Searching Game Trees in Parallel Using SSS*

Subir Bhattacharya and A. Bagchi
Computer Aided Management Centre
Indian Institute of Management Calcutta
P.O. Box 16757, Calcutta - 700 027, INDIA

Abstract

PARSSS* is a parallel formulation of SSS* that is suitable for shared-memory multiprocessor systems. It is based on the distributed tree search paradigm of Ferguson and Korf. The main difficulty in parallelizing SSS* lies in achieving proper coordination between processes running on different subtrees of the game tree. This has been resolved in PARSSS* by the use of a shared array which maintains summary information on all processes that are currently in execution. Problem-independent speed-up values for PARSSS* have been obtained experimentally. It is shown that an earlier algorithm of the authors, called ITERSSS*, which allows SSS* to run in restricted memory, can also be parallelized using the above scheme.

1 Introduction

Shared memory multiprocessor systems, local area networks, and other types of parallel and distributed computer systems have become increasingly available in recent years. This has caused a spurt of activity in the development of parallel versions of existing sequential algorithms. The general objective has been to achieve an effective reduction in net processing time through simultaneous execution of code by more than one processor. In the area of AI search methods, parallel versions of A* [Kumar *et* a/., 1988] and IDA* [Rao *et al,* 1987] have been announced. Alpha-Beta and other game tree search algorithms have been similarly parallelized ([Finkel and Fishburn, 1982, Leifkar and Kanal, 1985]).

An efficient parallelization of Alpha-Beta has recently been achieved by Ferguson and Korf [1988] using their notion of *distributed tree search.* But a corresponding parallelization of SSS* has not yet been reported. Indeed, there appears to be no completely satisfactory parallelization of SSS* in existence. The main hurdle lies in coming up with a suitable implementation for a global OPEN list from which the highest-valued node must be selected at each iteration, *and* from which some non-promising nodes must be purged at intervals. A convenient distributed realization would make it imperative to maintain many local OPEN lists. Coordination between processes would then become a problem, and would have to be accomplished either through shared memory or by message-passing.

In this paper we propose a new parallel implementation of SSS* based on distributed tree search. The basic idea is to let a number of processes execute in parallel, each searching a different subtree of the given game tree, and each running SSS* on a local OPEN list. The game tree can be irregular in shape with a non-uniform branching factor and depth. The version of our algorithm presented here, which we call PARSSS*, achieves coordination through the use of shared memory. Only minor changes are required to get an alternative version where coordination is achieved through message-passing. The maximum number N of processes that can execute at the same time, which can be thought of as being equal to the actual number of physical processors, can be fed as a parameter at runtime. Our experimental results show the approximate variation of speed-up with the number N of processors.

There are two primary reasons for the preference shown by users for Alpha-Beta over SSS* : the apparent simplicity of the Alpha-Beta algorithm, and the high memory requirement of SSS*. It is known, however, that SSS* never examines more terminals than Alpha-Beta and frequently examines less [Pearl, 1984]. In [Bhattacharya and Bagchi, 1986], a method was proposed for running SSS* in restricted memory. SSS* was modified slightly to yield ITERSSS*, which could be fed at runtime the memory M available for use by the OPEN list. For successful operation, M had to lie above a small threshold value Mo. The number of terminals examined by ITERSSS* was a function of M, but never exceeded the number of terminals examined by Alpha-Beta. In PARSSS* each process maintains its own local OPEN list, but it is still possible for the size of a local OPEN list to become unwieldy. To solve the problem we can wed ITERSSS* and PARSSS*. In the resulting

algorithm PARITERSSS*, each process would run ITERSSS* instead of SSS* on its local OPEN, thereby cutting down significantly on the memory requirement.

In Section 2 we explain how SSS* can be modified to run in parallel. The detailed formulation of PARSSS*, together with an example of its operation, is given in Section 3. The next section summarizes our experimental results. PARITERSSS*, the parallel version of ITERSSS*, is described in Section 5, and the concluding section contains suggestions for further work.

2 Running SSS* in Parallel

When searching a game tree T, SSS* sends out simultaneous probes across the entire breadth of T, in contrast with Alpha-Beta which searches T in an essentially left-to-right order. At all instants, SSS* strives to keep in OPEN a representative node from each of the constituent solution trees of T. However, nodes representing sub-optimal solution trees can get purged from OPEN. SSS* scans from left to right the most promising solution tree currently known; if this is an optimal solution tree then the root s gets SOLVED, otherwise a more promising solution tree is found and the search continues.

