Sun Sparc machines, ranging from small Sparc 1 machines with 12MB of memory to additional 128MB 670MP processors running SunOS (versions 4.1.1 through 4.1.3). These machines are physically scattered throughout two campus buildings and are distributed among three TCP/IP subnets interconnected by gateways. Note that none of the additional machines are intrinsically faster than the main processor; indeed, the majority have much slower CPUs and far less memory than does the main processor. Furthermore, unlike the main processor, the nagging processors represent a shared resource and are used to support some number of additional users throughout the experiment.

Three serial configurations of DALI are tested. Σ_0 is a simple serial system that is essentially equivalent to a WAM-level reconstruction of PTTP modulo the previously cited difference in depth bound calculation.[2] Σ_1 adds intelligent backtracking, while Σ_2 incorporates intelligent backtracking, cycle detection, and a 100-element least-recently used subgoal cache. Each configuration performs unit-increment depth-first iterative deepening and is limited to exploring $1,000,000$ nodes before abandoning the problem and marking it as unsolved; elapsed CPU time (sum of system time and user time) is recorded for each problem. Note that the size of the cache is quite arbitrarily selected; larger or smaller caches may well result in improved performance. In addition, a unit increment may well be a substantially suboptimal increment for iterative deepening. Depending on the domain, increasing the increment value or changing the cache size may have a significant effect on the system's performance. The results reported in this paper are clearly dependent on the values of these parameters, but the conclusions we draw from these results are based only on comparisons between identically-configured systems.

The fourth tested configuration, Σ_3, adds 99 nagging processors to the configuration of Σ_2. Nagging processors are identically configured with intelligent backtracking, cycle detection, and 100-element least-recently used subgoal caches. For each problem, the currently "fastest" 99 machines (as determined by elapsed time for solving a short benchmark problem set) in the processor pool are selected for use as nagging processors. These additional processors are organized hierarchically, with 9 processors nagging the main processor and 10 more processors nagging each of these in turn. Hierarchical nagging, or *meta-nagging*, reduces the load on the main processor by amortizing nagging overhead costs over many processors. Recursively nagged processors are more effective as naggers in their own right, since nagging these processors prunes their search and helps them to exhaust their own search spaces more quickly. The main processor is subject to the same $1,000,000$ node resource constraint as the serial configurations tested.

Σ_0 solves 384 problems within the allotted resource bound, or 26.35% of the 1457 problems attempted. Σ_1,

[2] As a rough measure of performance, this configuration running on the hardware just described performs at about 10K LIPS on a benchmark definite-clause theory.

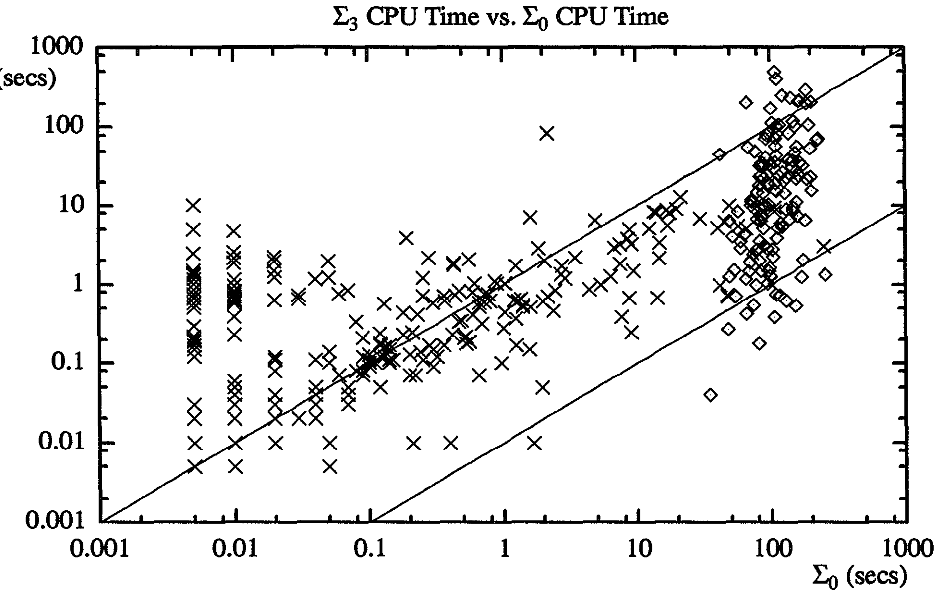

Figure 1: Performance of Σ_3 (log elapsed CPU time to solution on main processor) vs. performance of Σ_0 (log elapsed CPU time to solution or failure). The "cross" datapoints correspond to the 384 problems solved by both systems, while the "diamond" datapoints correspond to the 130 problems solved only by Σ_3; x-coordinate values for the "diamond" datapoints represent recorded time-to-failure for Σ_0, an optimistic estimate of actual solution time. The two lines represent $f(x) = x$ and $f(x) = x/100$. Since the granularity of our metering software is only 0.01 seconds, any problem taking less than 0.01 CPU seconds to solution is charged instead for 0.005 seconds.

