Exploiting Algebraic Structure in Parallel State-Space Search (Extended Abstract)

Jon Bright, Simon Kasif and Lewis Stiller
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218

16 October 1992

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 algorithms have applications to many important problems in Artificial Intelligence and Combinatorial Optimization such as planning problems and scheduling problems.

1 Introduction

Many key computational problems in Artificial Intelligence (AI) such as planning, game playing and scheduling can be solved by performing very large state-space searches. Our main interest in this paper is in the problem of *planning*, a well-studied problem in the AI literature. The best known algorithm for finding optimal solutions for general planning problems on sequential computers is IDA*, developed by Rich Korf [Kor85a, Kor85b]. IDA* has an exponential worst-case time complexity and its efficiency heavily depends on the ability to synthesize good heuristics. There are many attempts to parallelize state space search [EHMN90, PK91, PFK91, PK89, RK87]. In this paper we examine problems for which the computational cost of exploring the entire space of possible states may be prohibitive and derivation of good heuristics is difficult. For these problems we must reduce the worst case complexity by a significant amount. Surprisingly, due to the special algebraic structure of the problems we are considering, we can substantially reduce the time and space complexity of the algorithm. For example we will consider problems that have $O(k^N)$ states (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 $O(k^{N/4})$. Whereas the time-space complexity remains exponential, these algorithms are capable of solving problems that were not tractable for conventional architectures. For example, if we consider a problem that requires 10^{25} time using a brute force algorithm we can potentially solve it in 10^9 time and 10^6 space with 1000 processors. This class of

problems seem to be particularly well matched with massively parallel machines such as the CM-2 or CM-5 [Thi92, Thi91, Hil85]. Our technique offers several exciting new possibilities for solving many problems, ranging from toy problems such as Rubik's Cube to scheduling problems for Hubble Space Telescope.

Our approach is strongly influenced by the elegant sequential algorithm proposed by Shamir and his group, which we will review in the next section [SS81, FMS+89]. In fact, we view our work as a parallel implementation of this algorithm.

In this paper we describe parallel algorithms using the 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 convolution. Ultimately, we plan to achieve efficient implementation of our algorithms by exploiting the techniques to map algorithms for parallel architectures that are being developed by Stiller [Sti92a, Sti92b, Sti91a]. This approach has already resulted in several important applications to chess endgame analysis and several scientific problems [Sti91b, Sti91c].

2 Review of the Shamir et al algorithm

In this section, we consider the example of SUBSET SUM to illustrate this approach. A similar approach can be used to solve 0-1 knapsack problems. The input to the algorithm is a set of integers S and a number x. The output is a subset of S, S', such that the sum of elements in S' is equal to x. This problem is fundamental in scheduling, bin packing and other combinatorial optimization problems. For example, we can use solutions to this problem to minimize total completion time of tasks of fixed duration executing on two processors. This problem is sometimes referred to as the partition problem, and is known to be NP-complete. It is also known that in practice it can be solved in pseudo-polynomial time by dynamic programming. 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 $a \epsilon A$ and $b \epsilon B$. Define $A - B$ to be the set of integers c such that $c = a - b$ where $a \epsilon A$ and $b \epsilon B$.

Observation 1:

The SUBSET SUM problem can be solved in $O(N2^{N/2})$ time and $O(2^{N/2})$ space.

Proof. We partition the S into disjoint sets S_1 and S_2 such that the size of S_1 is $N/2$. Let G_1 and G_2 be the sets of all sums of subsets of elements in S_1 and S_2 respectively. Clearly, our problem has a solution if and only if the set G_1 has a non-empty intersection with $\{x\} - G_2$. However, note that we can sort both sets and find the intersection by merging. Thus, the time complexity of this algorithm can be seen to be $N2^{N/2}$ using any optimal sorting algorithm and noting the trivial identity $2\log(2^{N/2}) = N$. Unfortunately, the space complexity is also $2^{N/2}$ which makes it prohibitive for solution on currently available machines for many interesting problems. This approach was first suggested by Horowitz and Sahni for 0-1 knapsack problems [HS86].

In the AI literature this algorithm has a strong similarity to bidirectional search studied by Pohl [Poh71]. The next observation allows us to reduce the space complexity to make the algorithm practical [SS81].

