Exploiting Algebraic Structure in Parallel State Space Search

Jonathan Bright* **Simon Kasif†** **Lewis Stiller‡**

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

Abstract

In this paper we present an approach for performing very large state-space search on parallel machines. While the majority of searching methods in Artificial Intelligence rely on heuristics, the parallel algorithm we propose exploits the algebraic structure of problems to reduce both the time and space complexity required to solve these problems on massively parallel machines. Our algorithm runs in $O(N^{1/4}/P)$ time using $O(N^{1/4})$ space with P processors where N is the size of the state space and P is the number of processors. The technique we present is applicable to several classes of exhaustive searches. Applications include the knapsack problem and the shortest word problem in permutation groups which is a natural generalization of several common planning benchmarks such as Rubik's Cube and the n-puzzle.

Introduction

The best-known algorithm for finding optimal solutions for general planning problems on sequential computers is IDA* developed by R. Korf (Korf 1985a; 1985b). IDA* has an exponential worst-case complexity and its efficiency heavily depends on the ability to synthesize good heuristics. There are many attempts to parallelize state space search (Evett *et al.* 1990; Powley & Korf 1991; Powley, Ferguson, & Korf 1991; Powley & Korf 1988; Rao & Kumar 1987). In this paper we examine several problems for which the computational cost of exploring the entire space of possible states may be prohibitive and derivation of good heuristics is difficult. Exploitation of the algebraic structure of the problems we are considering substantially reduces the time and space complexity of the algorithm. For example, we consider problems that have brute-force $O(k^n)$ solutions (where n is the size of the input to the algorithm) and suggest parallel solutions whose time complexity is $O(k^{n/2}/P)$ (where P is the number of processors) and whose space complexity is $O(k^{n/4})$. While the time-space complexity remains exponential, these algorithms are capable of solving problems that were not tractable for conventional architectures. There are several examples where reducing the time complexity of an algorithm from $O(k^n)$ to $O(k^{n/2})$ has resulted in significant progress. Chess is one such example since it is known that alpha-beta at best accomplishes this type of complexity reduction. Thus our main goal is to double the depth of the search current technology can perform. We feel that this class of problems seem to be particularly well matched with massively parallel machines such as the CM-2 or CM-5 (Thinking Machines Corporation 1992; Hillis 1985).

Our approach is influenced by the elegant sequential algorithm proposed by Schroeppel and Shamir and the later generalization by Fiat, Moses, Shamir, Shimshoni and Taros, which we will review in the next section (Schroeppel & Shamir 1981; Fiat *et al.* 1989). We view our work as a direct parallel implementation of this algorithm.

In this paper we describe parallel algorithms using the shared memory (CREW PRAM) model of parallel computation. This model allows multiple processors to read from the same location and therefore hides the cost of communication. While this assumption is unrealistic in practice it allows us to simplify the description of a relatively complex algorithm. Detailed analysis of our algorithms suggests that they can be expressed by efficient composition of computationally efficient primitives such as sorting, merging, parallel prefix and others (Stiller 1992).

Review of the Schroeppel and Shamir Algorithm

In this section we consider the knapsack (decision) problem to illustrate this approach. This is a canonical NP-complete problem of the "monotonic and decomposable" genus proposed by Schroeppel and Shamir. The knapsack problem has also been used as the basis for certain cryptographic schemes and other applications (Merkle & Hellman 1978; Diffie & Hellman 1976;

*Supported by the 1993 CESDIS Cray Research Earth and Space Science Fellowship

†Supported in part by NSF/DARPA Grant CCR-8908092

‡Supported by U.S. Army Grant DAAL03-92-G-0345

Tarjan & Trojanowski 1977). The input to the algorithm is a list of integers S and a number x. The output is a sublist of S, call it S', such that the sum of the elements in S' is equal to x. This problem has applications in scheduling, cryptography, bin packing and other combinatorial optimization problems. For example, we can use solutions to this problem to minimize the total completion time of tasks of fixed duration executing on two processors.

Let n the size of S. We will introduce some helpful notation. Let A and B to be sets of integers. Define $A + B$ to be the set of integers c such that $c = a + b$ where $a \in A$ and $b \in B$. Define $A - B$ to be the set of integers c such that $c = a - b$ where $a \in A$ and $b \in B$.

