

Adaptive Parallel Search for Theorem Proving

Diane J. Cook and Charles Hannon

Box 19015

University of Texas at Arlington

Arlington, TX 76019

Email: {cook,hannon}@cse.uta.edu

Abstract

Many of the artificial intelligence techniques developed to date rely on heuristic search through large spaces. Unfortunately, the size of these spaces and the corresponding computational effort reduce the applicability of otherwise novel and effective algorithms. Because automated theorem provers rely on heuristic search through the space of possible inferences, they are subject to the same difficulties.

A number of parallel and distributed approaches to search have considerably improved the performance of the search process. Our goal is to develop an architecture that automatically selects parallel search strategies for a variety of search problems. We describe one such architecture realized in the EUREKA system, which uses machine learning techniques to select the parallel search strategy for a given problem space. Although EUREKA has successfully improved parallel search for problem solving and planning, parallelizing theorem proving systems introduces several new challenges. We investigate the application of the EUREKA system to a parallel version of the OTTER theorem prover and show results from a subset of TPTP library problems.

Introduction

Because of the dependence AI techniques demonstrate upon heuristic search algorithms, researchers continually seek more efficient methods of searching through the large spaces created by these algorithms. Advances in parallel and distributed computing offer potentially large increases in performance to such compute-intensive tasks. In response, a number of parallel approaches have been developed to improve various search algorithms including depth-first search (Kumar & Rao 1990), branch-and-bound search (Agrawal, Janakiram, & Mehrotra 1988), A* (Mahapatra & Dutt 1995), IDA* (Powley & Korf 1991), and game tree search (Feldmann, Mysliwicz, & Monien 1994), as well as to improve the run time of specific applications such as the fifteen puzzle problem and robot arm path planning. While existing approaches to parallel search have many contributions to offer, comparing these approaches and deter-

mining the best use of each contribution is difficult because of the diverse search algorithms, implementation platforms, and applications reported in the literature.

In response to this problem, we have developed the EUREKA parallel search engine that combines many of these approaches to parallel heuristic search. EUREKA (Cook & Varnell 1998) is a parallel search architecture that merges multiple approaches to task distribution, load balancing, and tree ordering. The system can be run on a MIMD shared memory or distributed memory parallel processor, a distributed network of workstations, or an SMP machine with multithreading. EUREKA uses a machine learning system to predict the set of parallel search strategies that will perform well on a given problem, which are then used to complete the search task. In this paper we will describe the EUREKA system and investigate methods of applying adaptive parallel search methods to the parallelization of an automated theorem prover, OTTER.

Parallel Search Approaches

A number of researchers have explored methods for improving the efficiency of search using parallel hardware. We have focused on the integration and comparison of techniques for task distribution, for dynamically balancing work between processors, and for changing the left-to-right order of the search tree. Additional decisions involved in parallel search strategies can also be investigated and included in the EUREKA architecture with minimal overhead.

Distributed search algorithms require a balanced division of work between contributing processors (or agents) to reduce idle time and minimize wasted effort. Powley and Korf (Powley & Korf 1991) introduce one method of dividing work by assigning unique depth thresholds to individual processors. This method of task distribution achieves the best results when multiple iterations of a depth-first search (as found in IDS and IDA*) are needed to find a goal node. An alternative approach distributes individual subtrees to the processors (Kumar & Rao 1990). This approach is effective when the branching factor of the search space is large and when the tree is evenly balanced. Previous work has demonstrated that neither approach is consis-

tently effective and a compromise between the methods can sometimes produce the best results (Cook & Varnell 1997).

When a problem is broken into disjoint subtasks the workload often varies between processors. When one processor runs out of work before the others, load balancing can be used to activate the idle processor. Load balancing requires many decisions to be made – in particular, when to initiate balancing, which processors to initiate load balancing and to approach for work, and how much work to give. Nearest neighbor approaches (Mahapatra & Dutt 1995) are compared with random selection approaches and greedy approaches to selecting a processor from which to request work. In a search algorithm, work may be given in the form of nodes from the open list near the front (deep in the tree), the end (high in the tree), or from a sampling of all levels (Kumar & Rao 1990). Dutt and Mahapatra (Mahapatra & Dutt 1995) suggest an anticipatory approach that triggers load balancing when the load is not quite empty, so that a processor may continue working on remaining nodes while waiting for a response to the load balancing request.

