Maximizing the Benefits of Parallel Search Using Machine Learning

Diane J. Cook and R. Craig Varnell

University of Texas at Arlington
Box 19015, Arlington, TX 76019
{cook,varnell}@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 corresponding computational effort reduce the applicability of otherwise novel and effective algorithms. A number of parallel and distributed approaches to search have considerably improved the performance of certain aspects of the search process.

In this paper we describe the EUREKA system, which combines the benefits of many different approaches to parallel heuristic search. EUREKA uses a machine learning system to decide upon the optimal parallel search strategy for a given problem space. When a new search task is input to the system, EUREKA gathers information about the search space and automatically selects the appropriate search strategy. EUREKA includes diverse approaches to task distribution, load balancing, and tree ordering, and has been tested on a MIMD parallel processor, a distributed network of workstations, and a single workstation using multithreading. Results in the fifteen puzzle domain, robot arm path planning domain, and an artificial domain indicate that EUREKA outperforms any existing strategy used exclusively for all problem instances.

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 approaches to parallel AI have been developed that make use of MIMD and SIMD hardware to improve various aspects of search algorithms including depth-first search (Kumar & Rao 1990), branch-and-bound search (Agrawal, Janakiram, & Mehrotra 1988), A* (Mahapatra & Dutt 1995; Evett *et al.* 1995), and IDA* (Mahanti & Daniels 1993; Powley & Korf 1991; Powley, Ferguson, & Korf 1993). While existing approaches to parallel search have many contributions to offer, comparing these approaches and determining the best use of each contribution is difficult because of the diverse search algorithms, machines, 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 is a parallel IDA* search architecture that merges multiple approaches to task distribution, load balancing, and tree ordering, and can be run on a MIMD parallel processor, a distributed network of workstations, or a single machine with multithreading. EUREKA uses the C4.5 machine learning system to predict the optimal set of parallel search strategies for a given problem, and will automatically use the selected strategies to complete a search task.

Parallel Search Approaches

A number of researchers have explored methods for improving the efficiency of search using parallel hardware. We will review existing methods for task distribution, for balancing work between processors, and for changing the left-to-right order of the search tree.

Task Distribution

Effectively dividing up a search problem among contributing processors reduces processor idle time and thus improves performance. One method of dividing up the work in IDA* search is using parallel window search (PWS), introduced by Powley and Korf (Powley & Korf 1991). Using PWS, each processor is given a copy of the entire search tree and a unique cost threshold. The processors search the same tree to different thresholds simultaneously. If a processor completes an iteration without finding a solution, it is given a new unique threshold (deeper than any threshold yet searched) and begins a new search pass with the new

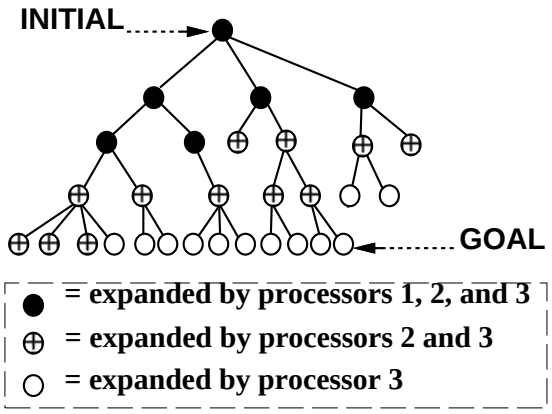

Figure 1: Division of work in parallel window search

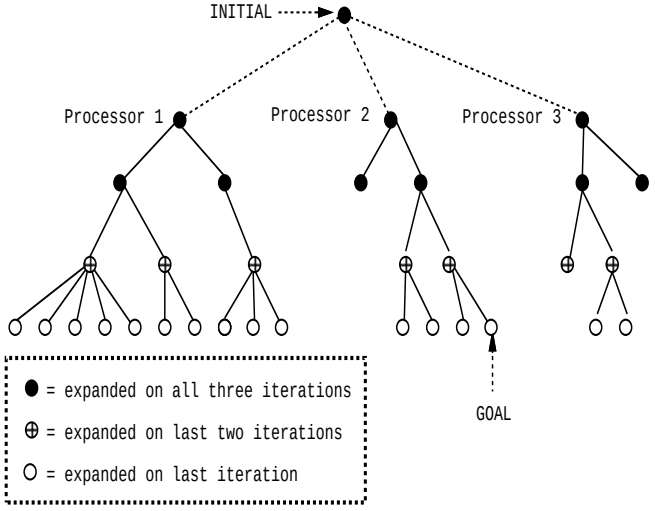

Figure 2: Division of work in distributed tree search

threshold. When an optimal solution is desired, processors that find a goal node must remain idle until all processors with lower cost thresholds have completed their current iteration. A typical division of work using PWS is illustrated in Figure 1.

