A Hybrid Approach to Improving the Performance of Parallel Search *

Diane J. Cook

Department of Computer Science and Engineering
Box 19015
University of Texas at Arlington
Arlington, TX 76019
(817) 273-3606
cook@centauri.uta.edu

Many artificial intelligence techniques and applications rely on performing heuristic search through large problem spaces. Iterative-Deepening-A* (IDA*) search has proven to be effective for large search spaces, because it requires no intermediate state storage and is guaranteed to find optimal solutions. However, the time taken to perform IDA* search on real-world tasks often prevents the everyday usage of AI techniques.

Parallel processing can considerably reduce the time spent in search, and can thereby speed up AI applications. This paper describes HyPS, a hybrid parallel window / distributed tree search algorithm. Using this algorithm, the set of available processors is divided into clusters. Each cluster searches simultaneously through the same search space, but to a unique cost threshold. Within each cluster, the search space is divided so that an individual processor will search a fraction of the total search space. Operator ordering and load balancing techniques are used to further improve the performance of HyPS. Results on two real-world and one artificial domain show a substantial performance improvement over serial search algorithms, and indicate an improvement over existing parallel search approaches. In this paper we also describe a mechanism for automatically selecting the optimal number of clusters to use.

1. INTRODUCTION

Heuristic search provides the driving force for many applications of artificial intelligence including problem solving, robot motion planning, concept learning, theorem proving and natural language understanding [9,21]. Computational complexity is a major limitation of search, thus the research community is continually trying to develop more efficient search algorithms.

Parallel search algorithms significantly increase the size of the search space that can be traversed in a given amount of time. Parallel search techniques have been implemented on MIMD [3,20,21] and SIMD [4,7,8,12,19] architectures. Because of the overwhelming size of real-world AI applications, and because of the increasing power and accessibility

*This work is supported by National Science Foundation grants IRI-9308308 and IRI-9502260.

of parallel computers, there exists a constant need for improvement of parallel search algorithms.

This paper introduces a hybrid parallel search technique (HyPS) that improves the performance of search using a MIMD architecture. The idea behind the approach is to blend the strengths of existing parallel search algorithms. In particular, HyPS merges the power of a parallel window search with that of a distributed tree search. The resulting algorithm offers improvements over either approach used by itself. The addition of load balancing and operator ordering further improves the performance of the HyPS system.

The remainder of this paper describes the HyPS system and demonstrates its performance on two well-known AI application areas as well as an artificially-generated search space. Section 2 defines the search problem and describes the basic search techniques employed by HyPS, and section 3 provides an overview of existing parallel search techniques. The following section introduces the HyPS approach along with operator ordering and load balancing extensions. Section 5 demonstrates the performance of HyPS applied to two real-world and one artificial search-intensive problems. Section 6 overviews a technique for automatically selecting the number of clusters to use for a given problem. We conclude this paper with observations and a discussion of future research directions.

2. HEURISTIC SEARCH

Heuristic search techniques are used to find a sequence of operators that lead from the initial state (root node) of a problem to a goal state (goal node). An example search tree is given in Figure 1. Search begins by evaluating the root node and generating the children of the root node. At each later step, one of the previously generated child nodes is evaluated and its children are generated. This process continues until a node is evaluated that meets the goal criteria. Search time can be reduced by using estimated distances to the goal (heuristics) to direct the search. A* is a type of heuristic search algorithm which selects a node from the search queue that minimizes the function $f(n) = g(n) + h(n)$. In this function, $g(n)$ represents the cost of the path from the initial state to node n, and $h(n)$ represents the heuristic estimate of the least-cost path from node n to a goal node. If the function $h(n)$ never overestimates the distance to the goal, A* is guaranteed to find an optimal (least-cost) solution.

The main drawback of any heuristic search algorithm is that the memory requirement is exponential in the depth of the tree [17]. The resulting demand for memory often exceeds the resources of available machines.

Iterative-Deepening-A* (IDA*) search [10] provides a solution to this problem. IDA* performs a series of incrementally-deepening depth-first searches through the search space. In each iteration through the space, the depth of the search is controlled by the cost threshold. The cost of a node is calculated using the A* function $f(n) = g(n) + h(n)$. The initial cost threshold is computed as the estimated distance from the root node to the goal. Search down any branch of the space terminates when the $f(n)$ value of a generated child node exceeds the cost threshold for that iteration. If a goal node is not found during a given iteration, the search starts back at the root node, but the cost threshold is set to the minimum $f(n)$ value in the search space that exceeded the previous threshold. Successive iterations continue until a goal node is found.

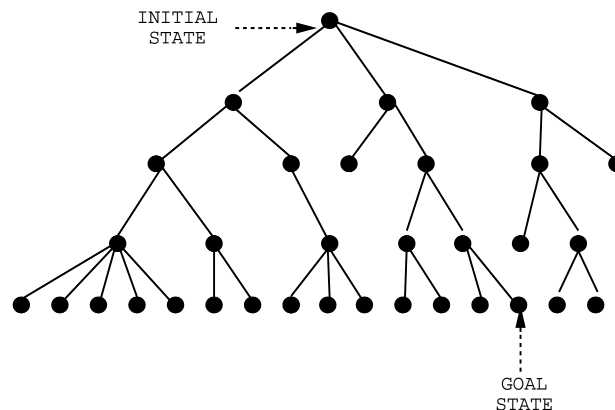

Figure 1. Application search tree

IDA* also guarantees optimal solutions if $h(n)$ does not overestimate the distance to a goal. Although redundant work is performed, the number of nodes that are expanded more than once is a small fraction of the total work. Unlike A* search, the memory requirement for IDA* is linear in the depth of the search space.

3. PARALLEL SEARCH

A number of researchers have explored methods for improving the efficiency of search. These methods include making use of background knowledge, learning search macro operators, and designing parallel search algorithms. Parallel architectures have been shown to be useful in reducing the amount of serial time spent in search by dividing the work among multiple processors. Many of the existing parallel search efforts can be classified as parallel window searches or distributed tree searches.

3.1. Parallel Window Search

One popular type of parallel search algorithm is a parallel window search (PWS), introduced by Powley and Korf [20]. Using PWS, each processor is given a copy of the entire search tree and a unique window size or 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 threshold. The first processor to reach a solution informs the rest of the processors. The remaining processors stop their search if this solution is acceptable. If an optimal solution is desired, processors searching at lower thresholds finish their current iteration and the least-cost goal is returned. Given the search tree shown in Figure 1, the division of work using PWS is illustrated in Figure 2.