In SSS*, when a LIVE MAX node x is expanded, all its LIVE MIN immediate successors enter OPEN, since they represent different solution trees in T. We do not know in advance which one of these MIN nodes will ultimately cause x to get solved. Thus, when parallelizing SSS*, it would appear advisable to run independent processes on each of these MIN nodes. This is what we do in PARSSS*. Each process mns on a subtree rooted at a MIN node of T, and all the processes are identical. Initially, the algorithm creates a process for the root node s; this is the only process that runs on a subtree rooted at a MAX node. During execution of this root process, the MAX node s splits up into its MIN successors. The root process continues the search under the leftmost of these MIN nodes, and creates new processes for as many as possible of the other MIN successors of s. Each of the newly created processes maintains its own local OPEN list, and can in its own turn spawn processes at lower level MIN successors. At no instant of time, however, can the total number of processes exceed the given bound N.

How is coordination between processes achieved ? In SSS*, when a SOLVED MIN node x.j is selected from OPEN, its father MAX node x can be labelled SOLVED, and all successors of x can be purged from OPEN. But this cannot be done in PARSSS*. Suppose that a process P is running on the subtree of the game tree rooted at a MIN node y. When P selects a SOLVED MIN node x.j from its local OPEN, other processes may be running on other descendants of x, so it may not be correct to label x SOLVED. To resolve this difficulty, we keep a global array MINPROC in shared memory; for each process, there is an entry in MINPROC specifying the current h-value of the process. The entry for process P stores the associated root MIN node y, and the current h-value of the subtree below y. This value is the maximum of the h-values of nodes in the local OPEN of P *and* the current h-values of the subprocesses spawned by P. When a process P* is spawned by the process P at a LIVE MIN node z, the h-value of P' is initialized to the current h-value of z in P. As the search below z progresses, P* updates its own h-value to the value of the last node selected by it from its local OPEN (assuming no other processes are running below z). Thus when the process P selects a SOLVED MIN node x.j from its local OPEN, the current h-values of the processes spawned by P at nodes below the MAX node x are all available in MINPROC. Among these, those that are no greater than the h-value of x.j are no longer required, and the corresponding processes can be killed. If all satisfy the condition, then x can be labelled SOLVED. Otherwise we must wait until the h-values drop or some other MIN successor of x solves x. Since each process confines its attention to its own local OPEN, the total number of terminals examined by PARSSS* can in general exceed the number examined by SSS*. But PARSSS* takes less time than SSS* since the processes run in parallel.

3 Algorithm PARSSS*

For a description of SSS*, we refer the reader to [Stockman, 1979], [Pearl, 1984, pp. 240-245], and [Bhattacharya and Bagchi, 1986]. Algorithm PARSSS* consists of a short root procedure PARSSSROOT, and a process PARSSS*. Many copies of process PARSSS* are simultaneously in execution, as has been explained above. Before presenting the algorithm, we clarify some of its features below.

(a) Each copy of process PARSSS* has an entry in the global list MINPROC which is located in shared memory. The entry has three components :
   (i) the root node x of the subtree on which the process is running (except for s, all such nodes are MIN nodes);
   (ii) the current h-value of the subtree;
   (iii) the current status, LIVE or SOLVED, of x.

(b) The local OPEN list of each process has been split up for convenience into two sublists, OPEN and LIVEMINS. The list OPEN keeps track of SOLVED nodes and LIVE MAX nodes, while LIVEMINS contains the LIVE MIN nodes on which other processes can be initiated in future. The list LIVEMINS is kept sorted on the depth of the MIN nodes.

(c) The procedure FIRST returns the highest valued node from OPEN U LIVEMINS.