Another approach to parallel search relies on distributing the tree among the processors (Kumar & Rao 1990; Rao, Kumar, & Ramesh 1987). With this approach, the root node of the search space is given to the first processor and other processors are assigned subtrees of that root node as they request work. As an alternative, the distributed tree search algorithm (DTS) employs breadth-first expansion until there are at least as many expanded leaf nodes as available processors. Processors receive unique nodes from the expanding process and are responsible for the entire subtree rooted at the received node. Communication-free versions of this distribution scheme are also reported (Mahapatra & Dutt 1995; Reinefeld & Schnecke 1994) In these approaches, the processors perform IDA* on their unique subtrees simultaneously. All processors search to the same threshold. After all processors have finished a single iteration, they begin a new search pass through the same set of subtrees using a larger threshold. A sample distribution of the search space is shown in Figure 2.

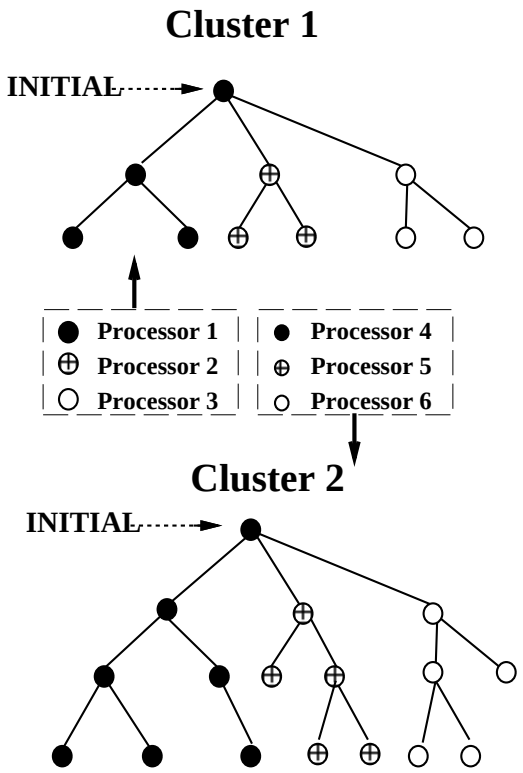

Figure 3: Space searched by two clusters, each with 3 processors

These described approaches offer unique benefits. Parallel window search is effective when many iterations of IDA* are required, when the tree is so imbalanced that DTS will required excessive load balancing, or when a deep, non-optimal solution is acceptable. On the other hand, dividing the search space among processors can be more effective when the branching factor is very large and the number of IDA* iterations is relatively small. A compromise between these approaches is to divide the set of processors into *clusters* (Cook & Nerur 1995; Cook 1997). Each cluster is given a unique cost threshold, and the search space is divided between processors within each cluster, as shown in Figure 3.

Load Balancing

When a problem is broken into disjoint subtasks the workload will likely vary among processors. Because one processor may run out of work before others, load balancing is used to activate the idle processor. The first phase of load balancing involves selecting a processor from which to request work. One example is the nearest neighbor approach (Mahapatra & Dutt 1995). Alternative approaches include selecting random processors or allowing a master processor to keep track of the load in the other processors and to send the ID of a heavily loaded processor to one that is idle. During the second phase of load balancing, the requested processor decides which work, if any, to give. A search algorithm typically stores nodes which have not been fully expanded on an Open list. When giving work to another processor, nodes can be given from the top of the list (deep in the tree), the bottom of the list (near the root), or from a sampling of all levels (Kumar & Rao 1990). Dutt and Mahapatra (Mahapatra & Dutt 1995) suggest an anticipatory approach in which nodes send out a work request when their load is near, but not at, empty, so that they can continue processing remaining nodes while waiting for a response.

Tree Ordering

Problem solutions can exist anywhere in the search space. Using IDA* search, the children are expanded in a depth-first manner from left to right, bounded in depth by the cost threshold. If the solution lies on the right side of the tree, a far greater number of nodes must be expanded than if the solution lies on the left side of the tree. If information can be found to re-order the operators in the tree from one search iteration to the next, the performance of IDA* can be greatly improved.

Powley and Korf suggest two methods of ordering the search space (Powley & Korf 1991). First, children of each node can be ordered and expanded by increasing heuristic distance to a goal node. Alternatively, the search algorithm can expand the tree a few levels and sort the *frontier set* (the set of nodes at that level in the tree) by increasing h value. Search begins each iteration from the frontier set and this frontier set is updated each iteration. In both of these cases, although the nodes may have the same f value, nodes with smaller h values generally reflect a more accurate estimated distance and are preferred.

