PARALLEL HEURISTIC SEARCH
ON SHARED MEMORY MULTIPROCESSORS:
PRELIMINARY RESULTS

V. NAGESHWARA RAO, VIPIN KUMAR
AND K. RAMESH

TECHNICAL REPORT AI85-45
REVISED, JUNE 1987

This work is supported by the Army Research Office under grant no. ARO DAAG29-84-K-0060.
Parallel Heuristic Search on Shared Memory Multiprocessors: Preliminary Results

V. Nageshvara Rao, Vipin Kumar & K. Ramesh
Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin
Austin, Texas 78712

ABSTRACT

In this paper we discuss two different ways of parallelizing the A* state space search algorithm for shared memory multiprocessors. We implement these parallel algorithms to solve the 15-puzzle problem and the TSP problem on the Sequent Balance 21000 and the Butterfly parallel processor. Our preliminary results are very encouraging, as we are able to obtain linear speedups upto 100 processors. Since the best-first search paradigm of A* is a very commonly used, we expect these parallel versions to be effective for a variety of problems.

---

1 This work was supported by Army Research Office grant #DAAG29-84-K-0060 to the Artificial Intelligence Laboratory and by the ONR Parallel Processing equipment grant to the Computer Science Department, at the University of Texas at Austin.

2 Arpanet Address: kumar@sally.UTEXAS.EDU
Parallel Heuristic Search on Shared Memory Multiprocessors:
Preliminary Results

V. Nageshvara Rao, Vipin Kumar and K. Ramesh

Artificial Intelligence Laboratory
Computer Science Department
University of Texas at Austin

ABSTRACT

In this paper we discuss two different ways of parallelizing the A* state space search algorithm for shared memory multiprocessors. We implemented these parallel algorithms to solve the 15-puzzle problem and the TSP problem on the Sequent Balance 21000 and the Butterfly parallel processor. Our preliminary results are very encouraging, as we are able to obtain fairly linear speedups up to 100 processors. Since the best-first search paradigm of A* is a very commonly used, we expect these parallel versions to be effective for a variety of problems.

1. INTRODUCTION

Heuristic search is an important AI technique that is used to solve a variety of problems [NILSSON 80]. Search techniques are useful when one is able to specify the space of potential solutions, but the exact solution is not known beforehand. In such cases a solution can be found by searching the space of potential solutions. Clearly, if many processors are available, they can search different parts of the space concurrently.

For many problems, heuristic domain knowledge is available, which can be used to avoid searching some (unpromising) parts of the search space. This means that parallel processors following a simple strategy (such as divide the search space statically into disjoint parts and let each one be searched by a different processor) may end up doing a lot more work than a sequential processor. This would tend to reduce the speedup that can be obtained by parallel processing. If the amount of work done by a sequential processor is $W_s$
and the total amount of work done by n parallel processors is \( W_p \), then slowdown factor due to parallel-control-strategy is given by \( W_s/W_p \), and the upperbound on the speedup (assuming that the parallel architecture is ideal) is \( n.W_s/W_p \).

However, a given parallel architecture may not be able to deliver linear speedup due to memory contention, communication overheads, etc. Hence the resulting speedup could be even less than \( n.W_s/W_p \). This slow-down due to architecture is determined both by a given parallel control strategy and a given parallel architecture.

We have been investigating the use of parallel architectures for speeding up heuristic search [KUMAR 84][RAO 87]. In this paper, we will discuss two different ways of parallelizing the A* state-space search algorithm and present performance results for their implementations on Sequent Balance 21000 and the Butterfly parallel processor. Sequent Balance is a shared memory multiprocessor system with up to 30 processors accessing shared memory through a common bus. Butterfly is composed of up to 256 processor memory pairs. Each processor's local memory is accessible to other processors via a fast switch; hence it is essentially a shared-memory multiprocessor.

In section 2, we give a brief description of A*. In section 3, we present a parallel version of A* and its implementation for the Traveling Salesman Problem (TSP) and the 15-puzzle problem [NILSSON 80]. In section 4, we present a different way of parallelizing A* and its implementation for the 15-puzzle problem. Section 5 contains concluding remarks.


We assume familiarity with the A* algorithm. See [NILSSON 80] for a good introduction to A*. We will also use the terminology presented in [NILSSON 80]. Here we provide a brief overview of the algorithm.

A* is used to find a least-cost path between a start state and a (set of) goal state(s) of a given state-space graph. The state-space graph is implicitly specified by the start state, a move generator (a procedure that can generate successors of any given state in the state-space graph) and a function to recognize the goal-state(s).