(d) N is an upper bound on the maximum number of concurrent PARSSS* processes that can be spawned by the algorithm. The value of N is fed as a parameter to PARSSSROOT, and is stored in shared memory.

```
procedure PARSSSROOT(s, N);                    (* root program *)
begin
 MINPROC[1].node := s; MINPROC[1].h := ∞;
 MINPROC[1].state := LIVE; N := N - 1;
 spawn PARSSS*(s, 1); wait until PARSSS*(s, 1) terminates;
 output MINPROC [1].h;
end;

process PARSSS*(p, MINPROC_ENTRY);
var  OPEN, LIVEMINS : list of nodes;
     x, x' : node;
     i, j, k, entry_in_MINPROC : integer;

begin
 if p is MAX then                                        (* p = s *)
   begin put p in OPEN; set LIVEMINS empty; end
 else begin set OPEN empty; put p in LIVEMINS; end;
 repeat
   transfer to OPEN from MINPROC all SOLVED descendants
     of p, if any;                                     (* Remark1 *)
   x := FIRST(OPEN U LIVEMINS);
   remove x from OPEN U LIVEMINS;
   update MINPROC[MINPROC_ENTRY].h;        (* Remark2 *)
   k := min {N, SizeOf(LIVEMINS)};
   N := N - k;                                         (* Remark3 *)
   for i := 1 to k do
     begin
       transfer LIVEMINS[i] to MINPROC;
                                 (* at entry entry_in_MINPROC *)
       spawn PARSSS*(LIVEMINS[i], entry_in_MINPROC);
     end;

   case : x terminal and state(x) = LIVE :
          insert x in OPEN with
            h(x) := min {h(x), computed value of x},
            state(x) := SOLVED;

        : x nonterminal MIN and state(x) = LIVE:
          insert x.1 in OPEN with
            h(x.1) := h(x), state(x.1) := LIVE;

        : x nonterminal MAX and state(x) = LIVE:
          for each successor x.j of x do
            insert x.j in LIVEMINS with
              h(x.j) := h(x), state(x.j) := LIVE;

        : x = x'.j MAX and x <> s and state(x) = SOLVED:
          if x is rightmost successor of x' then
            insert x' in OPEN with
              h(x') := h(x), state(x') := SOLVED
          else insert x'.j+1 in OPEN with
              h(x'.j+1) := h(x), state(x'.j+1) := LIVE;

        : x = x'.j MIN and x <> p and state(x) = SOLVED:
                                                  (* Remark4 *)
          remove successors of x' from OPEN U LIVEMINS;
          kill processes running on MIN successors of x' with
            current h-value ≤ h(x) and update N;
          if x' has no remaining descendant in MINPROC then
            insert x' in OPEN with
              h(x') := h(x), state(x') := SOLVED
          else  put back x in OPEN;
   end;
 until (x = p) and (state(x) = SOLVED);
 kill processes running on successors of p and update N;
 MINPROC[MINPROC_ENTRY].state := SOLVED;
 N := N + 1;
end;
```

Remark 1 The SOLVED descendants of p which are being transferred from MINPROC to OPEN correspond to those MIN nodes below p at which processes were initiated earlier from this process and which have now been solved.

Remaik2: On selecting the node x from OPEN U LIVEMINS, the h-value of the process p in MINPROC must be updated to max (h(x), h(q): q is the root of a process below p}. This is done to avoid purging nodes below which other processes are running with higher h-values.

Remark3: The algorithm tries to distribute its workload by initiating independent processes at as many LIVE MIN nodes below p as possible. LIVEMINS is kept sorted by depth, so nodes higher up in the tree get priority in the assignment of processors.

Remark4: When a SOLVED MIN node x = x'.j is selected from OPEN, instead of immediately inserting its father MAX node x' in OPEN, PARSSS* first finds out whether other processes are running at descendants of x\

The two shared databases are MINPROC and N. Only operations on N need to be locked. Although MINPROC is modified by different processes, it is possible to implement it without locking since a specific entry can be modified by only one process. MINPROC is small in size and has no more entries than the maximum number of processes that are allowed to run concurrently.

We illustrate the operation of PARSSS* with an example.

Example : Let the game tree be a uniform binary tree of depth 4, so that there are 16 terminal nodes. Let us suppose that the terminal node values from left to right are as follows

4 4 4 X 4 4 X X 6 5 6 5 5 4 3 X

where X indicates a DON'T-CARE term. We make the simplifying assumption that during the execution of SSS* and PARSSS*, only the examination of terminal nodes takes any significant time, and we ignore all other factors contributing to processing time. Thus our unit of time is the *terminalcount,* which is the time taken to examine one terminal node. This time is taken to be the same for all terminal nodes. The above simplifying assumption is not needed for the correct operation of PARSSS*; its purpose is to make the example easier to follow, and it plays a role in our experiments described in the next section.