Observation 1:
The knapsack problem can be solved in $O(n2^{n/2})$ time and $O(2^{n/2})$ space.

Proof: We partition S into disjoint lists S_1 and S_2 such that the size of S_1 is $n/2$. Let G_1 and G_2 be the lists of all sums of sublists of elements in S_1 and S_2 respectively. Clearly, our problem has a solution iff the list G_1 has a non-empty intersection with $\{x\} - G_2$. However, note that we can sort both lists and find the intersection by merging. Thus, the time complexity of this algorithm can be seen to be $O(n2^{n/2})$ using any optimal sorting algorithm and noting the trivial identity $2\log(2^{n/2}) = n$. Unfortunately, the space complexity is also $O(2^{n/2})$ which makes it prohibitive on currently available machines for many interesting problems. This approach was first suggested by Horowitz and Sahni (Horowitz & Sahni 1974). However, in the AI literature the algorithm has a strong similarity to bi-directional search studied by Pohl (Pohl 1971). The next observation allows us to reduce the space complexity to make the algorithm practical.

Observation 2: (Schroeppel and Shamir 1981)

The knapsack problem can be solved in $O(n2^{n/2})$ time and $O(2^{n/4})$ space.

Proof: We partition S into four lists, S_i, $1 \leq i \leq 4$. Each set is of size $n/4$. Let G_i, $1 \leq i \leq 4$ be the lists of all possible sublist sums in S_i respectively. Clearly, the partition problem has a solution iff $G_1 + G_2$ has a non-empty intersection with the list $(\{x\} - G_4) - G_3$.

This observation essentially reduces our problem to computing intersections of $A + B$ with $C + D$ (where A, B, C and D are lists of integers each of size $n/4$). To accomplish this we utilize a data structure that allows us to compute such intersections in $O(n2^{n/2})$ time without an increase in space. The main idea is to create an algorithm that generates the elements in $A + B$ and $C + D$ in increasing order, which allows us to compute the intersection by merging. Since we will use a very similar data structure to the one proposed in (Fiat *et al.* 1989) we review their implementation in the next section.

A Parallel Solution to Intersecting $A + B$ with $C + D$

We will first review how to generate elements in $A + B$ in ascending order sequentially. First, assume without loss of generality that A and B are given in ascending sorted order. During each phase of the algorithm, for each element a_k in A we keep a pointer to an element b_j such that all the sums of the form $a_k + b_i$ $(i < j)$ have been generated. We denote such pointers by $a_k \rightarrow b_j$. For example, part of our data structure may look similar to the figure below:

$$\begin{array}{lcl} a_1 & \rightarrow & b_{10} \\ a_2 & \rightarrow & b_7 \\ a_3 & \rightarrow & b_6 \\ a_4 & \rightarrow & b_4 \end{array}$$

Additionally, we will maintain a priority queue of all such sums. To generate $A + B$ in ascending order we repeatedly output the smallest $a_k + b_j$ in the priority queue, and insert the pointer $a_k \rightarrow b_{j+1}$ into the data structure and also insert $a_k + b_{j+1}$ into the priority queue. It is easy to see that, if A and B are of size $2^{n/4}$ we can generate all elements in $A+B$ in ascending order in $O(n2^{n/2})$ time. However, this algorithm is strictly sequential as it generates elements in $A + B$ one at a time.

Our algorithm is based on the idea that instead of generating the elements of $A + B$ one at a time we will in parallel generate $2^{n/4}$ elements at a time. To simplify our presentation, we describe the algorithm assuming that we have $2^{n/4}$ processors (which is of course unfeasible). By applying Brent's theorem (Brent 1974) we obtain speed-up results for any number of processors less than or equal to $2^{n/4}$. This is important since $2^{n/4}$ in practice will be far larger than the number of processors in the system.

The main idea of the algorithm is as follows. Each element a_k in A points to the smallest element of the form $a_k + b_j$ that has not been generated yet. Our algorithm works as follows. We first insert these $2^{n/4}$ elements in an array *TEMP* of size $2 * 2^{n/4}$ which will keep track of the elements that are candidates to be generated. The reader should note that we cannot just output these elements. We call those a_k such that $a_k + b_j$ is in *TEMP alive* (this notion will be clarified in step 5 below). We execute the following procedure:

1. *offset*=1.
2. Repeat 3–6 until *offset* equals $2 * 2^{n/4}$.
3. Each a_k that is alive and points to b_j $(a_k \rightarrow b_j)$ inserts all elements of the form $a_k + b_{j+m}$, where $m \leq$ *offset* in the array *TEMP*.
4. Find the $2^{n/4}$th smallest element in *TEMP* and delete all elements larger than it.
5. If the element of the form $a_k + b_{j+offset}$ remains in *TEMP* we will call a_k alive. Otherwise, a_i is called

dead, and will not participate in further computations.

