Parallelization of Tree-Recursive Algorithms on a SIMD Machine *

Curt Powley, Richard E. Korf, and Chris Ferguson
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

ABSTRACT

The set of tree-recursive algorithms is large, including constraint satisfaction using backtracking, iterative-deepening search such as IDA*, depth-first branch-and-bound, two-player game minimax search, and many divide-and-conquer algorithms. We describe a structured method for implementing such algorithms on SIMD machines, and identify measures for determining if a tree-recursive application is amenable or ill-suited for SIMD parallelization. Using these ideas, we evaluate results from four testbeds.

1 Introduction

1.1 Tree-Recursive Algorithms

A tree-recursive algorithm is one that traverses a tree in executing multiple recursive calls. A simple example is this procedure for calculating Fibonacci (n), where n is a non-negative integer.

Fib (n)

if ($n \leq 1$) return (1)

else return (Fib ($n-1$) + Fib ($n-2$))

*This work is supported in part by W. M. Keck Foundation grant W880615, NSF grants DIR-9024251 and IRI-9119825, NSF Biological Facilities grant BBS-8714206, the Defense Advanced Research Projects Agency under Contract MDA 903-87-C0663, Rockwell International, the Advanced Computing Facility of the Mathematics and Computer Science Division of Argonne National Laboratory, the University of Maryland Institute for Advanced Computer Studies, the University of Minnesota/Army High Performance Computing Research Center, the Massively Parallel Computer Research Laboratory of Sandia National Laboratories, and by Thinking Machines Corporation.

The set of tree-recursive algorithms includes constraint satisfaction using backtracking, iterative-deepening search such as IDA*, depth-first branch-and-bound, two-player game minimax search, and most divide-and-conquer algorithms.

1.2 SIMD Machines

Two examples of single-instruction, multiple-data (SIMD) machines are the MasPar[1] and the Connection Machine (CM)[2]. The MasPar is composed of up to 16,384 (16K) processors. The Connection Machine, the computer used in this study, is composed of up to 65,536 (64K) one-bit processors.

Every processor on a SIMD machine must execute the same instruction at the same time, or no instruction at all. This can make programming and verification of straightforward tasks easier, but can considerably complicate the programming of complex tasks, such as tree-traversal. Because of this programming constraint, SIMD machines have largely been used for "data parallel" applications. Traversing a large, irregular tree that is dynamically generated does not fit this computational paradigm. On the other hand, tree traversal consists of performing the same computation of node expansion and evaluation repeatedly. This suggests that fine-grain SIMD parallel computers might be appropriate for implementation of tree-recursive algorithms.

Some of this research was previously reported in [14], [15], [16], [17], and [18].

2 SIMD Tree Search (STS)

Our basic SIMD Tree Search algorithm, or *STS*, consists of an initial distribution of nodes to processors, followed by alternating phases of depth-first tree-traversal (search) and load balancing [15].

[1]MasPar is a trademark of MasPar Computer Corporation.

[2]Connection Machine is a trademark of Thinking Machines Corporation.

Initially, the tree consists of just the root node, located on a single active processor of the machine, and all the remaining processors are inactive. This processor expands the root, generating its children, and assigns each child to a different processor. Each of these processors, in parallel, expands its node, generating more children that are assigned to different processors. This process continues until there are no more free processors available.

If the processor assigned to a node remains associated with that node after it is expanded, that processor will be wasted by not having any work to do during the subsequent depth-first search. To avoid this, when a node is expanded, its first child is assigned to the same processor that held the parent, and only additional children are assigned to free processors. This guarantees that once the initial distribution is completed, all processors will be assigned to nodes on the frontier of the tree, and can participate in the depth-first search.

Once the initial distribution is completed, each processor conducts a depth-first search of its assigned frontier node. Processors use a stack to represent the current path in the tree. Unfortunately, the trees generated by almost all tree-recursive problems of interest have irregular branching factors and depths, and some processors will finish searching their subtrees long before others. Thus, load balancing is necessary to effectively use a large number of processors.

On a MIMD machine, these idle processors can get work from busy processors without interrupting the other busy processors [19]. On a SIMD machine, however, since every processor must execute the same instruction at the same time, or no instruction at all, in order to share work, all search activity must be temporarily suspended. Thus, when the number of active processors becomes small enough to make it worthwhile, search stops and load balancing begins.