Observation 2:

The SUBSET SUM problem can be solved in $O(N2^{N/2})$ time and $2^{N/4}$ space.

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

This observation essentially reduces our problem to computing intersections of $A + B$ with $C + D$ (where A, B, C and D are sets of integers each of size $N/4$). To accomplish this we utilize a data structure that allows us to compute such intersections without an increase in space in $O(N2^{N/2})$ time. The main idea is to create an algorithm that generates elements in $A + B$ and $C + D$ in increasing order, which allows us to compute the intersection by merging. However, their algorithm is strictly sequential as it generates elements in $A + B$ one at a time. We will review their data structure in a latter section.

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

Since we will use a very similar data structure to the one proposed in [FMS+89] it is worthwhile to review their implementation. We will first show how to generate elements in $A + B$ in ascending order. 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 as $a_k \rightarrow b_j$ For example, part of our data structure may look similar to the figure below:

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

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 time $cN2^{N/2}$, where c is a small constant.

Our algorithm is based on the idea that instead of generating a single element of $A + B$ one at a time we will in parallel generate the next K elements $(K \leq N)$. We describe the algorithm assuming that we have $P = K$ processors. By the standard use of Brent's theorem [Bre74] we obtain speed-up results for any number of processors less than or equal to K. This is important since K in practice might be far larger than the number of processors in the system.

We now describe the algorithm to find the next K elements. 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 the smallest K of these elements in an array *TEMP* of size $2K$ which will keep track of the elements that are candidates to be generated. The reader should note that we cannot just output these K 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 $2K$.
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 Kth 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 $2K$. 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* $\times L \leq K$, the number of pairs in *TEMP* never exceeds $2 \times K$.

It is easy to see that the procedure above terminates in $O(\log M)$ steps where M is the size of the set B. Therefore, the entire process can be accomplished in $O(\log^2 M)$ steps with M processors using at most $M = 2^{N/4}$ storage. We use the procedure above repeatedly to generate the entire set $A + B$ in ascending order.

3.1 Parallel Solution to SUBSET SUM

Using our idea above we can obtain significant speed-ups in the implementation of each phase of the SUBSET SUM algorithm. We describe a crude version of our algorithm with P processors assuming P is constant. We provide informal analysis of each phase suggesting that significant speed-ups are possible in each phase.

1. We partition S into four sets, S_i, $1 \leq i \leq 4$. Each set is of size $N/4$.
2. We generate the sets G_i, $1 \leq i \leq 4$, i.e., the sets 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 for maximal speed-ups.

3. We invoke our algorithm for computing the next K, $(K = M)$ elements in the set $A+B$ as described in the previous section to generate $G_1 \circ G_2$ and $G_3 \circ G_4$ 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 substantial speed-up without sacrificing the good space complexity of the best known sequential algorithm. The only non-trivial 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. Since we are in the process of designing an efficient implementation on existing architectures such as the CM-5, the final details will vary from the algorithm sketched above. The main purpose of this note is to illustrate the potential for parallelism. The key to the success of the final implementation is our ability to map the algorithm above to available computational primitives such as parallel prefix and convolution.

4 Parallel Planning

It turns out that the algorithm sketched above has applications to a variety of problems on surface different from the SUBSET SUM problem one of which is the planning problem in AI. The approach was first outlined in a paper by Shamir and his group for planning in permutation groups [FMS+89]. We give a very informal description of their idea. A planning problem 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 $g_i g_j$. 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}.

A 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 F respectively. In this paper we make the planning problem slightly simpler by asking whether there exists a sequence of operators of length l 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 $O(k^l)$. We can also solve it with $O(l)$ space by iteratively-deepening depth-first search. We are interested in problems for which k^l is prohibitive but $\sqrt{k^l}$ as 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. E.g, in the example above the states are sums of subsets ordered by the $<$ relation on the set of integers. In the case of permutations groups considered in Shamir et al, permutations can

be ordered lexicographically. There are several additional mathematical assumptions that must be satisfied, some of which are outlined in Shamir et al. and 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 F (i.e., $g_1, , ..., g_l(0) = F$), we instead ask the question whether F 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 $G_2^{-1} \circ G_1^{-1}(F)$ has a non-empty intersection with $G_3 \circ G_4(F)$.