Instead of ordering individual nodes, Cook and Hall (Cook, Hall, & Thomas 1993) order the set of operators to guide the next IDA* iteration to the *most promising node*. The most promising node is the node from the cut-off set (a child node not expanded in the previous iteration) with the smallest h value. As an example, Figure 4 shows a search tree expanded using one iteration of IDA*. The path to the most promising leaf node (indicated with a star) is 1 3 3 2 0. The new operator ordering is computed using the order of operators as they appear in this path after removing duplicates. Operators not appearing in the path are added to the end of operator list, retaining their original relative ordering. Thus the ordering of operators for the example in Figure 4 changes from 0, 1, 2, 3 (always try operator 0 first, operator 1 next, operator 2 next, and operator 3 last) to 1, 3, 2, 0.

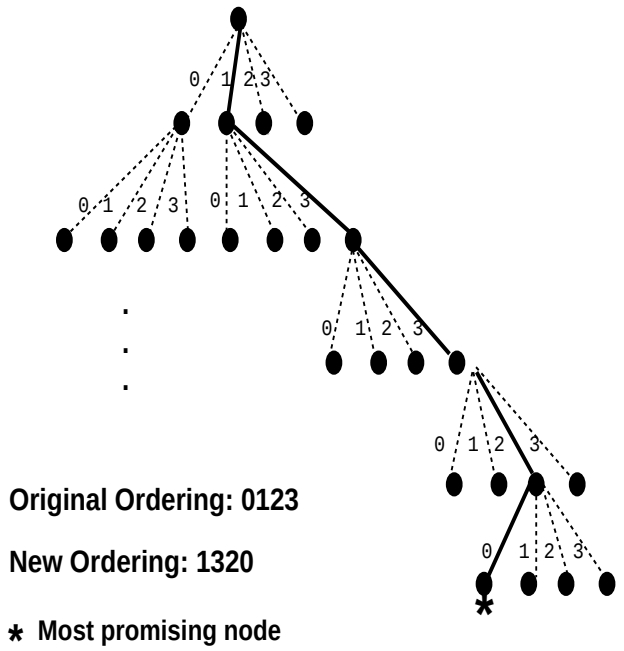

Figure 4: Operator ordering example

The EUREKA System

The EUREKA system merges together many of the strategies discussed in the previous section. Parameters can be set that control the task distribution strategy, the load balancing strategies, and the ordering techniques. In particular, the strategies that can be selected include:

- Distribution strategy [Kumar and Rao, distributed tree search]
- Number of clusters [1..#processors]
- Fraction of Open list to donate upon neighbor request [0..1]

- Donating strategy [TopOfList, BottomOfList]
- Anticipatory load balancing [On, Off]
- Load balancing [On, Off]
- Ordering [None, Local, Operator]

To maximize the potential benefit of parallel search without placing the responsibility of selection on the user, we use the C4.5 machine learning system (Quinlan 1993) to assume this task. C4.5 induces a decision tree from the pre-classified test cases that can be used to classify new problem instances. A rule base is generated for each concept to be learned, corresponding to each of the strategy decisions listed above that need to be made. Test cases are fed to the system from a variety of problem domains and parallel architectures. Each test case is described using features of the corresponding problem that are known to affect the optimal choice of strategy. These features include:

Branching Factor (b): The average branching factor of the search tree.

Heuristic Error ($herror$): The difference, on average, between the estimated distance to a goal node and the true distance to the closest goal node.

Imbalance (imb): The degree to which nodes are unevenly distributed among subtrees in the search space.

Goal Location (loc): The left-to-right position of the first discovered optimal solution node.

Heuristic Branching Factor (hbf): The ratio of nodes expanded between the current and previous IDA* iterations.

Once the learned rule base is in place, C4.5 can make strategy selections for a new problem instance. Experiments reveal that all of the sampled features are fairly stable in many search trees. Features such as branching factor, heuristic error, imbalance, and so forth, will have similar values from level to level in the tree. As a result, these values can be collected from an early level in the tree and represent good indicators of the rest of the search tree.

To solve a new problem, EUREKA searches through the space until roughly 100,000 nodes are expanded. The features of the tree are calculated at this point and used to index appropriate rules from the C4.5 database. Parallel search is then initiated from the root of the space, employing the selected strategies. In general, the initial expansion takes only a few seconds and does not greatly affect the runtime of the search algorithm.