One advantage of parallel window search is that the redundant search inherent in IDA* is not performed serially. A serial algorithm would search the space to threshold t, then

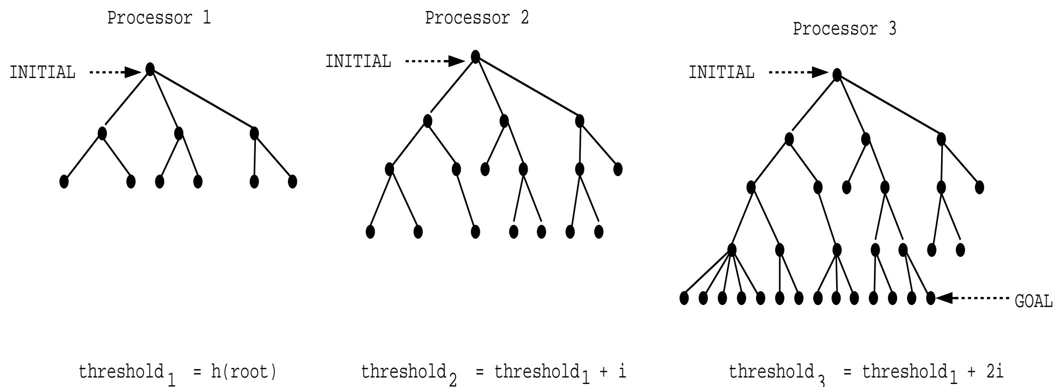

Figure 2. Division of work in parallel window search

would search the same space to threshold t plus some increment $(t+i)$, then to threshold $t+2i$, etc., until a goal node is found. On each iteration, all of the nodes expanded in the previous iteration are expanded again. Using multiple processors, this redundant work is performed concurrently.

A second advantage of parallel window search is the improved time in finding a first solution. If the solution density is high (there are many solutions in the search space), the depth-first search used by IDA* may find a deep solution much more quickly than an optimal solution. Parallel window search can take advantage of this type of search space. Processors that are searching beyond the optimal threshold may find a solution down the first branch they explore, and can return that solution long before other processors finish their search iteration. This will often result in superlinear speedup in comparison to the serial algorithm, because the serial algorithm will always increment the cost threshold by the least possible amount and not look beyond the current threshold.

On the other hand, parallel window search can face a decline in efficiency when the number of processors is significantly greater than the number of iterations required to find an optimal (or a first) solution. This situation will occur when a machine is used that offers many processors, yet few iterations are required because the heuristic estimator is fairly accurate.

3.2. Distributed Tree Search

An alternative approach to parallel search distributes the search space among available processors and is referred to here as distributed tree search (DTS) [11,21]. Using our version of DTS, one processor traverses the search tree until there are at least as many nodes in the queue as there are available processors. Once a sufficient number of nodes have been generated, the first processor passes a unique node from the search queue to the remaining processors. Each processor is thus responsible for the entire subtree rooted at the node it received. The processors perform IDA* on their unique subtrees simultaneously. All processors search to the same threshold. After all processors have

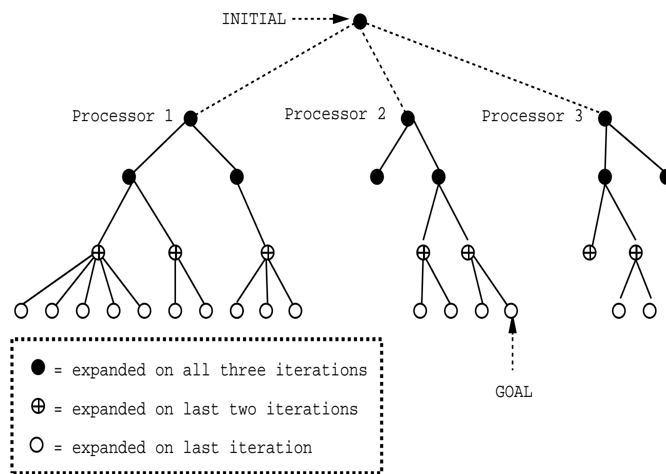

Figure 3. Division of work in distributed tree search

finished a single iteration, they begin a new search pass through the same set of subtrees using a larger threshold. Given the search tree shown in Figure 1, processors running DTS search the spaces shown in Figure 3.

One advantage of DTS is that no processor is performing wasted work beyond the goal depth. As the algorithm searches the space completely to one threshold before starting the search to a new threshold, none of the processors is ever searching at a level beyond the level of the optimal solution. It is possible, however, for DTS to perform wasted work at the goal depth. For example, in Figure 3 processor 3 searches nodes at the goal level that would not be searched in a serial search algorithm moving left-to-right through the tree.

A disadvantage of DTS is the fact that processors are often idle. This is because a processor that finishes an iteration quickly must wait for all other processors to finish before starting the next iteration (in order to ensure optimality). This idle time can make the system very inefficient and reduce the performance of the search application. The efficiency of this approach can be improved by performing load balancing between neighboring processors working on the same iteration. Load balancing itself will generate overhead and communication costs, however. A second disadvantage of DTS is that it does not provide a mechanism for finding a quick first solution.

The original DTS algorithm described by Kumar and Rao assigns the root node of the space to the first processor, and load balancing must be performed for other processors to receive initial work. We modified this algorithm to reduce load balancing time by distributing nodes to all processors from the host.

3.3. Other Parallel Search Techniques

Both parallel window search and distributed tree search are implemented on MIMD architectures. Additional MIMD and SIMD approaches have been suggested which have

not yet been incorporated into the HyPS system. Bidirectional search has been used to search the space forward from the initial state and backward from the goal state concurrently [13]. Variations on the distributed tree search technique have been tailored to specific applications such as robot path planning [2]. Yet another body of research focuses on concurrent random movements through a search space that cover the search space quickly [6].

Ferguson, Powley and Korf describe a SIMD search technique in which, like DTS, the work is distributed among processors and each of the processors search to the same threshold [18,19]. The distribution relies on a series of copies which takes O(log p) steps, where p represents the number of available processors. Another SIMD search techniques which relies on fast information distribution and a distributed tree search is the Massively Parallel IDA* (MIDA*) system [4]. MIDA* also takes O(log p) steps to perform distribution, but search is performed as the information is distributed, so that the amount of work performed during the parallel IDA* step is reduced. A third SIMD search techniques again uses distributed tree search, but focuses on techniques for load balancing between processors during the parallel IDA* stage [12].

4. HyPS

We introduce a new type of parallel search that combines characteristics of existing approaches. We refer to the algorithm as Hybrid Parallel Search (HyPS) [14–16]. In particular, HyPS combines parallel window search with distributed tree search. As the experiments will show, the combination of search techniques outperforms either technique used in isolation.