6. Double the offset (i.e., $offset := 2*offset$).

Note that the number of elements in *TEMP* never exceeds $2 * 2^{n/4}$. This is true for the following reason. Assume that at phase t the number of live elements a_k is L. Each of these L elements contributes exactly *offset* pairs $a_i + b_j$ to *TEMP*. In the next phase, each such a_k will contribute $2 \times offset$ pairs, doubling its previous contribution. Thus, we will add $offset \times L$ new pairs to *TEMP*. Since $offset \times L \leq 2^{n/4}$, the number of pairs in *TEMP* never exceeds $2 \times 2^{n/4}$.

It is easy to see that the procedure above terminates in $O(\log(2^{n/4}))$ iterations since we are doubling the offset at each iteration. Therefore, the entire process of generating the next $2^{n/4}$ elements can be accomplished in $O(\log^2(2^{n/4})) = O(n^2)$ time using $O(2^{n/4})$ storage. We use the procedure above repeatedly to generate the entire set $A + B$ in ascending order.

Parallel Solution to KNAPSACK

Using our idea above we can obtain significant speed-ups in the implementation of each phase of the knapsack algorithm. We provide informal analysis of each phase indicating the speed-up that is possible in each phase.

1. We partition S into four lists, S_i, $1 \leq i \leq 4$. Each list is of size $n/4$.
2. We generate the lists G_i, $1 \leq i \leq 4$, i.e., the lists of all possible subset sums S_i, $1 \leq i \leq 4$. Each set is of size $2^{n/4} = M$. It is trivial to obtain M/P time complexity for this problem for any number of processors $P \leq M$. We sort G_1, G_2, G_3 and G_4. Parallel sorting is a well studied problem and is amenable to speedup.
3. We invoke our algorithm for computing the next M elements in the set $A+B$ as described in the previous section to generate G_1+G_2 and G_3+G_4 in ascending order. This phase take M/P time.
4. We intersect (by merging) the generated sets in M phases, generating and merging M elements at a time. Merging can be accomplished in $M/P + O(\log \log P)$ time by a known parallel merge algorithm. Since at least M elements are eliminated at each phase, and the number of possible sums is M^2, the total time to compute the desired intersection is $O(M^2/P)$ time.

The algorithm we sketched above provides a framework to achieve speed-up without sacrificing the good space complexity of the best known sequential algorithm. The novel aspect of the algorithm is step 3 where we suggest an original procedure. There are many details missing from the description of the algorithm above.

There have been a variety of approaches to parallelizing this problem (Chen & Jang 1992; Chen, Chern, & Jang 1990; Teng 1990; Lee, Shragowitz, & Sahni 1988; Lin & Storer 1991). Dynamic programming is efficient in certain cases. Karnin (Karnin 1984) proposes a parallel PRAM algorithm that takes $O(2^{n/2})$ time and uses space $O(2^{n/6})$ but it requires $O(2^{n/6})$ processors. Ferreira (Ferreira 1991) gives an $O(2^{n/4})$ time algorithm with $O(2^{n/4})$ processors but which uses $O(2^{n/2})$ space. Since the space requirements are quadratic in the time and in the number of processors, space becomes a bottleneck on most machines. An open question he posed, therefore, was to find an algorithm with $O(2^{n/4})$ space complexity(Ferreira 1991). This paper provides such an algorithm. Furthermore, our methodology can be applied to the taxonomy of problems presented in (Schroeppel & Shamir 1981) to parallelize of monotonically decomposable problems along a range of time-space tradeoffs, although we feel that the $S = O(2^{n/4})$, $T = O(2^{n/4})$, $P = O(2^{n/4})$ is the simplest and most useful.