Two important problems are determining when to trigger load balancing, and how to distribute the work. STS uses a dynamic trigger [14] that automatically adjusts itself to the problem size, and to different stages within a single problem. It is a greedy approach that maximizes the average rate of work over a search/load-balance cycle. After triggering occurs, the work is redistributed during load balancing as follows. Each busy processor scans its stack to find a node with unexplored children [19]. When there are more active processors than idle processors, a subset of the active processors must be selected to export work. Criteria for choosing exporting processors include: (1) largest estimated load [14], (2) random selection [7], (3) selection of least-recently used [8], and (4) location of the processor's node in the tree so that the tree is searched in more of a left-to-right fashion [20]. The effectiveness of the work-distribution method depends on the application.

3 Measures of Performance

Our primary measures of performance are *speedup* and *efficiency*. Speedup is the time that would be required by the most efficient serial program for the application, running on one processor of the SIMD machine, divided by the time required by STS. Efficiency is simply speedup divided by the number of processors. Since we are interested in the factors that contribute to overall efficiency, we decompose efficiency into four components: raw speed ratio R, fraction of time working F, utilization U, and work ratio N. The product of these four factors equals efficiency.

The raw speed ratio is the ratio of the node generation rate of a single busy processor in the parallel STS algorithm, compared to the serial algorithm, and reflects the overhead of SIMD machines in executing conditional code. Next is the fraction of total time that is devoted to working (searching), as opposed to load balancing. A third factor is the processor utilization, which is the average fraction of processors that are busy during search phases. Utilization reflects the extent to which processors finish their work and are idle while waiting to receive work in the next load balancing phase. The final factor is the ratio of total nodes generated by the serial algorithm to total nodes generated by the parallel algorithm, or the work ratio. Depending on the application, this may or may not be significant.

4 Constraint Satisfaction

Our simplest testbed for STS is backtracking for constraint-satisfaction problems. For example, the N-Queens problem is to place N queens on an $N \times N$ chessboard so that no two are on the same row, column, or diagonal. In such applications, a path is cut off when a constraint is violated, or a solution found. The N-Queens problem can be solved by placing queens one at a time. When a constraint is violated, the search backtracks to the last untried legal move.

Using 16K processors on a CM2, we solved a 16-queens problem with a speedup of 10,669, for an efficiency of 65%. This corresponds to a raw speed ratio of .750, a fraction of time working of 0.870, a utilization of 0.998, and a work ratio of 1.0. The work ratio is one because we found all solutions, making the search tree identical in the serial and parallel cases. The total work done was 18.02 billion node generations in 5.2 hours, for a rate of 58 million node generations per minute. In contrast, the node generation rate on a Hewlett Packard 9000 model 350 workstation was 2.1 million per minute.

5 Iterative-Deepening

Another important algorithm in this class is iterative-deepening search, such as depth-first iterative-

deepening (DFID) and Iterative-Deepening A* (IDA*) [9]. DFID performs a series of depth-first searches. A path is cut off when the depth of the path exceeds a depth threshold for the iteration. Thresholds for succeeding iterations are increased so that when a solution is found it is guaranteed to be of lowest cost. IDA* reduces the search space by adding to the path cost a heuristic estimate of the cost to reach a goal node.

Adapting STS to IDA* required the addition of iterations and a global threshold, and the maintenance of the path from the root to each node. Our testbed was IDA* on the Fifteen Puzzle [15]. Treating all 100 problems of [9] as one large problem, the speedup with 16K processors was 8389, for an efficiency of 63.7%. The overall efficiency is the product of a raw speed ratio of .830, fraction of time working of .857, processor utilization of .895, and work ratio of 1.0. The work ratio was set to one, because its fluctuation represents noise in this application [15]. The total work done by STS was 45.2 billion node generations in 5.7 hours, for an overall rate of 131 million node generations per minute.

6 Branch and Bound

Depth-first branch-and-bound (B&B) is a classic tree-recursive algorithm. In B&B, there is only one iteration, and the global threshold decreases as better solutions are found. Because of the changing threshold, the amount of work done in one part of the tree depends on results from searching other parts of the tree, and hence the work ratio is of interest.