HyPS divides the set of processors into groups called *clusters*. Each cluster performs parallel window search — each cluster receives a copy of the entire search space, and each cluster searches that space to a unique cost threshold. Within each cluster, the space is distributed among the cluster's processors and the processors perform distributed tree search. In this way, clusters perform parallel window search, and the processors within each cluster perform distributed tree search.

HyPS has been implemented in C on a nCUBE and in C* on a Connection Machine 5. The next two subsections describe the distribution and search phases of the HyPS algorithm. The following subsections describe improvements that can be made using operator ordering and load balancing.

4.1. Distribution Phase

During the distribution phase, the host program distributes work among the node processors. The host program first determines how many clusters will be used and divides the set of processors equally among the clusters. The search tree is then expanded to the point that distinct subtrees can be distributed to each processor within a cluster.

The number of distinct subtrees being explored is equal to the number of processors available within a cluster. For example, if we have $n = 6$ available processors and we wish to use two separate parallel windows, then HyPS will form two clusters, each with $n/2 = 3$ processors. Hence, the host program will have to divide the initial tree into three distinct subtrees, which in turn are distributed to the three processors in each cluster (see Figure 4).

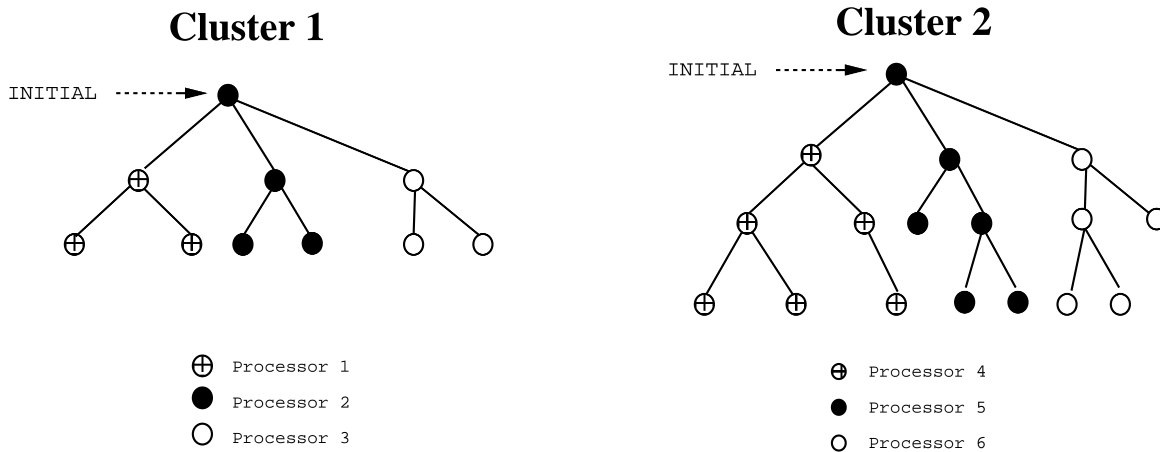

Figure 4. Space searched by two clusters, each with 3 processors

The host processor stores generated nodes in a queue (Q). The number of generated nodes is dependent upon the number of distinct subtrees required for the distributed tree search. The steps taken in the formation of the queue are detailed below.

1. Initialize Q to contain the root of the search tree.
2. While length(Q) $<$ number of processors in each cluster do
   - Remove node n from head of Q.
   - Evaluate node n. If n is not a goal node, then generate the children of n and append them onto Q.
3. If length(Q) $>$ number of processors in each cluster, *compress* nodes at end of Q.

Note that using a level-by-level search, more nodes may be generated than available processors. In this case, nodes at the end of the search queue are removed from the queue and replaced by their parent nodes. This compression is performed as many times as needed until the length of the queue is less than or equal to the number of processors in each cluster. If the final queue length is less than the number of processors needing information, child nodes from the most recent compression are distributed to the idle processors. In this case, both the parent node and some of its children are being searched by separate processors, yielding some redundancy in the search process.

We provide an example of the distribution algorithm for an instance of the Fifteen Puzzle problem where each cluster contains $p = 6$ processors. The expansion of the search tree is shown in Figure 5. In this picture, invalid nodes (the blank square cannot move beyond the board boundaries and cycles of length two are disallowed) are not shown. The final queue will contain the leaf nodes of this partial tree, and will be distributed

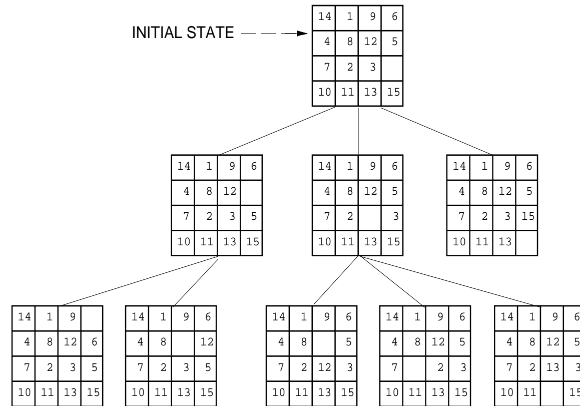

Figure 5. Distribution of work for a fifteen puzzle problem

to the respective processors in each cluster. Once distribution is complete, the signal is given to begin the search phase.

4.2. Search Phase

The search phase is performed in parallel by the processors within each cluster. To begin the search phase, the host sends the nodes defining the subtrees to the appropriate processors within each cluster. The cluster also receives the cost threshold to which it will search. Each processor then begins searching at the root of its subtree. The first cluster receives a cost threshold equal to the heuristic estimate of the distance from the initial state to the goal. All of the remaining clusters are given incrementally larger thresholds. For many problems, the appropriate increment can be predetermined in a way that guarantees no loss of optimality.

Figure 4 illustrates the case where $n = 6$ processors are available and we wish to use two separate parallel windows. By changing the cluster size, a greater or lesser number of windows can be employed. When the number of clusters c is equal to the number of processors available on the machine, HyPS simulates pure parallel window search. When c is equal to 1, HyPS simulates pure distributed tree search.

Because clusters search the space to unique depths and processors within each cluster search a unique portion of the space, it is unknown where the first solution found will lie, and whether its cost will be optimal or nonoptimal. HyPS can be used to specifically return an optimal solution or to return the first (optimal or nonoptimal) solution found.

4.3. Operator Ordering

Problem solutions can exist anywhere in the search space. Using IDA* search, the children are expanded in a depth-first search 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.