On the above sequence, the execution of SSS* takes 12 terminalcounts. Now suppose N = 2, which means there are two physical processors. Table 1 shows what happens when PARSSS* runs on the sequence. Nodes are represented in Dewey decimal notation, with 1 corresponding to a left link and 2 to a right link. Time

is indicated in terminalcounts. Process 1 runs on the root and its left subtree; it spawns Process 2 which runs on the subtree rooted at node 2. At each terminalcount we show the values of a terminal node when it is LIVE (i.e., prior to being examined) and when it is SOLVED (i.e., after being examined).

	Process 1			Process 2		
terminal count	terminal node	LIVE value	SOLVED value	terminal node	LIVE value	SOLVED value
1	1111	∞	4	2111	∞	6
2	1121	∞	4	2121	∞	6
3	1112	4	4	2112	6	5
4	1211	4	4	2122	6	5
5	1212	4	4	2211	5	5
6				2212	5	4
7				2221	4	3

Table 1

PARSSS* executes for 7 terminalcounts. After 5 terminalcounts, the first process solves node 1, but the root node s cannot be solved yet, since the h-value in MINPROC corresponding to process 2 is still too high. Only at the end of the 7th terminalcount can s be solved. Note that h-values in MINPROC are being updated right after FIRST selects the highest-valued node from OPEN.

4 Experimental Results

We conducted some experiments on a VAX 11/750 to find out the order of speed-up obtained by PARSSS* over SSS* when run on uniform game trees (with branching factor b and depth d) with randomly generated terminal values. To get problem-independent results, running time was determined in terminalcounts. The computer system had only one physical processor, but multiprocessing was simulated by setting N to a value greater than one and then running N processes concurrently under the VMS operating system. Programs were written in C. Terminalcount was initialized to zero for the process corresponding to the root node s, and thereafter whenever a new process got spawned, its terminalcount was initialized to the current terminalcount of its father. Since in this environment the speed-up obtained becomes dependent on the order in which processes get scheduled by the operating system from the ready queue, it is necessary to run the same problem instance a number of times. For the purposes of this experiment we randomly selected only one problem instance for each set of (b, d) values. We ran each problem instance 20 times and took the *minimum* of the 20 running times, as shown in Table 2. The corresponding values of the total number of terminals examined by all the processes together are also shown in the table. In order to restrict the number of processes that get created within a reasonable bound, we did not allow processes to be spawned on nodes at heights < 4. Owing to the use of local OPEN and LIVEMIN lists, the speed-ups are less than linear. In a true parallel processing environment it is likely that higher speed-ups would be realised for the following reason. Before declaring a MAX node SOLVED, PARSSS* updates the terminalcount of the current process with the maximum of its terminalcount *and* the terminalcounts of all processes spawned from the current process. When there is a single processor, time slices get allocated to processes in a sequential manner, and no true parallel processing takes place; this puts some restrictions on when processes can get killed. When processes execute simultaneously, it is possible for a process P to be killed by an ancestor process P' in the middle of a timeslice as soon as the h-value of P' is updated by a descendant of P\ This also explains why it is more reasonable to take the *minimum* rather than the *average* of the running times for a specific instance. The execution of a true parallel processing system cannot be simulated exactly on a single processor system because the user has little direct control on the length of the time slice or on the way processes get scheduled from the ready queue. The smaller the time slice, and the closer it is in duration to the time taken to examine a terminal, the closer the analogy.

No.	(b, d)	N	running time (terminal counts)	speed-up	total no. of terminals examined
1	2, 15	1	1874	1.00	1874
2		2	976	1.92	1952
3		5	676	2.77	2850
4		8	558	3.36	3388
5	3, 10	1	3363	1.00	3363
6		2	1900	1.77	3800
7		3	1900	1.77	4494
8		8	1397	2.41	6644
9	5, 7	1	5720	1.00	5720
10		2	3107	1.84	6323
11		5	1616	3.54	8155
12		8	1354	4.22	8322

Table 2

5 Running ITERSSS* in Parallel

A scheme almost identical to the one described above enables us to parallelize ITERSSS* [Bhattacharya and Bagchi, 1986]. The total memory requirements of PARSSS* and SSS* are of the same order. If so much storage is not available, the memory that is available can be distributed among a number of processes; of course, each process must be given the minimum storage space required for running ITERSSS* on the subtree assigned to this process.