Problem solutions can exist anywhere in the search space. However, many search approaches such as depth-first search, IDS, and IDA* yield a run time that is highly dependent on the left-to-right position of the goal in the space. Assuming the space is expanded left to right, a far greater number of nodes must be expanded to find a solution lying on the far right side of the tree than a solution lying on the left side of the tree. A variety of methods have been explored to use information found during the search itself to reorder the left-to-right positioning of nodes in the tree, with varying success. These methods include local ordering of children as they are generated (Powley & Korf 1991), sorting of an entire frontier set of nodes (Powley & Korf 1991), or ordering operators (Cook & Varnell 1998).

The EUREKA System

Based on the characteristics of a search space, EUREKA automatically configures itself to optimize performance on a particular application. Sample problems are fed as training examples to a machine learning system, which in turn learns the strategies to apply to particular classes of search spaces. In particular, sample problems are described using pertinent features such as branching factor, heuristic error, tree imbalance, heuristic branching factor, and estimated distance from the root to the nearest goal. Problems are collected from a variety of domains and are run with multiple strategy settings. The strategy choice(s) that yields the best speedup is labeled as the correct strategy “classification” for that problem. In our initial implementation, we use C4.5 (Quinlan 1993) to induce a decision tree from these classified training cases. More recently, we have also constructed a belief network using Netica to make decisions with highest predicted utility (Cook & Gmytrasiewicz 1999). To solve a new problem, EUREKA

Approach	Speedup
Eureka	74.24
Random Processor LB	70.75
Local Ordering	68.92
Transformation Ordering	66.13
Kumar and Rao	65.89
Distributed Tree	65.82
Fixed Evaluation 1	65.41
1 Cluster	65.21
Neighbor LB	65.21
30% Distribution	65.21
2 Clusters	64.97
50% Distribution	61.94
Fixed Evaluation 2	49.58
4 Clusters	49.57
Avg. of Fixed Strategies	63.43

Table 1: Performance comparison of Eureka with fixed strategies

searches enough of the tree to measure the features of the problem space. The tree features are calculated and used to make appropriate strategy decisions for the remainder of the parallel search.

Results generated from four problem domains and three parallel/distributed architectures indicate that EUREKA can automatically select strategies and parameters based on problem feature descriptions. In fact, EUREKA outperforms any strategy choice used exclusively on all problem instances. For example, Table 1 shows the speedup achieved when EUREKA makes all strategy decisions (task distribution, load balancing, ordering) at once and employs the chosen strategies, compared with all strategy choices used by itself for all problems, averaged over 50 instances of the fifteen puzzle problem. This experiment was run on 32 processors of an nCUBE II message-passing multiprocessor. As the results indicate, superlinear speedup (speedup greater than the number of processors) sometimes occurs because the parallel algorithms do not completely imitate the serial algorithms.

As the table demonstrates, an adaptive approach to parallel search can yield performance improvement over serial search and outperforms any fixed approach to parallel search in these domains.

Parallel Search for Theorem Provers

Automated theorem provers have been useful at proof generation, proof checking, and query answering. Theorem provers have come up with novel mathematical results and are continually being infused with greater deductive power and greater flexibility. However, theorem provers employ search methods to drive the inference process. As a result, these systems may suffer from prohibitive computational expense and do not always scale well with increasing problem size.

Our goal is to employ parallel search techniques to

improve the performance of automated theorem proving. Other parallel theorem proving systems have been introduced (Wolf & Letz 1998; Lusk, McCune, & Slaney 1992). Our approach differs in its ability to adapt the parallelization to the specific problem and machine architecture. We select as our theorem prover the OTTER system (McCune 1992). OTTER performs resolution-style theorem proving on first-order statements. Unlike the previous applications we have investigated which employ IDA* search, OTTER uses best-first search in which "lighter" clauses are selected from an "sos" list to resolve with clauses residing in a "usable" list. The default method of weighting each clause is by summing the weights of its literals, and individual weights can be provided by the user.

To produce an efficient application of parallel search to automated theorem proving, we tightly couple the EUREKA and OTTER systems. The existing top-level search loop inside OTTER is removed and all search functions are performed by EUREKA. The new parallel system calls all of the original OTTER deduction functions, but can employ adaptive parallel search techniques to improve the overall run time by parallelizing the search process.

All of the previous applications of EUREKA employed a form of iterative deepening search. The non-iterative approach to search employed by OTTER does not make use of many of the parallel search techniques existing in EUREKA. As a result, we refine EUREKA to alter the parallel search approach. During each search iteration of the serial theorem prover, one clause is selected to add to the usable list and resolve with other clauses there. Thus the system iterates through each clause in the sos list, adding each in turn to the usable list and generating all new clauses that can be inferred from that parent. EUREKA parallelizes this search effort by dividing the sos search space between multiple processors. In particular, EUREKA allows the following search strategy choices to be made.