A* maintains two lists OPEN and CLOSED. OPEN is a set of nodes whose successors have not been generated yet. CLOSED is a set of nodes whose successors have been generated. The process of generating successors of a node \( m \) is also referred to as "expanding \( m \)". For a node \( m \) on OPEN, \( g(m) \) is the cost of the current best path from start state to \( m \), \( h(m) \) is a heuristic estimate of the cost of the shortest path between \( m \) and a goal state, and \( f(m) = g(m) + h(m) \) is the overall evaluation of the node \( m \).

In each iteration, A* selects a most promising node \( n \) (i.e., the node with the smallest \( f \)-value) from the OPEN list for expansion, generates its successors and puts the node \( n \) into CLOSED and its successors into OPEN. Whenever a goal-node is chosen for expansion, A* terminates with \( n \) as the solution. It was proved in [NILSSON 80] that if the heuristic estimate \( h \) is admissible, then A* would terminate with an optimal solution (if a solution exists).

3. A SIMPLE DEMAND DRIVEN STRATEGY.

Given \( n \) processors, the simplest strategy would be to let each parallel processor work on one of the current best nodes in the OPEN list. We shall call it simple demand driven strategy, as each processor demands work when needed from the global OPEN list. As discussed in [LI 84], [IRANI 86], this strategy

\[ \text{3 Since there can be more than one path by which a particular node can be reached from the start node, this step is a bit more complicated. See [Nilsson 80] for details.} \]
should not result in much redundant search, hence slowdown due to this strategy should be small.

There are two problems with this approach.

(1) The termination criterion of sequential A* does not work any more, i.e., if a current processor picks up a goal-node m for expansion, the node m may not be the best goal node. But the termination criterion can be easily modified to ensure that termination occurs only after a best solution has been found.

(2) Since the search tree will be accessed by all the processors very frequently, it will have to be maintained in shared memory that is easily accessible to all the processors. Hence loosely coupled architectures such as Hypercube are effectively ruled out. Even for shared memory architectures like Sequent or Butterfly, frequent access to the shared data structures can cause memory contention and degrade the performance.

We have implemented this scheme for solving the Traveling Salesman Problem (TSP) and the 15-puzzle. Next we discuss these implementations and present performance results.

3.1 Simple demand driven parallel control strategy for TSP.

The Traveling Salesman Problem can be stated as follows : Given a set of cities and inter-city distances, find a shortest tour that visits every city exactly once and returns to the starting city. The state-space graph of TSP grows exponentially with the number of cities, hence it appears quite amenable to parallel search.

TSP can be solved using Branch and Bound algorithms. One such algorithm is the LMSK algorithm [LITTLE 63]. The LMSK algorithm performs a search on the circuit space of the input graph. The root node corresponds to the set of all possible circuits. The operators consist of including or excluding an edge. Given a node S, the include child consists of those circuits of S in which a particular edge is included and the exclude child consists of those circuits of S in which that particular edge is excluded\(^4\). The LMSK algorithm (like many other B&B algorithms for searching state space graphs) can be viewed as a typical A* algorithm. In LMSK algorithm, the search graph is a tree - no state is generated more than once. Hence there is no need to check for repeated states; i.e., we do not need to maintain an explicit closed list to check whether a node has already been explored. In LMSK, \(g(n)\) = sum of costs of all include edges on the path from n to start-node in the search tree, and \(h(n)\) is a lower bound on all tours permitted by the reduced cost matrix corresponding to n.

We implemented a parallel version of LMSK using demand-driven control strategy for TSP on the Sequent Balance 21000 and on the Butterfly. We tested the parallel version for upto 20 processors on Sequent, and upto 100 processors on Butterfly. The cost matrices for TSP instances were generated by a uniform random number generator. We found the average slow down factor due to parallel control-strategy to be over .9 even for 100 processors.

Figures 1 and 2 give the actual speedup obtained for different number of processors used. The speedup is fairly linear for as many as 20 processors on Sequent and 100 processors on Butterfly. This shows that the simple demand driven approach is quite effective for parallelizing the LMSK algorithm.

\(^4\) Only one edge is selected for inclusion-exclusion. There is a selection heuristic to choose this edge. The edge selected is the one which is too costly to fore-go currently.
3.2 Simple demand driven parallel control strategy for 15 - PUZZLE.

The 15-puzzle problem [NILSSON 80] can be solved using A*. It exhibits the following properties.

(1) There are many heuristics for the 15-puzzle. The manhattan distance heuristic [NILSSON 80] appears to be the best among the known admissible heuristics for 15-puzzle.

(2) The search space is a graph. Two nodes in this search space are connected by many different paths and there is a need to maintain a closed list to avoid duplicate searches.