The left-to-right direction of search represents the order in which search operators are

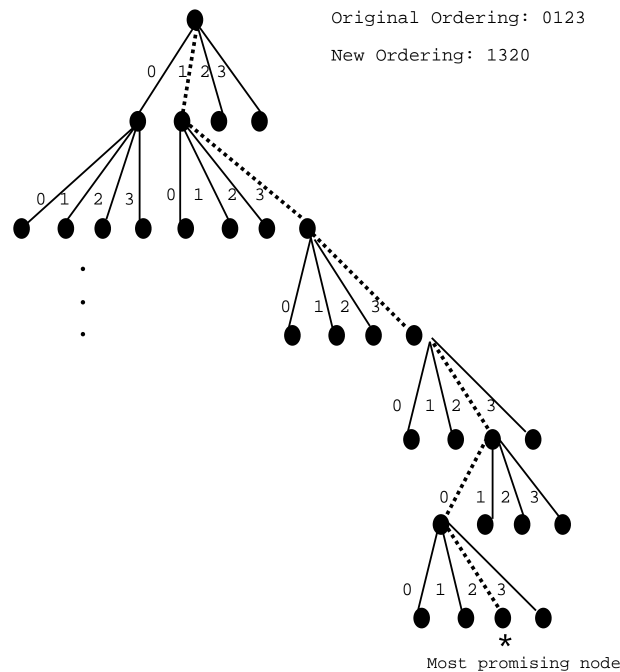

Figure 6. Operator ordering example

applied. 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. As an example, one of the tested Fifteen Puzzle problem instances can require expansion of 900,587 search nodes for the best ordering of operators, yet requires expansion of 4,570,051 search nodes for the worst operator ordering. HyPS attempts to find a good operator ordering for each iteration based on information gained from the previous iteration [3]. The reduction in number of expanded nodes can be exponential for some problem instances.

HyPS stores the order of operator applications that lead to the most promising node at the bottom of the subtree explored in the previous search pass. The "most promising" node is the one that has the minimum h value. Since all nodes at the bottom of a subtree represent equal f values (by definition of IDA*), nodes with smaller h values are expected to be more accurate. Consider the situation in which a car needs to be fixed. A customer generally feels more confidence about the statement that his car will be ready in 2 days ($h = 2$) if it has already been in the shop 28 days ($g = 28$), than he would feel about the statement that his car will be ready in 28 days ($h = 28$) if the car has only sat in the shop for 2 days ($g = 2$). If, as in many cases, the error in the heuristic function h grows monotonically with the distance from the goal, nodes with the lowest h values represent the most accurate estimated distances to the goal and contain fewer nodes in their subtrees [20].

The following example illustrates the operator ordering technique. Figure 6 shows the search tree expanded on one iteration of IDA*. The most promising leaf node is starred.

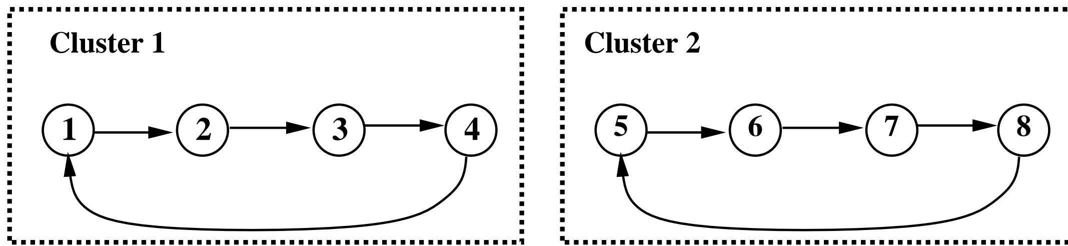

Figure 7. Load balancing of two clusters, each with 4 processors

The current operator order is 0123 (always try operator 0 first, operator 1 next, operator 2 next, and operator 3 last), but the sequence of operators that leads to the most promising node at the current cutoff point is 133202, so the operator order is changed to reflect this search direction. Because operator 1 appears first in the path to the most promising node, operator 1 will always be tried first. The next unique operator in the sequence is operator 3, then operator 2, and finally operator 0. This ordering is updated at the end of each parallel window search iteration.

4.4. Load Balancing Within a Cluster

In the same way that PWS is aided by the addition of operator ordering, DTS benefits from the addition of load balancing. Distributed tree search suffers greatly due to processor idling. Load balancing can be added to improve the intra-cluster performance by reducing this idle time. Using load balancing, a processor that has finished its current search iteration asks for work from its neighbor and helps that neighbor to complete its search iteration.

We have added load balancing to HyPS to improve the overall performance. Load balancing here is implemented using a ring model of communication. Consider the case where we have eight available processors divided into two clusters. Load balancing will be performed within each of these four-processor clusters. If processor p_1 finishes work, then p_1 asks for work from its right neighbor, p_2. Each processor performs a similar type of request except for the last processor in the ring, such as p_4, which requests work from p_1. Figure 7 depicts this communication model. When work is requested from a processor, the processor first determines if it has work to spare (the number of nodes on the stack is greater than a specified threshold). If there is work to spare, the donating processor sends information corresponding to the node at the bottom of the stack, or the node closest to the root which has children left to expand. If the processor has no work to spare, then the requesting processor must remain idle until the search is finished or a new threshold is specified. In this load balancing algorithm, communication is restricted to the adjacent processors within the cluster.

Load balancing was added to HyPS to see how the performance of DTS and PWS would be affected. To improve the performance further, more advanced load balancing techniques could be employed [1,23]. As the experiments demonstrate, integrating the

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 8. The fifteen puzzle

additional strategies of load balancing and operator ordering improve the performance of HyPS.

5. EXPERIMENTS

This section demonstrates the performance of HyPS on two real-world applications and one artificially-crafted domain. The experiments on real-world domains reveal that 1) HyPS does provide a substantial speedup over a serial algorithm, 2) operator ordering and load balancing can in many cases improve the performance of HyPS, and 3) the optimal number of clusters varies, verifying that PWS and DTS perform better in combination than in isolation. We will first introduce the two real-world applications and provide the experimental results, then describe our artificial domain experiments.

5.1. The Fifteen Puzzle

One well-known discrete optimization problem (DOP) is the Fifteen Puzzle. It consists of a 4×4 grid containing tiles numbered one to fifteen, and a single empty tile position called the *blank* tile. A tile can be moved into the blank position from an adjacent position, resulting in the four operators up, down, left, and right. Given the initial and goal configurations, the problem is to find a sequence of moves to reach the goal. An example of a goal configuration is given in Figure 8.