- **Selection of sos clause [partition, random].**

To parallelize the sos search, EUREKA will allow each processor to potentially select a different clause from the sos list to add to the usable list and thus search different portions of the space. As one option, EUREKA will give each processor a separate window indicating a portion of clauses from the sos list that the processor can choose. The processor will choose the lightest clause from that partition only, in effect dividing the sequential search through the sos list among multiple processors. As a second option, EUREKA will allow each processor to make a separate random choice among all clauses in the sos list.

- **Load balancing [load balance, no load balance].**

Because the search effort is divided among multiple processors, some processors may move quickly toward the goal clause while others veer off of the optimal path. To prevent idle processors and ensure that pro-

cessors are working on a potentially useful portion of the search space, EUREKA can perform load balancing at regular intervals during the search process. If load balancing is selected, all processors will periodically share their best (lightest) clauses with all other processors. These shared clauses will be integrated into each processor's sos list and can more quickly direct a floundering processor toward a solution. This option is useful when a large weight imbalance exists between portions of the search space, but can actually degrade performance on occasion because of the communication overhead.

EUREKA uses machine learning techniques to select a parallel search strategy for each problem. To perform adaptive parallel search, EUREKA executes the following steps:

1. Generate timings from sample problem instances by running the EUREKA/OTTER multiple times on each problem, using each possible strategy. The strategy in each case that yields the best performance is considered to be the correct "classification" for the problem.
2. For each problem, capture features of the problem space that affect the optimal choice of strategy. Using the OTTER theorem prover, these features include:
 - **Highest input clause weight (continuous).** The highest weight of all input clauses.
 - **Lowest input clause weight (continuous).** The lowest weight of all input clauses.
 - **Average input clause weight (continuous).** The weight of input clauses, averaged over all clauses in the input list.
 - **Horn (0 or 1).** Whether any of the input clauses are Horn clauses.
 - **Equality (0 or 1).** Whether equality statements are present in the input clause list.
 - **MaxLits (continuous).** The maximum number of literals in any of the input clauses.

These features are used to describe each problem and can be calculated before any search is performed. Many of these features directly affect the size and nature of the search space, and the values of these features often change dramatically from one problem to the next. As a result, the features provide good indicators of the resulting search space.
3. Generate a database of problem descriptions and corresponding classifications and supply the database as input to a machine learning systems. We use C4.5 (Quinlan 1993) to induce a decision tree from the pre-classified cases. A rule base is generated for each concept to be learned, corresponding to each of the strategy decisions that need to be made.
4. To solve a new problem, generate the problem features, select the corresponding parallel search strategies and execute the parallel theorem prover with the selected strategies on the problem.

Approach	Speedup
Partition	9.68
Random	8.37
Eureka	12.66

Table 2: Results of varying strategies for selecting sos clause.

Approach	Speedup
Load balance	16.69
No load balance	9.68
Eureka	16.78

Table 3: Results of varying strategies for load balancing.

Experimental Results

Our test domain consists of the 25 largest run-time problems of the TPTP problem library (Suttner & Sutcliffe 1998) that were used for the CADE-14 theorem prover competition. Each problem was run serially and using each described parallel search strategy. All of the parallel executions were run on four Pentium PCs using the PVM message-passing libraries for communication.

In these experiments, we compare the results of EUREKA-selected strategies to each strategy used exclusively for all problems. Speedup results calculated as serial run time / parallel run time are averaged over all problems. The C4.5 results are captured by performing a ten-fold cross validation on the data set. To account for differences in problem size, we weight problems according to the variance in speedup between different strategies. C4.5 speedup is averaged over all test cases for one iteration of this process, and the final values are averaged over all cross-validation iterations.

As Table 2 shows, EUREKA produces greater speedup than either of the strategy choices used exclusively for all problems. In this experiment, the partition approach performed best in 23 out of 25 cases, the random approach performed best in 1 out of 25 cases, and in 1 case the two approaches yielded the same run time. C4.5 correctly classifies 24 out of 25 cases. One of the cases in which the random strategy outperforms the partition strategy has a large speedup of 149. On 19 of the problems, at least one of the parallel strategies yields a run time that is worse than that of the serial algorithm.