(3) 15-puzzle is known to be a difficult problem [KORF 84]. Hence it promises realistic search instances.

We can afford to store the configuration corresponding to each OPEN and CLOSED node element as it takes only 4 words of memory. Evaluating change in h value from a node to any of its children is a unit time operation. Hence time taken for expanding a node is very small.

In our implementation, the data structures for OPEN and CLOSED are as follows. OPEN is a hash table over f values. Nodes with the same f values are chained together in one bucket. Note that using a hash table for OPEN list is simple and efficient for 15-puzzle as it requires only one step to add or delete a node. CLOSED is also implemented as a hash table. This is the traditional data structure used in sequential A*. The hash table implementation of CLOSED and OPEN is especially suited for our parallel implementation for the following reason. In a shared memory system such as Sequent, simultaneous reads are always allowed and mutual exclusion for simultaneous writes can be provided by having a separate lock on each chain. Thus if two processors were to simultaneously insert different elements with different hash values they could do so simultaneously.

At any given instance, all processors executing Parallel A* would be exploring nodes with almost identical costs. So if the hashing function for CLOSED is random over cost space, then nodes of the same cost are likely to have different hash values, and we have very limited contention for writes on CLOSED. In our implementation the hashing function is the product of the diagonal vector of the configuration with [1 9 81 729].

Anomalies in PARALLEL A* for 15 - puzzle.

The nodes expanded by Parallel A* can vary greatly from one run to another run. This happens because there are large number of nodes with the same f-value as the goal node of the puzzle. Even though only some of these nodes need to be expanded to get to the nearest goal node, the evaluation function f is unable to direct the search at this point.

Since it is a matter of chance whether we hit upon the goal node early or late in Parallel A*, the speedup can vary greatly. As a result we observe both acceleration and deceleration anomalies [LAI 83], [LI 84], [QUINN 83]; i.e., the speedup with n processors is sometimes more than n (acceleration anomaly) and is sometimes less than 1 (deceleration anomaly).

To be able to properly evaluate the performance of our parallel A*, we modify the termination criterion so that both sequential and parallel A* terminate only after finding all optimal solutions. This modification ensures that both A* and parallel A* would expand all nodes with f-values less than or equal to f*, where f* is the cost of an optimal solution. This is enough to remove the acceleration and deceleration anomalies.

Performance Results

The modified version of A* was implemented on Sequent and tested on 6 different instances of the 15-puzzle. These instances were arbitrarily chosen such that they had
more than 40 moves for optimal solution to ensure that the search trees of these instances are reasonably large. The slow-down factor due to the parallel-control strategy for 15-puzzle is consistently above 0.98. It is clear from these results that the simple-demand-driven strategy does not result in much redundant search. Figure 3 shows the actual speedup acheived for different number of processors used. The speedup obtained is very poor and tapers off around 3. The reason for the poor performance is that the node expansion in 15-puzzle is very cheap, hence all the processors spend a good part of their time adding or removing elements from the OPEN list. This causes too much memory contention as most of the adds and deletes are on the same buckets of the OPEN list. In the case of TSP, the node expansion time is large, hence the accesses to OPEN are suitably interleaved making the memory contention small. That is why we get almost linear speedup for TSP. Next we present a parallel control strategy for problems for which the node expansion time is small, and present its performance results for the 15-puzzle problem.

4. DISTRIBUTED APPROACH

In this approach each processor has its own local OPEN list. This means that all the processors can select and expand nodes simultaneously without causing contention on the shared OPEN list as before. CLOSED is still maintained as a shared hash table as simultaneous accesses by different processors would tend to be randomly distributed over all the buckets. Recall that two insertions in CLOSED can be done simultaneously if they are done on buckets with different hash values. To ensure that each processor has its share of good nodes, we have a shared BLACKBOARD through which nodes are switched among processors as follows. After selecting a (least f-value) node from its local OPEN list, the processor proceeds with its expansion only if it is within a "tolerable" limit of the best node in the BLACKBOARD. If the selected node is much better than the best node in the BLACKBOARD, then the processor transfers some of its good nodes to the BLACKBOARD before expanding the current node. If the selected node is much worse than the best node in the BLACKBOARD, then the processor transfers some good nodes from the BLACKBOARD to itself, and then reselects a node for expansion.