identical to Σ_0 save for the use of intelligent backtracking and cycle detection, solves an additional 56 problems, or a total of 440 problems (30.19%). Σ_2, which adds a 100-element subgoal cache to the configuration of Σ_1, solves an additional 20 problems (76 more than Σ_0), for a total of 460 problems solved (31.57%). Finally, Σ_3, the 100-processor version of Σ_2, solves a total of 514 problems (35.27%). Note that in every case adding a search reduction technique results in the solution of additional problems.[3]

The additional problems solved by each successively more sophisticated configuration represent one important measure of improved performance. A second metric is the relative speed with which the different configurations solve a given problem; we consider here one such comparison between the most sophisticated system tested, Σ_3, and the least sophisticated system tested, Σ_0. Figure 1 plots the logarithm of the CPU time to solution for Σ_3 against the logarithm of the CPU required to either solve or fail to solve the same problem for Σ_0. Each point in the plot corresponds to a problem solved by at least one of the systems; the 384 "cross" datapoints correspond to problems solved by both systems, while the 130 "diamond" datapoints correspond to problems solved only by Σ_3. Datapoints falling below the line $f(x) = x$ represent problems that are solved faster by Σ_3, while datapoints falling below the line $f(x) = x/100$ represent problems that are solved more than 100 times faster with 100 processors.[4]

[3] While Σ_3 solves 56 problems not solved by Σ_2, 2 problems solved by Σ_2 were not solved by Σ_3. While no individual technique is likely to cause an increase in the number of nodes explored, interactions between techniques may result in such an increase. For example, changes to search behavior due to nagging will affect the contents of the main processor's cache; changes in cache contents will in turn affect the main processor's search behavior with respect to an identical serial system.

[4] The fact that some problems demonstrate superlinear speedup may seem somewhat alarming. Intuitively, using N identical processors should result in, at best, N times the performance. Here, the additional $N - 1$ processors are, on average, much slower than the main processor, and one would therefore initially expect sublinear speedup. However, superlinear speedup can result since the parallel system does not explore the space in the same order as the serial system. In particular, a nagging processor may explore a subgoal ordering that provides a solution with significantly less search than the original ordering, resulting in a net performance improvement much larger than N. In addition, the parallel system has the added advantage of subgoal caching and intelligent backtracking which also make substantial performance contributions on some problems.

If we consider only those problems solved by both systems (the 384 "cross" datapoints in Figure 1) and if we informally define "easy" problems to be those problems requiring at most 1 second to solve with the serial system, then we see that the performance of Σ_3 on such problems is often worse than that of the more naive serial system Σ_0; thus, many of these datapoints lie above the $f(x) = x$ line. We attribute this poor performance to the initial costs of nagging (*e.g.*, establishing communication and transmitting the domain theory to all processors). For "hard" problems, however, the initial overhead is easily outweighed by the performance advantage of nagging. Furthermore, the performance improvement on just a few "hard" problems dwarfs the loss in performance on all of the "easy" problems — an effect that is visually obscured by the logarithmic scale used for both axes of Figure 1.

A more precise way of convincing ourselves that Σ_3 is superior to Σ_0 is to use a simple nonparametric test such as the one-tailed paired sign test (Arbuthnott, 1710), or the one-tailed Wilcoxon matched-pairs signed-ranks test (Wilcoxon, 1945) to test for statistically significant differences between the elapsed CPU times for problems solved by both systems. These tests are nonparametric analogues to the more commonly used Student t-test; nonparametric tests are more appropriate here since we do not know anything about the underlying distribution of the elapsed CPU times.

The null hypothesis we are testing is that the recorded elapsed CPU times for Σ_0 are at least as fast as the recorded elapsed CPU times for Σ_3 on the 384 problems solved by both systems. The Wilcoxen test provides only marginal evidence for the conclusion that Σ_3 is faster than Σ_0 ($N = 384, p = .096$). However, if we only consider the "harder" problems, then there is significant evidence to conclude that Σ_3 is faster than Σ_0 ($N = 70, p < 10^{-6}$).

Of course, both our informal visual analysis of Figure 1 and the nonparametric analysis just given systematically understate the relative performance of Σ_3 by excluding the 130 problems that were solved only by Σ_3 (the "diamond" datapoints in Figure 1). Since these problems were not solved by Σ_0, we take the time required for Σ_0 to reach the resource bound as the abscissa for the datapoint in Figure 1: this is an *optimistic* estimate of the real solution time, since we know Σ_0 will require *at least* this much time to actually solve the problem. Graphically, the effect is to displace each "diamond" datapoint to the left of its true position by some unknown margin. Note that even though their x-coordinate value is understated, 117 (90%) of these datapoints still fall below the $f(x) = x$ line, and a 12 (roughly 10%) still demonstrate superlinear speedup. If we could use the actual Σ_0 solution time as the abscissa for these 130 problems, the effect would be to shift each "diamond" datapoint to the right to its true position, greatly enhancing Σ_3's apparent performance advantage over Σ_0.