We implemented the Euclidean traveling salesman problem (TSP), using depth-first branch-and-bound. A path in the tree represents a sequence of cities. Searching the tree corresponds to extending paths until the estimated cost of the path equals or exceeds the current global threshold. As with IDA*, the estimated cost of a path is equal to the cost of the path (partial tour) so far, plus a heuristic estimate of the cost to complete the tour. Our heuristic function calculates the minimum spanning tree of unexplored cities, and we order children by nearest neighbor. We ran 100 random 25-city problems, with edge costs varying from zero to 999. The speedup over all 100 problems with 16K processors was 4649. This corresponds to an overall efficiency of 28.4%, which is the product of a raw speed ratio of .452, fraction of time working of .840, processor utilization of .805, and work ratio of .928. The efficiency is significantly lower for this testbed primarily because of a lower raw speed ratio, caused by conditionals in the minimum spanning tree code.

7 Two-Player Games

A special case of branch-and-bound is alpha-beta minimax search for two-player games, such as chess. Iterative-deepening is also used in these algorithms, and in fact originated in this setting [21]. Alpha and Beta bounds produced by searching one subtree affect the amount of work necessary to search neighboring subtrees. These local bounds must be propagated throughout the tree, and a parallel algorithm will almost certainly do more work than a serial algorithm. Thus, the work ratio becomes especially important in these applications.

Extending STS to alpha-beta search required the addition of a function to periodically propagate Alpha and Beta bounds among different parts of the tree. We saved the best path from each iteration, and expanded it first on the following iteration. Node ordering by heuristic evaluation of children was also used.

An important problem in parallel game-tree search is how to minimize the extra work done in parallel (maximize the work ratio), without sacrificing utilization. Much work has been done on this problem in a MIMD context, and many of the ideas are usable for a SIMD approach. See, for example, [1, 3, 4, 12, 13]. Our approach was to search the tree in a left-to-right manner [20], except that the assignment of processors was delayed in areas of the tree where pruning was likely to occur [1].

For our testbed, we chose the two-player game of Othello. However, because of insufficient memory, we simulated a fast table-lookup evaluation function [11]. To estimate the work ratio associated with a good evaluation function, we did additional experiments using trees with randomly-assigned edge costs. For these trees, the heuristic value of a node was computed as the sum of the edge costs from the root to the node. The other three factors of efficiency were calculated from the Othello experiments, but iterative-deepening and node ordering were used only for the random trees. With 128 processors, efficiency was only 1.5%, and consisted of a raw speed ratio of .360, a fraction of time working of .850, a utilization of .400, and a work ratio of .123.

Though disappointing, these results are probably representative of two-player games for several reasons. For applications such as Othello, there is typically a large variation in the time of a serial node generation, lowering the raw speed ratio. Another reason is that SIMD machines have many processors, and in two-player games, the work ratio tends to decrease as the number of processors increases [13]. Felten and Otto reported speedups of 101 with 256 processors on a MIMD machine for chess [4], implying a work ratio of at least 0.39, which is greater than what we achieved with 128 processors. They are able to achieve a higher work ratio because they use a global table to save the best move for many nodes from the previous iteration, whereas we save only the nodes on the best path from the previous iteration.

8 Performance Summary

What makes a tree-recursive application suitable or inappropriate for a SIMD machine?

One consideration for any parallel machine is the parallelizability of the algorithm. In general, algorithms contain a serial component, and a parallelizable component. Amdahl's law states that no matter how many processors are used, speedup will never exceed the total problem size divided by the size of the serial component [2]. However, in many applications, as the number of processors grows, the problem size can also grow. Typically, the parallelizable component grows fast enough in relation to the serial component that speedup will not be severely limited [6]. However, if the parallelizable component grows too slowly in relation to the serial component, problem size will have to grow much faster than the number of processors, making high speedups difficult or impossible to achieve.

For example, mergesort is a classic divide-and-conquer tree-recursive algorithm for sorting a list of size n in $O(n\ logn)$ time. The unordered list is divided in half, each half recursively sorted, and then the halves merged back together. A straightforward STS implementation would be ineffective because the $O(n)$ serial component, which is dominated by the final merge, is too large relative to the $O(n\ logn)$ parallelizable component. Since STS requires a minimum of $O(n)$ elapsed time, and a serial algorithm requires $O(n\ logn)$ time, speedup is limited to $O(n\ logn)/O(n) = O(log\ n)$. For a large number of processors, memory would be exhausted long before the problem size would be large enough to allow a high speedup.