Parallel Planning

It turns out that the algorithm sketched above has applications to a variety of problems which on the surface appear different from the knapsack problem. In particular, it is possible to adapt the approach to versions of planning problems. The idea was first outlined in a paper by Fiat, Moses, Shamir, Shimshoni and Tardos on planning in permutation groups (Fiat *et al.* 1989). We give a very informal description of their idea and then show how to use the parallel algorithm we discussed in the previous section, with some modifications, to implement the approach on massively parallel machines. A planning problem we consider may be stated as follows. Let S be a space of states. Typically, this space is exponential in size. Let G be a set of operators mapping elements of S into S. We denote composition of g_i and g_j by $g_i \circ g_j$. As before, by $G_i \circ G_j$ we denote the set of operators formed by composing operators in G_i and G_j respectively. We assume that each operator g in G has an inverse denoted by g^{-1}.

The planning problem is to determine the shortest sequence of operators of the form $g_1, g_2, \ldots$ that maps some initial state X_0 to a final state Y. Without loss of generality assume the initial and final states are always some state 0 and E respectively. In this paper we make the planning problem slightly simpler by asking whether there exists a sequence of operators of length n that maps the initial state into the final state. Assume k is the size of G. Clearly, the problem can be solved by a brute force algorithm of time complexity $O(k^n)$. We can also solve it with $O(n)$ space by iterative-deepening depth-first search. We are interested in problems for which k^n time is prohibitive but $\sqrt{k^n}$ as the total number of computations is feasible.

Let us spell out some assumptions that make the approach work. We assume that all the operators have inverses. We refer to G^{-1} as the set of all inverses

of operators in G. We also assume that it is possible to induce a total order $<$ on the states of S. For example, in the example above the states are sums of subsets ordered by the $<$ relation on the set of integers. In the case of permutation groups considered in (Fiat *et al.* 1989), permutations can be ordered lexicographically. There are several additional mathematical assumptions that must be satisfied, some of which are outlined in the full version of our paper.

Let $G_1 = G_2 = G_3 = G_4 = G \circ G \circ \cdots \circ G$, (G composed with itself $l/4$ times). To determine whether there exists a sequence $g_1, , ..., g_l$ that maps 0 into E (i.e., $g_1 ... g_l(0) = E$), we instead ask the question whether E is contained in $G_1 \circ G_2 \circ G_3 \circ G_4(0)$.

But since the operators have inverses we can ask the question of whether $G_2^{-1} \circ G_1^{-1}(E)$ has a non-empty intersection with $G_3 \circ G_4(0)$.

However, this naturally suggests the very similar scenario that we considered in the discussion on the knapsack problem. If the sizes of G_1, G_2, G_3 and G_4 are M, we can solve this intersection problem in time $O(M^2/P)$ and space $O(M)$. This assumes that we can generate the states in $A \circ B$ (where $\circ$ is now a composition on operators) in increasing order.

To illustrate this approach let us consider a very simple problem. Let $f_1(x) = x+5$ and $f_2(x) = 3*x$ be two functions (operators). Given integers c_1 and c_2 we want to find whether there is a sequence of applications of either f_1 or f_2 to c_1 that yield c_2. E.g., (10+5)*3)+5 = 50, that is 50 is reachable from 10 by $f_1(f_2(f_1(10) =$ 50. To find out whether c_2 is reachable from c_1 in 40 applications of our operators we first create a set F of all possible functions of the form $f(x) = a*x+b$ that can be created by composing f_1 and f_2 with each other ten times. This set is of size 2^{10}.

Now consider the two inverses of f_1 and f_2, namely $h_1(y) = y-5$ and $h_2(y) = y/3$. We create a set H of all possible functions of the form $h(y) = y/c - d$ that can be generated by composing h_1 and h_2 ten times. This set is also of size 2^{10}. Now we need to find out whether $F \circ F(c_1)$ has a non empty intersection with $H \circ H(c_2)$. The reader can verify that we can produce a monotone order on the operators in F and H that allows us to apply the approach we sketched above. Therefore, we can solve this problem sequentially in time proportional to roughly $20 * 2^{20}$ with 2^{10} storage. The brute force bidirectional search requires 2^{20} storage. Using the algorithms we developed in this paper we can parallelize this approach without increase in storage.

As another application of this approach, we can obtain a parallel algorithm for problems such as Rubik's Cube and other similar problems. The implementation of the algorithm sketched above becomes somewhat more involved because composition of operators is not necessarily monotonic, as defined later. We, nevertheless, can modify the algorithms sketched above to obtain efficient parallel implementations with reduced space.