In our problem instances, the heuristic estimate function used is the *Manhattan Distance Function*, which is known to be an admissible heuristic for the Fifteen Puzzle problem. The Fifteen Puzzle problem can generate approximately 2^{40} unique problem states [7], thus this represents a large search problem which can benefit from an efficient parallel algorithm.

5.2. Robot Arm Motion Planning

Discrete optimization problems belong to the class of NP-hard problems. Parallel processing of these problems cannot reduce their worst-case run time to a polynomial without using an exponentially large number of processors. However, the average-time complexity of heuristic search algorithms for many of these problems is polynomial. Robot motion planning, speech understanding, and task scheduling are some of the DOPs which require real-time solutions. Parallel processing of these problems can bring their real-time

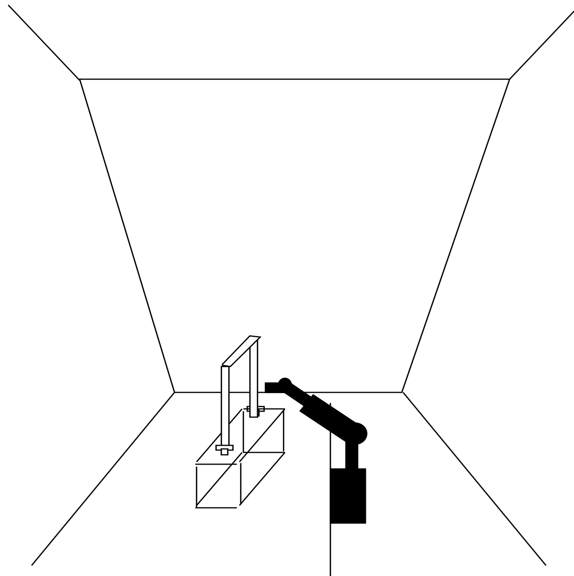

Figure 9. Robot arm motion planning problem instance

solutions closer to being realized.

Traditional motion planning methods are very costly when applied to a robot arm, because each joint has an infinite number of angles to which it can move from a given configuration, and because collision checking must be performed for each arm segment. Craig provides a detailed description of the calculations necessary to determine the position of the end effector in the 3D workspace given the current joint angles [5]. For our experiments, we use the parameters defined for the Puma 560 robot arm with six degrees of freedom shown in Figure 9. The size and layout of the room is the same for each of our test problems.

5.3. Results from real-world experiments

We tested the performance of HyPS for the Fifteen Puzzle and Robot Arm Motion Planning domains using the basic HyPS approach as well as HyPS with operator ordering and load balancing added. Each of these experiments were run on 64 processors of an nCUBE 2.

Table 1 lists the speedup that resulted from each selected number of clusters, averaged over 20 instances of the Fifteen Puzzle problem. Speedup over the corresponding single-processor algorithm was generated using basic HyPS, HyPS with operator ordering, HyPS with load balancing, and HyPS with both operating ordering and load balancing. Results were collected for finding a first solution and an optimal solution. In all of the result summaries, the speedup of the best cluster distribution is highlighted.

As table 1 indicates, all versions of the program realize significant speedup over the serial algorithm. Speedup is greater when the program stops with a first solution, because

Table 1
Speedup for Fifteen Puzzle domain

	Number of clusters								
Algorithm	1 (DTS)	2	4	8	16	32	64 (PWS)	Avg	*Solution*
Basic	12.4	16.6	**24.2**	16.8	6.1	6.5	3.0	12.2	Optimal
	12.4	22.0	45.9	**46.4**	33.4	20.3	22.2	28.9	First
Operator Ordering	18.4	**43.1**	**43.1**	16.2	6.2	5.4	3.9	19.5	Optimal
	18.4	44.2	82.3	**85.3**	53.5	43.4	29.5	50.9	First
Load Balancing	17.6	**28.2**	24.9	17.3	9.2	7.5	2.7	15.3	Optimal
	17.6	30.6	47.1	**57.3**	36.7	26.8	16.3	33.2	First
Operator Ordering /	**41.3**	23.1	40.7	19.6	6.5	5.5	2.7	19.9	Optimal
Load Balancing	41.3	24.3	79.2	74.8	**87.3**	41.1	16.5	52.1	First

the serial algorithm is constrained to always find an optimal solution. In addition, both operator ordering and load balancing improve the speedup of HyPS, and the best speedup is realized when all of these techniques are used. Superlinear speedup may occur when finding a first solution because the comparison is made to a serial algorithm finding an optimal solution. Superlinear speedup may also result when the goal node lies on the right side of the space, because the space is divided evenly among processors and an individual processor will find that solution before a serial algorithm that must search the entire left side of the space before reaching the goal.

Table 2 lists the number of problems for which the corresponding number of processors yields the best performance. Note that in no case does pure DTS (number of clusters = 1) or pure PWS (number of clusters = 64) consistently perform best. In general, performance improves with a greater number of clusters when looking for a first solution. This occurs often in domains like the Fifteen Puzzle where the solution density (percentage of nodes representing goals) is high, and increasing the number of windows increases the chance of quickly finding a goal.

As expected, operator ordering and load balancing affect the ideal number of clusters. Because operator ordering benefits PWS, the optimal number of clusters increases for these tests. When load balancing is used in isolation, the performance of DTS improves and the optimal number of clusters becomes lower. The greatest number of clusters and the greatest speedup is realized when both operator ordering and load balancing are used to find a first solution.

The results for the Robot Arm Motion Planning domain are listed in table 3. The results are averaged over eight problem instances in which only optimal solutions are sought. As before, speedup is obtained over the serial algorithm and performance improves when operator ordering and load balancing are added. However, this domain differs from the Fifteen Puzzle domain in two main features. First, many of the solutions in this domain lie on the far right side of the tree. For this reason, superlinear speedup is often realized when operator ordering is employed. Second, the branching factor is nearly constant throughout

Table 2
Optimal number of clusters for Fifteen Puzzle domain

	Number of clusters								
Algorithm	1	2	4	8	16	32	64	Avg	*Solution*
Basic	4	7	3	6	0	0	0	3.9	Optimal
	1	3	6	6	3	0	1	10.1	First
Operator Ordering	1	6	7	3	1	2	0	7.3	Optimal
	0	3	7	5	3	1	1	10.9	First
Load Balancing	5	7	3	4	0	1	0	3.2	Optimal
	2	3	5	8	1	0	1	8.6	First
Operator Ordering /	3	6	6	3	1	1	0	5.6	Optimal
Load Balancing	0	1	8	6	3	1	1	11.3	First