In order for STS to be effective for an application, *all* four of the efficiency factors must be relatively high. In our four testbeds, we have seen situations which can lower each of these factors, and make SIMD machines inappropriate.

The work ratio N can be particularly detrimental in two-player games. The raw speed ratio R is affected primarily by the variation in the time of a node generation. Thus, applications such as the Fifteen Puzzle which have simple, constant-time algorithms for a node generation, such as table lookup for the heuristic function, tend to have high raw speed ratios. On the other hand, applications such as Othello and the traveling salesman problem, which have node-generation or evaluation algorithms with a large variation in time, will tend to have low raw speed ratios.

The effectiveness of load balancing is measured by the product $F \cdot U$. Figure 1 shows, for the four testbeds, how $F \cdot U$ varies with $W\ /\ P\ logP$, where P is the number of processors, and W is the total work done, measured in node generations. The two-player curve includes data for varying numbers of processors, from

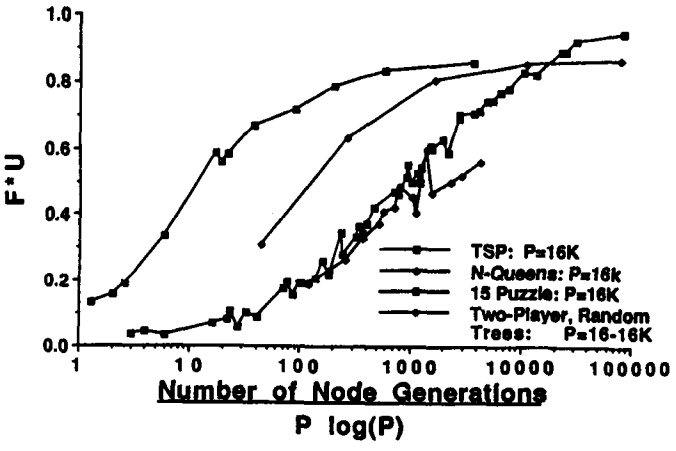

Figure 1: Load Balancing Effectiveness Across Domains

16 to 16K, while the other three curves are for 16K processors only.

The scalability of an algorithm is determined by its isoefficiency function [10], which is the rate at which total work must grow with respect to P, to maintain a constant efficiency. Our empirical results indicate a $P\ logP$ isoefficiency function in terms of $F \cdot U$, from 16 to 32,768 (32K) processors [15]. This is the reason the work is normalized by $P\ logP$ in Figure 1. In terms of the other factors, the raw speed ratio should remain constant with increasing number of processors[3], but the work ratio should decrease in the case of two-player games.

It is important to keep in mind that these curves only indicate load-balancing effectiveness, and do not portray the effect of the raw speed ratio, or the work ratio. For example, the traveling salesman problem has the best load-balancing performance through a wide range of problem sizes, but when the raw speed ratio is considered, it is outperformed by both the N-Queens and the Fifteen Puzzle applications.

In practice, if problems are too small, low efficiency is obtained since there is not enough work to share among the processors. This is seen in Figure 1 because low $F \cdot U$ is obtained in the leftmost portions of the curves. Thus, small problems are not suitable for the many processors associated with SIMD machines, and in these cases, it may be better to use a machine with fewer processors. In the case of two-player games such as chess or Othello, problems may be too small as a result of the time limit to make a move.

The critical factors causing the displacement between

[3] We assume here that the instruction cycle time of the machine will remain constant as larger machines are produced.

these four curves are the work distribution [15], the use of multiple iterations, and the relative cost of load balancing [19]. The relative cost of load balancing is the time of a load balance divided by the time of a node-generation.

9 Parallel Software Engineering

We structured our code to make an easy-to-use high-level environment for parallelization of tree-recursive algorithms. Our approach combines the following: (1) separation of domain-dependent and independent code, (2) minimization of the amount of domain-dependent parallel code, (3) a narrow, clean, and general interface between the domain-dependent and independent code, and (4) a method for extracting salient information from serial code. The idea is to confine the changes needed for a new application to those of a serial nature, while re-using the more complex code involving load balancing, communication, and other SIMD artifacts. Although the first three items are fundamental software-engineering tools, their application in this context produces something new because they abstract a class of algorithms, not just a class of domains.