To describe the algorithm, we assume that our operators are members of a permutation group Π. The *degree* of Π is defined to be the number of points on which Π acts. We set this to be q. We can suppose without loss of generality that each permutation P is a bijection from the set of integers $[1, 2, ..., q]$ to itself. The notation for P will be the vector $\langle P(1), P(2), \ldots, P(q) \rangle$. For example, $\langle 4, 3, 2, 1 \rangle$ is the reversal permutation of degree 4 that sends 1 into 4, 2 into 3, 3 into 2, and 4 into 1. If P and Q are permutations then we define their product $P \circ Q$ by $(P \circ Q)(i) = Q(P(i))$. This order of multiplying permutations is customary in much contemporary group-theoretic literature, so we will retain it. Let G be a subset (not a necessarily a subgroup) of k permutations in Π. The elements of G are our basic operators. Given an integer n and a group element γ we ask if γ can be written as the product of n elements from G. We have seen that this problem is reducible to the following: Given 4 sorted lists of permutations A, B, C, D, each of size $O(k^{n/4})$, does $A \circ B$ intersect $C \circ D$? This problem in turn reduces to the problem of generating the elements of $A \circ B$ in sorted order. We order permutations lexicographically. (Remark: the number of elements in A, B, C, D may be smaller, in certain cases, than $O(k^{n/4})$ because the same group element may be written in many different ways as words in G of length $k/4$.) We apply our earlier parallel algorithm for the knapsack problem to generate the elements of $A \circ B$ in batches of $k^{n/4}$ at a time.

A problem arises becauses our algorithm requires, given an $a_i \in A$ and a $b_j \in B$, the generation of the next *offset* elements of the form $a_i \circ b, b \in B$. When $\circ$ was the + operator, we could simply let these elements be $a_i + b_{j+1}, a_i + b_{j+2}, \ldots, a_i + b_{j+offset}$. This works because sum is a monotonic operator: $b < b' \iff a + b < a + b'$. This monotonicity fails in the case of permutations, however. For example, $\langle 1, 3, 2, 4 \rangle < \langle 1, 3, 4, 2 \rangle$, but

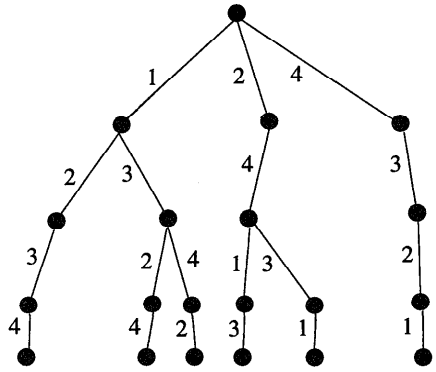
$$\begin{aligned} \langle 4, 3, 2, 1 \rangle \circ \langle 1, 3, 2, 4 \rangle &= \langle 4, 2, 3, 1 \rangle \\ &> \langle 2, 4, 3, 1 \rangle \\ &= \langle 4, 3, 2, 1 \rangle \circ \langle 1, 3, 4, 2 \rangle \end{aligned}$$

Thus, each a will induce a new ordering of B, and B will have to be traversed in this ordering, in parallel, for all a.

As in the sequential case, a list of permutations will be stored in a trie, a depth q tree whose leaves correspond to permutations in the list and whose unused branches are pruned. Each edge is labeled and the labels encountered when traversing the tree from root to leaf are the images of $[1, 2, \ldots, q]$ of the permutation that is represented by that leaf (see Figure 1).

If the edges emanating from any node in the tree are arranged in increasing order, then the leaves of the tree will be in lexicographically increasing order from

Figure 1: Tree representing the list of 6 permutations $\langle 1,2,3,4\rangle$, $\langle 1,3,2,4\rangle$, $\langle 1,3,4,2\rangle$, $\langle 2,4,1,3\rangle$, $\langle 2,4,3,1\rangle$, $\langle 4,3,2,1\rangle$. Each leaf represents a permutation and the leaves are in lexicographically increasing order from left to right. The leftmost leaf is the identity permutation $\langle 1,2,3,4\rangle$ and the rightmost leaf is the reversal permutation $\langle 4,3,2,1\rangle$.