Domain	b	herror	ibm	hbf
15 puzzle	0.01	1.32	0.02	1.68
RAP	0.32	0.06	0.04	0.0

Table 1: Normalized standard deviation of feature values over a range of levels in search trees.

The described set of features and the amount of time to spend on the initial EUREKA iteration are chosen to yield the most helpful information in the shortest time. Searching enough iterations of the problem space until 100,000 nodes are generated generally takes less than 10 seconds on the problem domains we tested. Spending less time than this may yield erroneous information because features of the tree do not stabilize until several levels down in the tree. Searching additional iterations in general does not significantly improve the quality of information and the time requirements grow exponentially.

The selected features each demonstrate a significant influence on the optimal search strategy. In addition, each feature remains fairly stable from one level of the tree to the next. As a result, computing the values of these features at a low level in the tree provides a good indication of the structure of the entire tree. Table 1 shows the standard deviation of each set of feature values sampled over a range of levels in the tree. The table lists the normalized standard deviations (the standard deviation is divided by the mean value) for each feature sampled from the first level that contains over 500 nodes to the goal level for 100 problem instances in the fifteen puzzle domain and 20 problem instances in the robot arm path planning domain.

Experimental Results

To train the learning system for EUREKA, we provided C4.5 with problem instances from the fifteen puzzle domain, the robot arm path planning domain, and an artificial search space. A complete description of these problem domains can be found in the literature (Cook 1997). A user can adjust the following characteristics of the artificial search space:

Maximum branching factor. No node in the tree will have more children than this specified value.

Load imbalance. A real number between 0 (perfectly balanced) and 1 (perfectly imbalanced) is used to indicate the amount of imbalance in the tree. An imbalanced tree contains the greatest number

of nodes in the middle of the tree, and the left and right sides contain sparse subtrees.

Accuracy of heuristic estimate. A real number between 0 and 1 is used to indicate the accuracy of the heuristic estimator. Because h should never overestimate the true distance to the goal, h is defined as $herror * true\_distance$.

Cost of optimal solution. Each move is assigned a cost of 1, so the optimal cost is the depth in the tree at which the optimal solution can be found.

Position of optimal solution. A sequence of moves is supplied to define the path to the first optimal goal found in a left-to-right search of the tree.

Solution density. A real number between 0 and 1 represents the probability that any given node found after the first optimal solution is also a goal node.

To create test cases, we ran each problem instance multiple times, once using each parallel search strategy in isolation. The search strategy that produces the best speedup is considered to be the "correct" classification of the corresponding search tree for C4.5 to learn. Test cases were run on 64 processors of an nCUBE, on 8 distributed workstations running the Parallel Virtual Machine (PVM) software, and on one DEC Alpha using an implementation of Posix threads.

We compared the results of C4.5-selected strategies to each strategy used exclusively for 100 fifteen puzzle problem instances. Speedup results for various strategy decisions averaged over all problem instances are shown in Table 2 below. The best average speedup is starred in each case.

These results were captured using 10-fold cross validation on the data sets. As indicated by these early results, C4.5 selection of search strategies yields better speedup in most cases than using a single strategy in isolation for all problem instances. Combining all test cases from the various problem domains and architectures together and allowing C4.5 to make all strategy decisions results in a speedup of 65.03 averaged over 100 fifteen puzzle instances run on the nCUBE, versus an average speedup of 54.06 when a fixed set of strategies is used for all problem instances averaged over all search strategies.

We compared the performance of C4.5 on these test cases to other machine learning systems: the ID3 decision tree induction algorithm modified to handle multiple classes, a k nearest neighbor algorithm, a perceptron, and a backpropagation neural network. For 100 cases of the search program, backpropagation performed best, followed by C4.5, a Bayesian classifier, KNN, ID3, and the perceptron.

Number of clusters on nCUBE			
1	2	4	EUREKA
52.02	57.04	56.83	58.90*
Task distribution strategy on nCUBE			
Kumar&Rao	DTS		EUREKA
52.02	53.03*		52.31
Load balancing strategy on nCUBE			
Neighbor	Random		EUREKA
52.02	55.35		55.55*
Ordering strategy on nCUBE			
None	Operator	Local	EUREKA
49.58	50.77	26.83	50.97*
Anticipation strategy on nCUBE			
None	4 node	10 node	EUREKA
52.02	51.48	51.30	55.75*
Number of clusters using PVM			
1	2		EUREKA
7.69	7.03		7.72*
Percentage giveaway using PVM			
30%	50%		EUREKA
7.69*	7.49		7.50

Table 2: Selection of the percentage of giveaway using a distributed network.