Table 3
Speedup for Robot Arm Motion Planning domain

	Number of clusters							
Algorithm	1 (DTS)	2	4	8	16	32	64 (PWS)	Avg
Basic	14.7	21.4	**70.2**	27.1	10.6	2.1	2.1	18.5
Operator Ordering	13.1	38.7	230.6	891.7	**1671.1**	2.0	2.0	356.2
Load Balancing	12.9	21.2	**78.0**	27.0	14.1	2.3	2.1	19.7
Operator Ordering / Load Balancing	12.8	84.9	155.0	**1518.4**	354.1	2.8	1.8	266.2

Table 4
Optimal number of clusters for Robot Arm Motion Planning domain

	Number of clusters							
Algorithm	1	2	4	8	16	32	64	Avg
Basic	0	0	7	1	0	0	0	4.5
Operator Ordering	0	0	0	7	1	0	0	9.0
Load Balancing	0	0	7	1	0	0	0	4.5
Operator Ordering / Load Balancing	0	0	1	6	1	0	0	8.5

the tree, so the tree is very uniform. As a result, the load is balanced and addition of load balancing does not greatly improve performance. In fact, in some cases load balancing can actually degrade performance due to the time taken to query neighboring processors for available work.

Table 4 lists the number of problems for which a given number of clusters performed best. As before, operator ordering causes an increase in the ideal number of processors. Because load balancing did not significantly change performance, the ideal number of clusters remains the same when load balancing is added. For both of these problems, the optimal number of clusters was not constant, but varied with characteristics of the problem domain and with the inclusion of operator ordering and load balancing.

5.4. Artificial Domain

Intuitively, selecting the number of clusters to use for a given search application seems to be affected by factors such as the branching factor of the tree, the load imbalance, the accuracy of the heuristic function, and the position of the solution in the tree. To isolate the factors that affect the performance of parallel search, we constructed an artificial search space. In this section, we will describe the artificial search space and show results of experiments using this space.

The artificial search space is represented as a tree. Instead of storing the entire space in memory, portions of the space are generated as needed while the algorithm traverses the tree. The following characteristics of the tree can be adjusted by the user.

- Maximum branching factor. The user can select a 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, and thus in the distributed work. 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 $error_h * distance$.

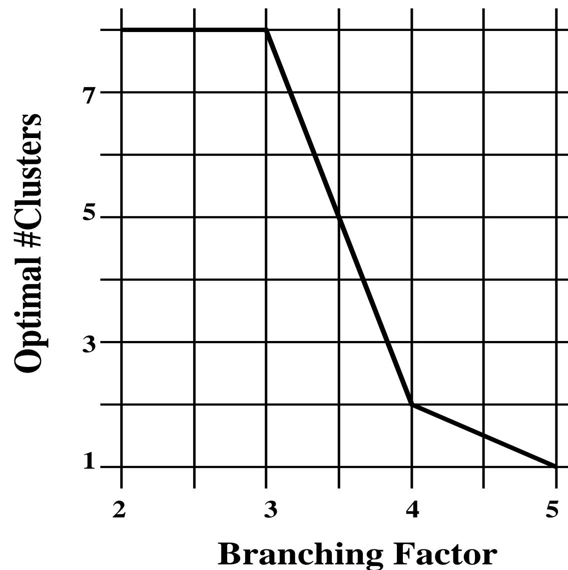

Figure 10. Artificial Experiment #1

- 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.

Using this artificial domain, we can test several hypotheses about the features of a search space that affect the optimal number of clusters to be used. Note that there are no random distributions used in the generation of the tree, because the tree must maintain the same structure for each iteration of IDA*. Instead, every feature of a node is determined as a function of the node's position in the search tree. All of the experiments described in this section were conducted using 32 processors on an nCUBE 2.

The first experiment compares the optimal number of clusters to the average branching factor in the tree. When the average branching factor is low, there is little work to distribute and better performance is obtained by parallelizing iterations of IDA* using PWS. As the branching factor increases, there is more work to distribute and a lower number of clusters is favored.

To verify this hypothesis, we ran HyPS on four different artificially-generated search spaces with the average branching factor ranging from 2 to 5. In each case, the tree is perfectly balanced, the heuristic estimate is 0.8 (fairly accurate), the optimal cost is 16, and the first optimal solution is positioned on the far right of the tree. No operator

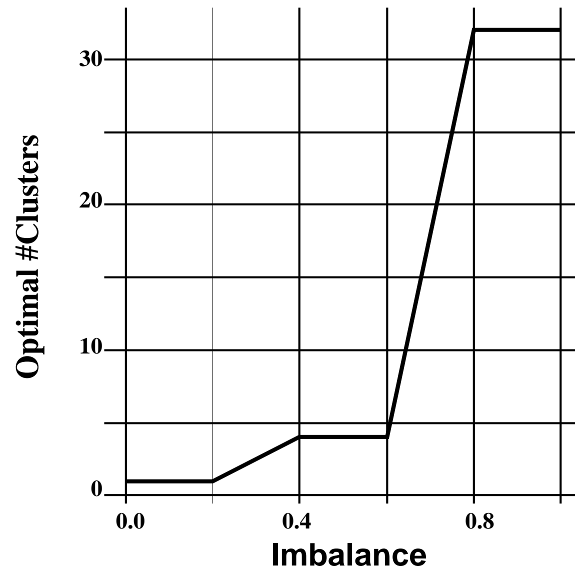

Figure 11. Artificial Experiment #2

ordering or load balancing is employed. As figure 10 shows, the hypothesis is verified by our experiment. When the branching factor is at the low end, the ideal number of clusters is 8. As the branching factor increases, the optimal number of clusters decreases eventually down to 1.

In the second experiment, we verify that a nonuniform tree requires a greater number of clusters. Figure 11 shows the result of this experiment. We ran HyPS on six trees with load imbalance ranging from 0 to 1. For each tree, the solution is positioned in the middle of the tree, the branching factor is 3, the optimal cost is 15, and no load balancing or operator ordering is used. As the imbalance increases, more windows are required. This is due to the fact that with a high level of imbalance, many of the subtrees are very small and the processors searching these trees will be idle.

As the first experiment in this section demonstrated, a high branching factor results in a lower optimal number of clusters. In contrast, a high number of IDA* iterations and a solution positioned on the left side of the tree will result in a higher optimal number of clusters. The third experiment controls the number of IDA* iterations required by varying the accuracy of the heuristic estimator function and by varying the position of the first optimal solution. For each run, the branching factor is 3 and the optimal cost is 15. While the accuracy of h varies from 0.2 to 1.0, the position of the first optimal solution also varies from the far left side of the tree to the far right side of the tree. Figure 12 shows that in fact with a low accuracy and solution on the left, more clusters are needed than when h is accurate and the first optimal solution appears on the right side of the tree.