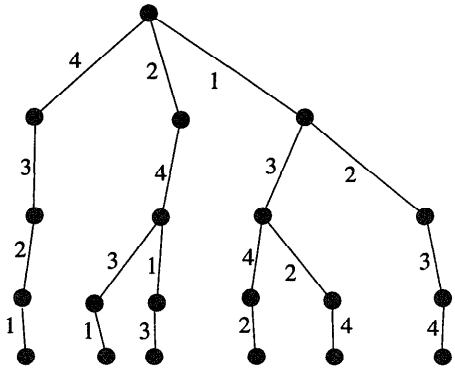


left to right. Given a permutation a, we can rearrange the order of edges emanating from each node to reflect their order in a. The leaves of the new tree, reading from left to right, are increasing with respect to the order induced by a. If the leaf b is to the left of the leaf b' in the new order, then $a \circ b < a \circ b'$ (see Figure 2).

In order for the earlier algorithm to go through, it is necessary, given a_i and b_j to find the b such that there are precisely m leaves between b_j and b in the order induced by a_i. This is done by initially storing the number of leaves in the rooted tree at each node of B with that node. In addition to the pointer from a_i to b_j, a_i also stores the index of b_j in the ordering it introduces. A top down search starting at the root can then find b by performing a prefix sum on the number of leaves in the subtrees of the children of each node in the a_i ordering. This requires only time linear in q, and from this information the child of the node in whose subtree b resides can be inferred. This requires $O(q^2)$ work, as in the sequential algorithm. Note that the step at each node can be performed in parallel with only q extra processors because in the algorithm we use we only need a subinterval of B so that nodes to the left of b are all generated. However, because q is usually fairly small it would almost certainly be a waste of time in practice to perform the prefix sum of the q values in parallel due to communication and synchronization overheads.

This method gives a complexity of $T = O(V/P)$, $S = O(\sqrt{V})$, for $P \leq \sqrt{V}$ up to logarithmic factors and multiplicative factors in q for finding a length n shortest word on k generators, where $V = k^{n/2}$, and

Figure 2: Reordered tree of Figure 1. Each interior node has been reordered according to the permutation $\langle 4,3,2,1\rangle$. If leaf b is to the left of leaf b' then the permutation $\langle 4,3,2,1\rangle \circ b$ is lexicographically less than the permutation $\langle 4,3,2,1\rangle \circ b'$.



the time complexity of the best sequential algorithm is O(V). This parallelizes the method of Fiat et al. efficiently with respect to work on a CREW PRAM when $P \leq sqrtV$ without asymptotic space utilization greater than the sequential method. Other time space tradeoffs, though possible, are not as useful. Although this technically only applies to a group of operators it is easy to modify the algorithm to apply to a groupoid of operators such as arises in the 15-puzzle.

The CREW PRAM model is unrealistic for current parallel architectures and was used here only for simplicity of exposition. A direct implementation of the algorithm we described would be slow on real machines because of its extensive reliance on complex communication and synchronization patterns. We have developed a slightly more complex algorithm with the same asymptotic time/space complexity (up to logarithmic factors) which is better suited to implementation because of its greater reliance on local memory accesses and should also perform better when $P \ll \sqrt{V}$. In general there are many implementation-dependent parameters that can affect the performance of parallel permutation group manipulation algorithms such as whether to perform permutation composition itself locally (York 1991). Although fundamental permutation group operations are theoretically parallelizable, their practical implementation remains challenging (Babai, Luks, & Seress 1987; Cai 1992; Stiller 1991).

Discussion

We presented an approach for massively parallel state-space search. This approach is a parallel implementation of the sequential algorithm proposed in (Fiat *et al.* 1989) for finding shortest word representations in permutation groups. Our implementation relies on a

new idea for computing the k-smallest elements in the set $A + B$. This idea is used to parallelize a key part of the algorithm.

As future work we would like to apply compression techniques such as binary decision diagrams to this state space (Clarke, Filkorn, & Jha 1993; Burch, Clarke, & McMillan 1992).

References

Babai, L.; Luks, E.; and Seress, A. 1987. Permutation groups in NC. In *STOC 87*, volume 19, 409–420.

Brent, R. P. 1974. The parallel evaluation of general arithmetic expressions. *Journal of the ACM* 21(2):201–206.