Table 3 lists the speedup results when load balancing is used and not used. Using load balancing yields greater speedup over no load balancing in 12 out of the 25 cases, no load balancing performs best in 5 out of the 25 cases, and the two strategies yield the same run time in 8 out of the 25 test cases. On this database, C4.5 correctly classifies 23 out of the 25 cases. At least one of the parallel strategies yields a run time that is worse than that of the serial algorithm for 14 cases, again demonstrating that careful selection of the paral-

lel search strategies is important.

Conclusions and Future Work

This paper reports on work performed to combine the benefits of parallel search approaches in the EUREKA system, and to apply these parallel search strategies to the OTTER theorem proving system. Experimentation reveals that strategies developed over the last few years offer distinct benefits to improving the performance of AI applications. However, while any particular algorithm can provide significant speedup for one type of problem, on other problems these algorithms can actually produce worse results than using a serial version of the search algorithm. As a result, these strategies need to be carefully chosen based on the characteristics of a particular problem.

In this paper we integrate into EUREKA parallel search strategies for the OTTER theorem prover. On 25 sample problems, EUREKA is able to substantially improve the performance of the theorem prover using parallel search. In addition, through EUREKA's use of the C4.5 machine learning system, we further improve the performance of the parallel search approach to theorem proving by providing a means of adapting the parallel strategy choices to each problem.

Two of the challenges introduced by our research are the ability to determine discriminating features and the ability to provide clear classifications for training examples. In our experiments we verified that the features we chose could be measured before search begins and still provide good indicators of the parallel search strategies that perform well during execution. As we apply our approach to a greater variety of problems we will need to develop a more formal methodology for selecting representative and discriminating features. In addition, we observed performance improvement when test cases were weighted according to the variance in speedup between strategies. We would like to develop a machine learning approach that will make use of probabilistic classes to learn from exact run times of the strategies, and not just from one classification for each problem.

EUREKA implementations are currently available on a variety of architectural platforms including MIMD distributed memory and shared memory multiprocessors, a distributed network of machines running PVM, Posix multithreading machines, and machines using Java threads and Cilk threads. Problem domains currently under investigation include additional combinatorial optimization problems such as the n-Queens problem and integration of machine learning, and natural language algorithms into this search architecture. We hope to demonstrate that parallel heuristic search algorithms can yield near-optimal *and* scalable approaches to planning, machine learning, natural language, theorem proving, and many other computation-intensive areas of AI.

Acknowledgements

This work was supported by National Science Foundation grants IRI-9308308, IRI-9502260, and DMI-9413923.

References

- Agrawal, D.; Janakiram, V.; and Mehrotra, R. 1988. A randomized parallel branch and bound algorithm. In *Proceedings of the International Conference on Parallel Processing*, 69–75. The Pennsylvania State University.
- Cook, D. J., and Gmytrasiewicz, P. 1999. Controlling the parameters of parallel search using uncertainty reasoning. In *Proceedings of the AAAI Symposium on Search Strategy under Uncertain and Incomplete Information*.
- Cook, D. J., and Varnell, R. C. 1997. Maximizing the benefits of parallel search using machine learning. In *Proceedings of the National Conference on Artificial Intelligence*, 559–564. AAAI Press.
- Cook, D. J., and Varnell, R. C. 1998. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research* 9:139–166.
- Feldmann, R.; Mysliwicz, P.; and Monien, B. 1994. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, 94–103. Association for Computing Machinery.
- Kumar, V., and Rao, V. N. 1990. Scalable parallel formulations of depth-first search. In Kumar; Kanal; and Gopalakrishnan., eds., *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag. 1–41.
- Lusk, E. L.; McCune, W.; and Slaney, J. K. 1992. Roo: A parallel theorem prover. In *CADE 1992*, 731–734.
- Mahapatra, N. R., and Dutt, S. 1995. New anticipatory load balancing strategies for parallel A* algorithms. In *Proceedings of the DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 197–232. American Mathematical Society.
- McCune, W. W. 1992. Automated discovery of new axiomatizations of the left group and right group calculi. *Journal of Automated Reasoning* 9(1):1–24.
- Powley, C., and Korf, R. E. 1991. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13(5):466–477.
- Quinlan, J. R. 1993. *C4.5: programs for machine learning*. Morgan Kaufmann.
- Suttner, C. B., and Sutcliffe, G. 1998. The cade-14 atp system competition. *Journal of Automated Reasoning* 21(1):99–134.
- Wolf, A., and Letz, R. 1998. Strategy parallelism in automated theorem proving. In *Proceedings of the Florida Artificial Intelligence Research Symposium*, 142–146.