A program "shell" is used for writing new domain-dependent code, and it specifies required functions, variables, and definitions. Once this shell is completed and linked to the rest of the STS program, load balancing and other parallel actions are automatic. Thus, the cost of developing and debugging the complex domain-independent code can be amortized over many applications. For example, the time to convert a serial N-Queens program to an STS version was about 8 hours. A related approach for parallelizing backtracking algorithms on MIMD machines was used in DIB [5].

10 Conclusions

This research extends the capability of SIMD machines, while at the same time demonstrating some of their limitations. STS is structured for relatively simple implementation of tree-recursive algorithms on a SIMD machine. Efficiencies of 64% were obtained for IDA* using the Fifteen Puzzle, 65% for backtracking for constraint satisfaction using the N-Queens problem, and 28% for Branch-and-Bound, using the traveling salesman problem. However, for domains that have a large variation in the time of a node generation, such as many two-player games, a SIMD approach does not seem appropriate. This highlights a fundamental limitation of SIMD machines, the idling of processors not executing the current instruction, which is measured by the raw speed ratio. Another intrinsic problem with two-player games on any parallel machine is that the work ratio decreases with increasing numbers of processors. In general, SIMD machines can be used effectively for tree-recursive applications in which the problems of interest are large enough, the time of node generations is relatively constant, and the parallelizable component is large relative to the serial component.

Acknowledgements

David Chickering wrote the serial version of the Othello code. Vipin Kumar provided access to a CM-200 with sufficient memory to run the STS Othello code.

References

[1] Selim G. Akl, David T. Barnard, and Ralph J. Doran. Design, analysis, and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4:192–203, March 1982.

[2] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computer capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, April 1967.

[3] R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger. Distributed game tree search. In Kumar, Gopalakrishnan, and Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 66–101. Springer-Verlag, Elmsford, New York, 1990.

[4] Edward W. Felten and Steve W. Otto. A highly parallel chess program. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988.

[5] Raphael Finkel and Udi Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[6] John L. Gustavson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[7] R.M. Karp and Y. Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the 20^{th} Annual ACM Symposium on Theory of Computing*, pages 290–300, Chicago, Il, May 1988.

[8] George Karypis and Vipin Kumar. Unstructured tree search on simd parallel computers. Technical Report 92-91, University of Minnesota, April 1992. Computer Science Department.

[9] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[10] V. Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[11] Kai-Fu Lee and Sanjoy Mahajan. The development of a world class othello program. *Artificial Intelligence*, 43:21–36, 1990.

[12] T. A. Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, July 1985.

[13] Curt Powley, Chris Ferguson, and Richard E. Korf. Parallel heuristic search: Two approaches. In Kumar, Gopalakrishnan, and Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 42–65. Springer-Verlag, 1990.

[14] Curt Powley, Chris Ferguson, and Richard E. Korf. Parallel tree search on a simd machine. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 249–256, Dallas, Texas, December 1991.

[15] Curt Powley, Chris Ferguson, and Richard E. Korf. Depth-first heuristic search on a simd machine. *Artificial Intelligence*, 1993. To appear.

[16] Curt Powley and Richard E. Korf. Simd and mimd parallel search. In *Proceedings of the AAAI Symposium on Planning and Search*, pages 49–53, Stanford, California, March 1989.

[17] Curt Powley and Richard E. Korf. Simd parallel search. In *Proceedings of the IJCAI-89 Workshop on Parallel Algorithms for Machine Intelligence*, Detroit, Michigan, August 1989.

[18] Curt Powley, Richard E. Korf, and Chris Ferguson. Simd ida*: Extended abstract. In *Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence*, pages 161–168, Sydney, Australia, August 1991. in conjunction with IJCAI91.

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

[20] Vikram A. Saletore and L. V. Kale. Consistent linear speedups to a first solution in parallel state-space search. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 227–233, August 1990.

[21] D.J. Slate and L.R. Atkin. Chess 4.5 - the northwestern university chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, New York, 1977.