Burch, J. R.; Clarke, E. M.; and McMillan, K. L. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2):142–170.

Cai, J.-Y. 1992. Parallel computation over hyperbolic groups. In *24'th Annual STOC*, 106–115. ACM.

Chen, G.-H., and Jang, J.-H. 1992. An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing* 18(7):811–821.

Chen, G.-H.; Chern, M.-S.; and Jang, J.-H. 1990. Pipeline architectures for dynamic programming algorithms. *Parallel Computing* 13(1):111–117.

Clarke, E. M.; Filkorn, T.; and Jha, S. 1993. Exploiting symmetry in temporal logical model checking. In *Proceedings of the Fifth Workshop on Computer-Aided Verification*, 450–462.

Diffie, W., and Hellman, M. 1976. New directions in cryptography. *IEEE Trans. Information Theory* IT-22:644–654.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1990. PRA*: A memory-limited heuristic search procedure for the Connection Machine. In *Third Symposium on the Fontiers of Massively Parallel Computations*, 145–149.

Ferreira, A. G. 1991. A parallel time/hardware tradeoff $T \cdot H = O(2^{n/2})$ for the knapsack problem. *IEEE Transactions on Computers* 40(2):221–225.

Fiat, A.; Moses, S.; Shamir, A.; Shimshoni, I.; and Tardos, G. 1989. Planning and learning in permutation groups. In *30th Annual Symposium on Foundations of Computer Science*, 274–279. IEEE Computer Society Press. Los Alamitos, CA, USA.

Hillis, D. 1985. *The Connection Machine*. MIT Press.

Horowitz, E., and Sahni, S. 1974. Computing partitions with applications to the knapsack problem. *Journal of the Association for Computing Machinery* 21(2):277–292.

Karnin, E. D. 1984. A parallel algorithm for the knapsack problem. *IEEE Trans. Comput.* C-33:404–408.

Korf, R. E. 1985a. Depth-first iterative-deepening: an optimal admissible tree search. *AI* 27:97–109.

Korf, R. E. 1985b. Iterative-deepening-A*: an optimal admissible tree search. In *International Joint Conference on Artificial Intelligence.*

Lee, J.; Shragowitz, E.; and Sahni, S. 1988. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing* 5(4).

Lin, J., and Storer, J. A. 1991. Processor-efficient hypercube algorithms for the knapsack problem. *Journal of Parallel and Distributed Computing* 13(3):332–337.

Merkle, R., and Hellman, M. 1978. Hiding information and receipts in trap door knapsacks. *IEEE Transactions on Information Theory* IT-24:525–530.

Pohl, I. 1971. Bi-directional search. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 6.* Edinburgh: Edinburgh University Press. 127–140.

Powley, C., and Korf, R. E. 1988. SIMD and MIMD parallel search. In *Proceedings of the AAAI Symposium on Planning and Search.*

Powley, C., and Korf, R. E. 1991. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13(5):466–477.

Powley, C.; Ferguson, C.; and Korf, R. E. 1991. Parallel tree search on a SIMD machine. In *Third IEEE Symposium on Parallel and Distributed Processing.*

Rao, V. N., and Kumar, V. 1987. Parallel depth-first search, part i. Implementation. *International Journal of Parallel Programming* 16(6):479–499.

Schroeppel, R., and Shamir, A. 1981. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing* 10(3):456–464.

Stiller, L. 1991. Group graphs and computational symmetry on massively parallel architecture. *Journal of Supercomputing* 5(2/3):99–117.

Stiller, L. 1992. An algebraic paradigm for the design of efficient parallel programs. Technical Report JHU-92/26, Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218.

Tarjan, R., and Trojanowski, A. 1977. Finding a maximum independent set. *SIAM Journal on Computing* 6:537–546.

Teng, S.-H. 1990. Adaptive parallel algorithms for integral knapsack problems. *J. Parallel and Distributed Computing* 8(4):400–406.

Thinking Machines Corporation. 1992. *Connection Machine CM-5 Technical Summary.* 245 First St., Cambridge, MA 02142–1264: Thinking Machines Corporation.

York, B. W. 1991. Implications of parallel architectures for permutation group computation. In Finkelstein, L., and Kantor, W., eds., *Proceedings of the Workshop on Groups and Computation*, 293–313. Rutgers, NJ: DIMACS.