BLACKBOARD is implemented as a hash table exactly like OPEN. The hash table implementation permits addition and deletion of more than one element in a single step. The choice of tolerance is important, as it affects the number of nodes expanded as well as the amount of node switching between local OPEN lists and the BLACKBOARD. If the tolerance is kept low then nodes will be switched frequently between local OPEN lists and the BLACKBOARD unless the best nodes in all the OPEN lists happen to have the same cost. If the tolerance is high then the node switching would happen less frequently, thus reducing contention on the shared BLACKBOARD. But in this case a processor can possibly expand nodes that are inferior to nodes waiting to be expanded in other processors. In the earlier stages of the search, we can keep the tolerance high as many of the nodes in the shallow part of the tree would be expanded anyway. Towards the end of the search process, there will usually be a lot of nodes on OPEN, many of the same cost as the least cost node on OPEN. In this case, reducing tolerance will not lead to frequent switchings. Furthermore, if the tolerance is kept high at this stage a lot of unnecessary nodes can be expanded. In our implementation, (some what arbitrarily) we keep the tolerance at 4 to begin with and reduce it to 1 when the number of nodes in the BLACKBOARD at the least cost becomes five times the number of processors.

Slow down factor for the distributed approach remains nearly as good (0.97) as it is for the simple demand driven strategy; hence this approach does not expand too many
unnecessary nodes. Since most of the time processors work with their individual OPEN lists, contention for shared data structure is reduced. Distributed approach takes care to see that everybody has a fair share of work. Hence, the actual speedup is fairly linear for up to eight processors on Sequent. Figure 4 shows the actual speedup achieved for 15-puzzle versus the number of processors used.

5. CONCLUDING REMARKS

We have presented two different ways of parallelizing the A* algorithm on a shared memory multiprocessor, and have implemented them for the 15-puzzle problem and the Traveling Salesman Problem. Parallel A* with simple demand driven scheme for solving TSP is essentially same as the parallel algorithm for TSP in [MOHAN 83]. Mohan reports a speedup of 8 on 16 processors on Cm* [MOHAN 83] whereas we obtain a fairly linear speedup for up to 20 processors on Sequent Balance 21000 and up to 100 processors on the Butterfly. The difference in the speedup figures can be attributed to the difference in the architectures of Sequent, Butterfly and CM*. We were unable to test our algorithm for larger number of processors, as we did not have access to a larger Butterfly.

Main reasons for such good performance of the demand driven scheme for TSP are:
(i) Slow-down factor of demand-driven control strategy is close to 1 even for large number of processors, as the search tree of TSP is very bushy. This can be expected for most difficult state-space search problems.
(ii) Granularity is large, i.e., the amount of computation done in expanding a node is non-trivial. This ensures low contention for shared data structures such as the OPEN list and the shared tree.

If we keep increasing the number of processors eventually the speedup curve for TSP would saturate either due to high contention for shared structures or due to poor slow-down factor. Note that the same scheme does not perform well for the 15-puzzle. For this problem the search tree is quite bushy but the granularity is very small. For 15-puzzle, we get much better speedup using the distributed control strategy, as the shared data is accessed less frequently. We are currently implementing this strategy on Butterfly. It is expected that both of the parallel control strategies presented in this paper would be applicable to many other problems solvable by the A* algorithm.

The idea of dividing the central work queue into many queues local to individual processors and allowing exchanges between them to ensure good balance of work in the context of distributed systems is often used (e.g. see [WAH 84]). The division of the central queue into local queues is a necessity in loosely coupled systems, as there is no shared memory. But exchanges between local queues are slow. In shared memory system the division is not necessary, but has the following advantages:
(i) Contention for access to shared data structures is reduced;
(ii) Since each local queue is smaller, manipulation time is reduced in general.
Exchanges between local queues can be done via a shared space. Note that exchanging information between processors is particularly easy, as only pointers need to be passed around. This makes load balancing much easier and less costly than it is in loosely coupled systems.

Our work has shown that it is possible to exploit parallelism in search to get one to two orders of magnitude speedup on shared memory multiprocessors such as Sequent and Butterfly. Given that each processor in these systems is an off-the-shelf microprocessor, these parallel processors can be cost effective high performance computing engines for solving search type problems.

Acknowledgements: We would like to thank Prof. Larry Davis (Centre for Automation
Research, University of Maryland) for giving us access to the Butterfly parallel processor, and Joe DeMartino (Sequent Computer Corp., Oregon) for giving us access to Sequent Balance 21000 system.

6. REFERENCES


Fig. 1. Actual speedup obtained using simple demand-driven-strategy

for TSP on Sequent Balance 21000
Fig. 2. Actual speedup obtained using simple demand-driven-strategy for TSP on Butterfly Parallel Processor
Fig. 3. Actual speedup obtained using simple demand-driven-strategy for the 15-puzzle on Sequent Balance 21000
Fig. 4. Actual speedup obtained using distributed control strategy for the 15-puzzle on Sequent Balance 21000.