However, this naturally suggests the very similar scenario that we considered in the discussion on the SUBSET SUM 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. The implementation of the algorithm sketched above becomes somewhat more involved because composition of operators is not necessarily monotonic in the order. We, nevertheless, can modify the algorithms sketched for SUBSET SUM to obtain efficient parallel implementations without losing efficiency. This implementation is discussed in the full version of our paper. As an application, we can obtain a parallel algorithm for problems such as Rubik's Cube and other problems.

5 Discussion

We presented an approach for massively parallel state-space search. This approach is a parallel implementation of the sequential algorithm proposed in [FMS+89] 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 mentioned before, we are in the process of designing a practical implementation of our approach on existing parallel machines. There are several non-trivial aspects of this implementation. Our main goal is solving extremely large instances of hard problems that are not tractable at all on existing sequential machines. Since the size of the state space generated is extremely large, we must be careful about the constants that influence dramatically the actual size of the problem this approach can attack. We believe that our practical implementation may have significant practical implications since it should indicate which parallel primitives are fundamental in the solutions of large state space search problems. As a long range goal we plan to develop automated mapping techniques that take advantage of the special algebraic structure of problems in order to construct an efficient mapping onto parallel machines.

Acknowledgements This research has been supported partially by U.S. Army Research Office Grant DAAL03-92-G-0345 and NSF/DARPA Grant CCR-8908092.

References

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[EHMN90] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: A memory-limited heuristic search procedure for the Connection Machine. In *Third Symposium on the Frontiers of Massively Parallel Computations*, pages 145–149, 1990.

[FMS+89] A. Fiat, S. Moses, A. Shamir, I. Shimshoni, and G. Tardos. Planning and learning in permutation groups. In *30th Annual Symposium on Foundations of Computer Science*, pages 274–279, Los Alamitos, CA, 1989. IEEE Computer Society Press.

[Hil85] D. Hillis. *The Connection Machine.* MIT Press, 1985.

[HS86] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 23:317–327, 1986.

[Kor85a] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kor85b] R. E. Korf. Iterative-deepening-A*: an optimal admissible tree search. In *International Joint Conference on Artificial Intelligence*, 1985.

[PFK91] C. Powley, C. Ferguson, and R. E. Korf. Parallel tree search on a SIMD machine. In *Third IEEE Symposium on parallel and distributed processing*, Dallas, Texas, December 1991.

[PK89] C. Powley and R. E. Korf. SIMD and MIMD parallel search. In *Proceedings of the AAAI Symposium on planning and search*, Stanford, California, March 1989.

[PK91] C. Powley and R. E. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, May 1991.

[Poh71] I. Pohl. Bi-directional search. *Machine Intelligence*, 16:127–140, 1971.

[RK87] V. Nageshwara Rao and V. Kumar. Parallel depth-first search, part i. Implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[SS81] R. Schroeppel and A. Shamir. 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, August 1981.

[Sti91a] L. Stiller. Group graphs and computational symmetry on massively parallel architecture. *Journal of Supercomputing*, 5(2/3):99–117, November 1991.

[Sti91b] L. Stiller. Karpov and kasparov: the end is perfection. *International Computer Chess Association Journal*, 14(4):198–201, December 1991.

[Sti91c] L. Stiller. Some results from a massively parallel retrograde analysis. *International Computer Chess Association Journal*, 14(3):129–134, September 1991.

[Sti92a] L. Stiller. How to write fast and clear parallel programs using algebra. Technical Report LA-UR-92-2924, Los Alamos National Laboratories, 1992.

[Sti92b] L. Stiller. White to play and win in 223 moves: a massively parallel exhaustive state space search. *Journal of the Advanced Computing Laboratory*, 1(4):1, 14–19, July 1992.

[Thi91] Thinking Machines Corporation. *Connection Machine CM-200 Series Technical Summary.* Thinking Machines Corporation, 245 First St., Cambridge, MA 02142–1264, June 1991.

[Thi92] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, 245 First St., Cambridge, MA 02142–1264, January 1992.