The last experiment focuses on the effects of employing operator ordering. In experi-

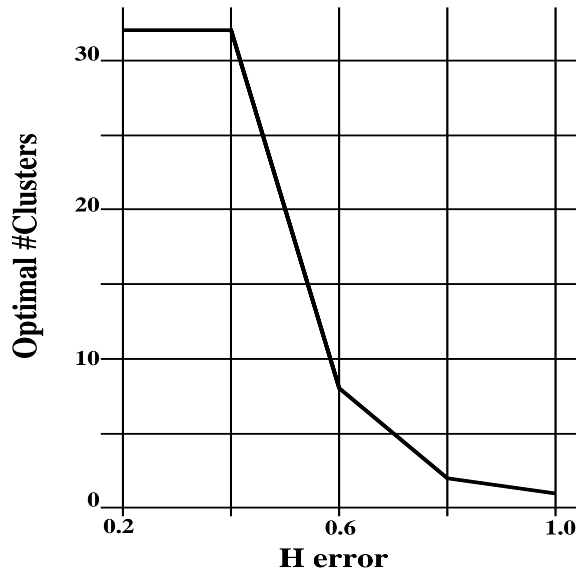

Figure 12. Artificial Experiment #3

ment four, we demonstrate that operator ordering becomes more effective the farther to the right the solution lies. Each search tree for experiment four has a branching factor of 4, a heuristic estimate of 0.2, a uniform tree, and an optimal cost of 12. The solution is first position 0/5 of the way through the tree (far left), and moved successively further to the right (positions of 1/5, 2/5, 3/5, 4/5, and 5/5 through the tree). For each test case, HyPS is run with and without operator ordering. Figure 13 graphs the change in optimal number of clusters that is required when operator ordering is used to guide search, and plots the resulting speedup when operator ordering is employed. As the figure verifies, operator ordering causes a greater increase in the ideal number of clusters and a more significant speedup as the solution is positioned farther to the right in the tree. The figure on the left graphs the increase in optimal number of clusters with the addition of operator ordering, and the figure on the right graphs the speedup that results from employing operator ordering.

In this section we have used an artificial domain to identify features that contribute to optimizing the performance of HyPS. In the next section, we will describe an architecture that uses these features and a set of rules to automatically select the number of clusters.

6. AUTOMATICALLY SELECTING OPTIMAL NUMBER OF CLUSTERS

In this section we describe our method of automatically selecting the number of clusters that will optimize the performance of HyPS. To make this decision, we note that many characteristics of a search space can be determined by looking at the first few levels of a tree. In particular, by exploring just a few levels down in the tree we can estimate factors such as the average branching factor, the average error of the heuristic estimate function,

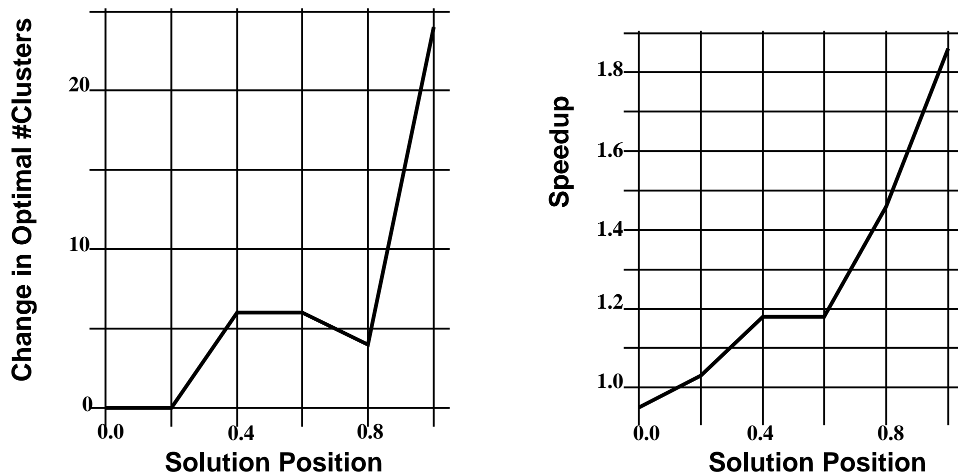

Figure 13. Artificial Experiment #4

the possible location of the optimal solution, and the amount of load imbalance.

Our architecture makes use of the host distribution mechanism to collect statistics about the search space. The amount of time required to search the first few levels of the tree is minimal, and valuable information can be gleaned about the nature of the problem. Using results from the real-world and artificial domains, rules were constructed to select the ideal number of clusters. Statistics are gathered about the problem space as the host searches a few levels down in the tree. The statistics are used to calculate the number of clusters for HyPS to use for the rest of the search. This method of gathering information about the search space to make decisions about the parallel search is similar to Suttner's SPS model, in which a small portion of the space is searched serially to determine whether the problem is a good candidate for parallelization, to control redundancy, and to determine an effective distribution scheme [22].

Using this architecture, we reran the artificial domain experiments and compared results to those shown in section 5.4. In particular, figures 14 through 17 graph the results of experiments 1 through 4 where the number of clusters is selected automatically by the system. In each case, the trend in selected number of clusters is the same as the trend for the verified optimal number of clusters, though actual numbers vary in a few cases.

We also reran the two real-world application domains, using this architecture to select the number of clusters for HyPS. We compared the number of clusters pick by our architecture to the optimal number of clusters verified by the experiments in section 5. For the Fifteen Puzzle, the average error in optimal number of cluster selection averaged over all 20 problem instances for the basic HyPS system was 1.1. This represents one *selection* away from optimal: choosing to use 2 clusters instead of 1, or 4 clusters instead of 8. For these results, the host system searched six levels into the tree. Similarly, the host searched six levels into the search tree for the Robot Arm Motion Planning domain. Averaged over the six problem instances using basic HyPS, the error in optimal number