Is it possible to tease apart the performance contribution due to each individual technique? We can use the Wilcoxon test to compare each pair of successively more sophisticated system configurations. We conclude that Σ_1 is significantly faster than Σ_0 ($N = 384, p < 10^{-6}$), indicating that intelligent backtracking and cycle detection together are effective serial speedup techniques. In a similar fashion, Σ_2 is in turn significantly faster than Σ_1 ($N = 440, p < 10^{-6}$), indicating that subgoal caching is also an effective serial speedup technique when used with intelligent backtracking and cycle detection. In contrast, we find only marginal evidence that Σ_3 is uniformly faster than Σ_2 over the entire problem collection ($N = 458, p = .072$). However, as with the Σ_3 vs. Σ_0 comparison, separating "harder" problems, where Σ_3 significantly outperforms Σ_2 ($N = 108, p < 10^{-6}$), from "easier" problems, where the Σ_2 significantly outperforms Σ_3 ($N = 350, p < 10^{-6}$) enables us to make statistically valid statements about the relative performance of Σ_3 and Σ_2. As before, all of these results — by ignoring problems left unsolved by one of the systems being compared — systematically understate the performance advantage of the more sophisticated system in the comparison.

Unlike nagging, where problem size is a good predictor of performance improvement, it is much more difficult to characterize when caching, intelligent backtracking, or cycle detection are advantageous. Some problems are solved more quickly with these techniques, while others problems are not; knowing whether a problem is "hard" or "easy" *a priori* gives no information about whether or not caching, intelligent backtracking, or cycle detection will help, a conclusion that is supported by our statistical analysis.

Conclusion

We have briefly reviewed the design and implementation of the DALI system. The premise of this paper is that efficient implementation technology, serial search reduction techniques, and nagging can be successfully combined to produce a first-order logic engine that can effectively bring hundreds of workstations to bear on large problems. We have supported our claims empirically over a broad range of problems from the theorem proving literature. While the results presented here are quite good, we believe we can still do better. We are now in the process of adding an *explanation-based learning* component that compiles "chains of reasoning" used in successfully solved problems into new macro-operators (Segre & Elkan, 1994). We are also exploring alternative cache-management policies, the use of dynamically-sized caches, and compile-time techniques for determining how caching can be used most effectively in a given domain. Similarly, we are studying compile-time techniques for selecting appropriate opportunities for nagging and we are also looking at how best to select a topology of recursive nagging processors. Finally, we are exploring additional sources of parallelism.

These efforts contribute to a larger study of practical, effective, inference techniques. In the long term, we believe that our distributed, adaptive, approach to first-order logical inference — driven by a broad-spectrum philosophy that integrates multiple serial search reduction techniques as well as the use of multiple processing elements — is a promising one that is also ideally suited to large-scale applications of significant practical importance.

Acknowledgements

We wish to acknowledge Maria Paola Bonacina, Randy Calistri-Yeh, Charles Elkan, Don Geddis, Geoff Gordon, Simon Kasif, Drew McDermott, David Plaisted, Mark Stickel, and three anonymous reviewers for their helpful comments on an early draft of this paper. Support for this research was provided by the Office of Naval Research through grant N00014-90-J-1542 (AMS), by the Advanced Research Project Agency through Rome Laboratory Contract Number F30602-93-C-0018 via Odyssey Research Associates, Incorporated (AMS), and by the Air Force Office for Scientific Research through a Graduate Student Fellowship (DBS).

References

Aït-Kaci, H. (1991). *Warren's Abstract Machine*. Cambridge, MA: MIT Press.

Arbuthnott, J. (1710). An Argument for Divine Providence, Taken from the Constant Regularity Observed in the Births of Both Sexes. *Philosophical Transactions, 27*, 186-190.

Calistri-Yeh, R.J. & Segre, A.M. (April 1993). *The Design of ALPS: An Adaptive Architecture for Transportation Planning* (Technical Report TM-93-0010). Ithaca, NY: Odyssey Research Associates.

Korf, R. (1985). Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence, 27*(1), 97-109.

Kumar, V. & Lin, Y-J. (August 1987). An Intelligent Backtracking Scheme for Prolog. *Proceedings of the IEEE Symposium on Logic Programming*, 406-414.

Plaisted, D. (1988). Non-Horn Clause Logic Programming Without Contrapositives. *Journal of Automated Reasoning, 4*(3), 287-325.

Segre, A.M. & Elkan, C.P. (*To appear*, 1994). A High Performance Explanation-Based Learning Algorithm. *Artificial Intelligence*

Segre, A.M. & Scharstein, D. (August 1993). Bounded-Overhead Caching for Definite-Clause Theorem Proving. *Journal of Automated Reasoning, 11*(1), 83-113.

Stickel, M. (1988). A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning, 4*(4), 353-380.

Sturgill, D.B. & Segre, A.M. (To appear, June 1994). A Novel Asynchronous Parallelization Scheme for First-Order Logic. *Proceedings of the Twelfth Conference on Automated Deduction*

Suttner, C.B., Sutcliffe, G. & Yemenis, T. (1993). *The TPTP Problem Library (TPTP v1.0.0)* (Technical Report FKI-184-93). Munich, Germany: Institut für Informatik, Tecnische Universität München.

Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. *Biometrics, 1*, 80-83.