In ITERSSS*, every node in OPEN has an

additional field named TYPE which can be either ACTIVE or INACTIVE. ITERSSS* starts by setting the root node s ACTIVE. When an ACTIVE MIN node that is selected from OPEN can not be expanded immediately because of lack of space, its TYPE is changed to INACTIVE. An INACTIVE node can be looked upon as the root of a subtree whose exploration has been temporarily suspended, to be taken up later when the required storage space has been released as a consequence of the selection of a SOLVED ACTIVE MIN node.

In the parallel algorithm PARITERSSS*, while spawning a new process, preference is given to those subtrees whose MIN roots are currently INACTIVE. This is achieved by keeping the list LIVEMINS sorted first on TYPE and then on depth. The function FIRST in PARITERSSS* selects the highest h-valued node from OPEN U LIVEMINS from among the nodes that are currently ACTIVE. But when MINPROC [MINPROC_ENTRY].h is updated, the values of INACTIVE nodes must also be taken into account.

When the root process PARITERSROOT is invoked, two parameters are supplied, viz. N, which is an upper bound on the number of simultaneous processes, and M, which specifies the storage space that each spawned process can utilise for storing the lists OPEN and LIVEMINS. For successful operation of PARITERSSS*, M must be > M_o where M_o = [d/21 * (b-1) + 1. The rest of the algorithm follows PARSSS* closely.

```
procedure PARITERSSSROOT(s, N, M);    (* root program *)
begin
 MINPROC[1].node := s; MINPROC[1].h := ∞ ;
 MINPROC[1].state := LIVE; N := N - 1;
 spawn PARITERSSS*(s, 1, M);
 wait until PARITERSSS*(s, 1, M) terminates;
 output MINPROC[1].h;
end;

process PARITERSSS*(p, MINPROC_ENTRY, M);
var  OPEN, LIVEMINS : list of nodes;
     x, x', y : node;
     i, j, k, entry_in_MINPROC, MM : integer;

begin
 type(p) := ACTIVE; MM := M;
 if p is MAX then                                   (* p = s *)
   begin put p in OPEN; set LIVEMINS empty; end
 else begin set OPEN empty; put p in LIVEMINS; end;
 repeat
   transfer to OPEN from MINPROC all SOLVED descendants
     of p, if any;
   x := FIRST(OPEN U LIVEMINS);
                        (* selects from ACTIVE nodes only *)
   remove x from OPEN U LIVEMINS;
   update MINPROC[MINPROC_ENTRY].h;
                        (* considers INACTIVE nodes also *)
   k := min {N, SizeOf(LIVEMINS)}; N := N - k;

   for i := 1 to k do
     begin
       transfer LIVEMINS[i] to MINPROC;
       spawn PARITERSSS* (LIVEMINS[i],
                entry_in_MINPROC, M);
     end;

   case : x terminal and state(x) = LIVE :
          insert x in OPEN with
            h(x) := min {h(x), computed value of x},
            state(x) := SOLVED, type(x) = ACTIVE;

        : x nonterminal MIN and state(x) = LIVE:
          if (x.1 terminal) or (MM+1> no of sons of x.1) then
            insert x.1 in OPEN with h(x.1) := h(x),
              state(x.1) := LIVE, type(x.1) := ACTIVE
          else put back x in LIVEMINS with
              type(x) := INACTIVE;

        : x nonterminal MAX and state(x) = LIVE:
          MM := MM + 1;
          for each successor x.j of x do
            begin
              insert x.j in LIVEMINS with h(x.j) := h(x),
                state(x.j) := LIVE, type(x.j) := ACTIVE;
              MM := MM - 1;
            end;

        : x = x'.j MAX and x <> s and state(x) = SOLVED:
          if x is rightmost successor of x' then
            insert x' in OPEN with h(x') := h(x),
              state(x') := SOLVED, type(x') := ACTIVE
          else insert x'.j+1 in OPEN with h(x'.j+1) := h(x),
              state(x'.j+1) := LIVE, type(x'.j+1) := ACTIVE;

        : x = x'.j MIN and x <> p and state(x) = SOLVED:
          for each successor y of x' in OPEN U LIVEMINS
                such that h(y) ≤ h(x) do
            begin
              remove the node y from OPEN U LIVEMINS;
              (* the node y can be ACTIVE or INACTIVE *)
              MM := MM + 1;
            end;
          kill processes running on MIN successors of x' with
            current h-value ≤ h(x) and update N;
          y := deepest INACTIVE successor of x' in
                LIVEMINS if any, otherwise null;
          if (x' has no remaining descendant in MINPROC)
            and (y is null) then
              insert x' in OPEN with h(x') := h(x),
                state(x') := SOLVED, type(x') := ACTIVE
          else
            begin
              put back x in OPEN;
              if y is nonnull then type(y) := ACTIVE;
            end;
   end;

   until (x = p) and (state(x) = SOLVED);
   kill processes running on successors of p and update N;
   MINPROC[MINPROC_ENTRY].state := SOLVED;
   N := N + 1;
end;
```