Even though a neural network may outperform C4.5 for the problem of selecting optimal parallel search strategies, we still choose to use C4.5 because the output produces understandable rules that provide insight in comparing existing parallel search strategies and suggesting areas for future work.

For example, one C4.5 rule states that when the branching factor is very low or when imbalance is very high, a larger number of clusters should be used. For a tree with a reasonable amount of balance and a higher branching factor, a lower number of clusters will perform better. This rule is consistent with two verifying experiments we performed using the artificial search space. In the first experiment, each move generates a cost of 1, the optimal solution cost is 16, and the first optimal solution is positioned at the far right side of the tree. As Figure 5 illustrates, a lower number of clusters is preferred as the branching factor increases. A higher amount of imbalance causes an increase in the optimal number of clusters. For the imbalance experiment, the solution is positioned in the middle of the tree.

C4.5 provides additional valuable insights that have been verified by experimentation in the artificial domain. The insights include the fact that operator ordering is more effective the farther to the right the so-

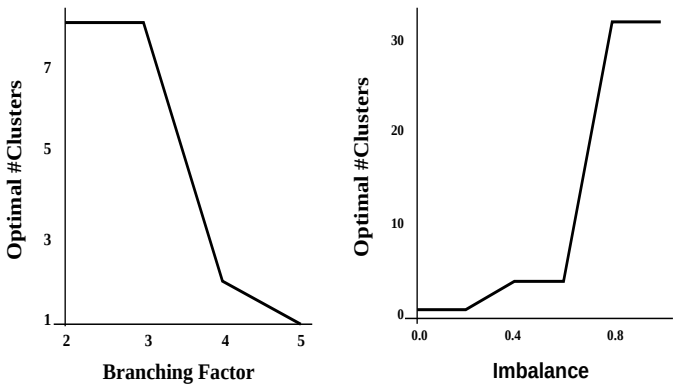

Figure 5: Artificial Experiments

lution lies, operator ordering causes an increase in the optimal number of clusters, load balancing becomes more necessary as imbalance increases, and a more accurate heuristic estimate function causes a decrease in the optimal number of clusters.

Conclusions and Future Work

This project reports on work performed to combine the benefits of parallel search approaches in the EUREKA system. Experimentation reveals that strategies developed over the last few years offer distinct benefits to improving the performance of AI applications, and strategies need to be chosen based on the characteristics of a particular problem. EUREKA employs the C4.5 machine learning system to make an intelligent choice of strategies for each new problem instance.

The EUREKA system can and will benefit from incorporation of additional search strategies, problem domain test cases and architectures. In particular, shared memory and distributed shared memory architectures, as well as additional multithreading utilities such as Java threads, can offer unique benefits to parallel search. Problem domains currently under investigation include additional combinatorial optimization problems such as the n-Queens problem and integration of machine learning and planning algorithms into this search architecture. We hope to demonstrate that parallel heuristic search algorithms can yield both optimal *and* scalable approaches to planning, machine learning, natural language, and theorem proving, and many other 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 International Conference on Parallel Processing.*

Cook, D. J., and Nerur, S. 1995. Maximizing the speedup of parallel search using hyps. In *Proceedings of the IJCAI Workshop on Parallel Processing for Artificial Intelligence*, 40–51.

Cook, D. J.; Hall, L.; and Thomas, W. 1993. Parallel search using transformation-ordering iterative-deepening A*. *The International Journal of Intelligent Systems* 8(8).

Cook, D. J. 1997. A hybrid approach to improving the performance of parallel search. In *Parallel Processing for Artificial Intelligence.* Elsevier Science Publishers.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing* 25:133–143.

Kumar, V., and Rao, V. N. 1990. Scalable parallel formulations of depth-first search. In Kumar; Kanal; and Gopalakrishan., eds., *Parallel Algorithms for Machine Intelligence and Vision.* Springer–Verlag. 1–41.

Mahanti, A., and Daniels, C. 1993. Simd parallel heuristic search. *Artificial Intelligence* 60(2):243–281.

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.*

Powley, C., and Korf, R. E. 1991. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13(5).

Powley, C.; Ferguson, C.; and Korf, R. E. 1993. Depth-first heuristic search on a simd machine. *Artificial Intelligence* 60(2):199–242.

Quinlan, J. R. 1993. *C4.5: Programs For Machine Learning.* Morgan Kaufmann.

Rao, V. N.; Kumar, V.; and Ramesh, K. 1987. A parallel implementation of Iterative-Deepening-A*. In *Proceedings of AAAI*, 178–182.

Reinefeld, A., and Schnecke, V. 1994. AIDA* - asynchronous parallel IDA*. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, 295–302.