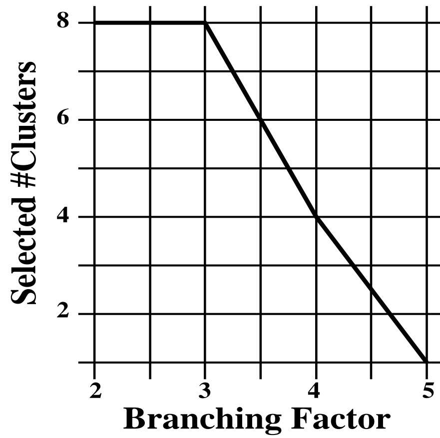

Figure 14. Artificial Experiment #1

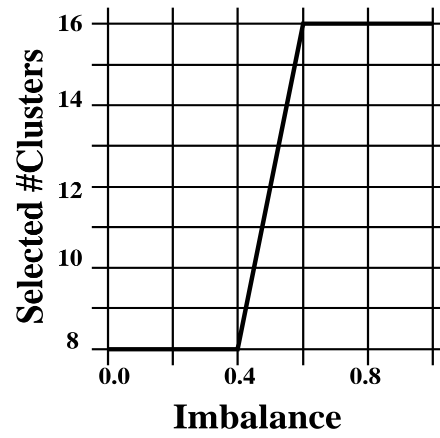

Figure 15. Artificial Experiment #2

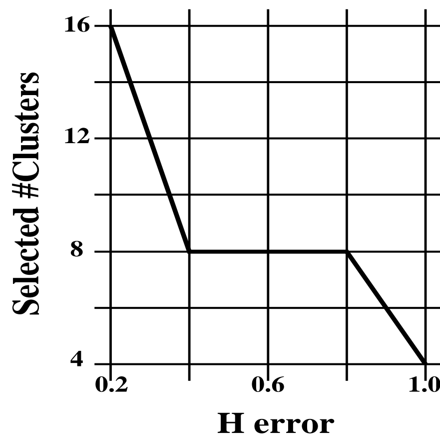

Figure 16. Artificial Experiment #3

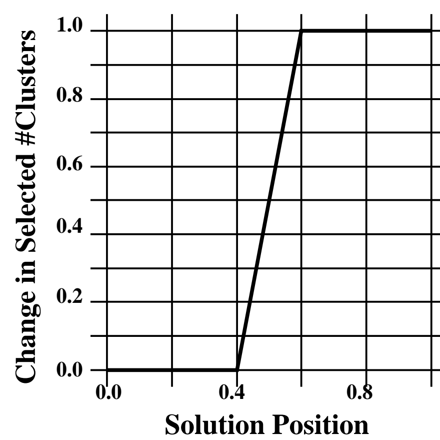

Figure 17. Artificial Experiment #4

of clusters selection using our architecture was 0.125.

7. CONCLUSIONS AND FUTURE WORK

The goal of this research is to improve search-intensive applications by improving the performance of parallel search algorithms. This paper introduces the Hybrid Parallel Search algorithm HyPS, which improves the performance of existing parallel search techniques such as parallel window search and distributed tree search, by combining the benefits of both approaches. The system combines the two techniques by dividing the set of processors into clusters. The clusters perform parallel window search, while the processors within a cluster perform distributed tree search. The performance of HyPS is improved by the addition of operator ordering (with strengthens inter-cluster performance) and load balancing (which strengthens intra-cluster performance).

The results of experiments applied to two problem domains reveal that HyPS outperforms both serial search algorithms and either PWS or DTS used in isolation. An artificial domain was used to identify the features that contribute to the selection of the optimal number of clusters. Finally, we introduced a mechanism for automatically selecting the number of clusters for each search domain that performed well in all three application domains.

While the HyPS system has been shown to be effective in these three domains, we are continuing to apply the system to a variety of applications. As more data is gathered, we can use machine learning techniques to generate a set of parameter-setting rules for HyPS that achieves the greatest performance possible for parallel heuristic search.

REFERENCES

1. I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, 1991.
2. D. Challou, M. Gini, and V. Kumar. Parallel search algorithms for robot motion planning. In *Proceedings of the AAAI Symposium on Innovative Applications of Massive Parallelism*, pages 40–47, 1993.
3. D. J. Cook, L. Hall, and W. Thomas. Parallel search using transformation-ordering iterative-deepening A*. *The International Journal of Intelligent Systems*, 8(8), 1993.
4. D. J. Cook and G. Lyons. Massively parallel IDA* search. *Journal of Artificial Intelligence Tools*, 2(2):163–180, 1993.
5. J. J. Craig. *Introduction to Robotics.* Addison-Wesley Publishing Company, Inc, 1989.
6. W. Ertel. Massively parallel search with random competition. In *Proceedings of the AAAI Symposium on Innovative Applications of Massive Parallelism*, pages 62–69, 1991.
7. M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25:133–143, 1995.
8. G. Karypis and V. Kumar. Unstructured tree search on simd parallel computers. In *Supercomputing 92*, pages 453–462, 1992.

9. H. Kitano. Massively parallel ai and its application to natural language processing. In *Proceedings of the First International Workshop on Parallel Processing for AI*, pages 99–105, 1991.
10. R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
11. V. Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Kumar, Kanal, and Gopalakrishan, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Springer–Verlag, 1990.
12. A. Mahanti and C. Daniels. Simd parallel heuristic search. *Artificial Intelligence*, 60(2):243–281, 1993.
13. P. C. Nelson and A. A. Toptsis. Superlinear speedup using bidirectional and islands. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 129–134, 1991.
14. S. Nerur and D. J. Cook. Maximizing the speedup of parallel search using hyps. In *Proceedings of the Third International Workshop on Parallel Processing for Artificial Intelligence*, pages 40–51, 1995.
15. S. S. Nerur. A hybrid parallel IDA* search. In *Proceedings of the National Conference on Artificial Intelligence*, 1994.
16. S. S. Nerur and D. J. Cook. A hybrid parallel-window/distributed tree algorithm for improving th e performance of search-related tasks. In *Proceedings of the Seventh International Conference on Industrial and Engineering Application of Artificial Intelligence and Expert Systems*, pages 629–637, 1994.
17. J. Pearl. *Heuristics*. Addison-Wesley, Reading, Massachusetts, 1984.
18. C. Powley, C. Ferguson, and R. E. Korf. Parallel tree search on a simd machine. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 249–256, 1991.
19. C. Powley, C. Ferguson, and R. E. Korf. Depth-first heuristic search on a simd machine. *Artificial Intelligence*, 60(2):199–242, 1993.
20. C. Powley and R. E. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5), 1991.
21. V. N. Rao, V. Kumar, and K. Ramesh. A parallel implementation of Iterative-Deepening-A*. In *Proceedings of AAAI*, pages 178–182, 1987.
22. C. B. Suttner. Static partitioning with slackness. In *Proceedings of the IJCAI Workshop on Parallel Processing for Artificial Intelligence*, pages 143–155, 1995.
23. J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18(1):1–13, 1993.