For PARITERSSS* we conducted some experiments on a uniform tree with b = 3 and d = 10. In this case the memory requirement for OPEN in SSS* is 243; and for ITERSSS*, M_o = 11. We ran PARITERSSS* for different combinations of N and M on a randomly gener-

No.	M	N	running time (terminal counts)	speed-up over SSS*	total no. of terminals examined
1	11	1	7207	0.47	7207
2		2	3380	0.99	7009
3		3	2770	1.21	6962
4		8	2220	1.51	8296
5	23	1	6467	0.52	6467
6		2	3301	1.02	6770
7		3	2617	1.28	6787
8		8	2133	1.58	9809
9	36	1	5927	0.57	5927
10		2	3090	1.09	6309
11		3	2409	1.40	6218
12		8	1298	2.59	6453
13	61	1	5439	0.62	5439
14		2	2649	1.27	5351
15		3	1900	1.77	5091
16		8	1249	2.69	5611

Table 3

ated problem instance. Results are given in Table 3. Speed-up computations were made relative to SSS*; since PARITERSSS* allocates memory M to *each* of the processes that get generated, it would not be meaningful to compute speed-up with respect to ITERSSS* running with memory M. Note that when M = 243, ITERSSS* is identical to SSS*. As in the case of PARSSS*, we did not allow processes to be spawned on nodes at heights < 4.

6 Conclusion

The objective of this paper has been to suggest ways to parallelize SSS* and ITERSSS*. Our stress has been on the *formulation* of the parallel algorithms because of our belief that no other completely satisfactory formulation exists in the literature. We have run the algorithms and obtained problem-independent speed-up estimates empirically. These results are only indicative. The algorithms should now be run on true multiprocessor systems and on real game trees from different games, and speed-ups should be determined for each problem on the basis of total running time. It also appears possible to reformulate PARSSS* and PARITERSSS* for distributed systems, though the details of the algorithm would depend on the specific features of the distributed system under consideration. Another interesting area where further work is possible concerns the theoretical derivation of upper and lower bounds on the speed-ups obtainable by PARSSS* and PARITERSSS* under different sets of assumptions.

References

[Bhattacharya and Bagchi, 1986] Subir Bhattacharya and A. Bagchi. Making best use of available memory when searching game trees. *Proc. AAAI-86,* pp. 163-167.

[Ferguson and Korf, 1988] C. Ferguson and R.E. Korf. Distributed tree search and its application to Alpha-Beta pruning. *Proc. AAAJ-88,* pp. 128-132.

[Finkel and Fishburn, 1982] R.D. Finkel and J.P. Fishburn. Parallelism in Alpha-Beta search. *Artificial Intelligence,* Vol. 19, 1982, pp. 89-106.

[Kumar *et al.,* 1988] Vipin Kumar, K. Ramesh and V. Nageshwara Rao. Parallel best-first search of state-space graphs: a summary of results. *Proc. AAA1-88,* pp. 122-127.

[Leifkar and Kanal, 1985] D.B. Leifkar and L.N. Kanal. A hybrid SSS*/Alpha-Beta algorithm for parallel search of game trees. *Proc. IJCA1-85,* pp. 1044 -1046.

[Pearl, 1984] J. Pearl. *Heuristics : Intelligent Search Strategics for Computer Problem Solving.* Addison-Wesley, 1984.

[Rao *et al,* 1987] Nageshwara Rao, V. Kumar and K. Ramesh. A parallel implementation of Iterative Deepening-A*. *Proc. AAAI-87,* pp. 133-138.

[Stockman, 1979] G. Stockman. A minimax algorithm better than Alpha-Beta ? *Artificial Intelligence,* Vol. 12, 1979, pp. 179-196.