MULTI-PROCESSOR GAME-TREE SEARCH

T.A. Marsland
Marius Olafsson
Jonathan Schaeffer

Computing Science Dept.
University of Alberta,
EDMONTON,
Canada T6G 2H1

Final Draft for Advances in Computer Chess 4, London, 17-18 April 1984

ABSTRACT

The design, implementation, and performance of two chess programs running on a multi-processor system are presented. Methods for distributing and enhancing the alpha-beta algorithm in a parallel environment are discussed. With the Principal Variation Splitting method, a four processor chess program yields a 3.35 fold speed-up over the sequential version.

[Recreated from files dated 4 May 1984 in /tony/Papers/papers/acc4]

1. Introduction

The performance of most contemporary chess programs is strongly correlated to the speed of the underlying hardware. The typical chess program is a finely tuned piece of software running on the fastest available computer at the programmer's disposal. With the advent of the micro-processor in the late 1970's, and the subsequent reduction in hardware costs and improved availability, a new dimension of performance became accessible to the computer chess programmer - parallelism. It is now feasible to consider chess programs distributed over many CPUs as a relatively inexpensive alternative to main-frame computing.

Ideally, if we distribute a program over N computers we would like to have that program run N times faster. Unfortunately, this usually turns out to be a very difficult goal to achieve. There may be problems with the algorithm (some algorithms have more inherent parallelism than others), the hardware configuration (the interconnection pattern of processors may not reflect the needs of the algorithm), the cost of communicating between processors, the amount of information shared between processors, or a variety of other reasons.

This paper discusses the design, implementation and performance of two parallel chess programs. The parallelism is achieved through the division of work done by the alpha-beta algorithm between multiple processors. In the sequential environment, alpha-beta is usually enhanced by a variety of features to increase performance further. These enhancements are analyzed to assess their impact in a parallel environment. Unfortunately, the methods employed fall short of achieving a linear speed-up and an explanation for this are presented.

2. Background

There have been several radically different approaches taken to the parallelization of the alpha-beta algorithm. Baudet[1] partitioned the search window, so that each processor looked at disjoint windows. For example, when four processors are available, an alpha-beta window [-999,999] could be partitioned into disjoint intervals of [-999,-500], [-499,0], [1,499], and [500,999], with each window being assigned a different processor. All processors would search the game tree rooted at the same node but should only do a fraction of the work of the full alpha-beta search because of the smaller window size. One processor is guaranteed to have the correct answer fall within its search window. Baudet argues that this approach is very quickly self-limiting (every processor has at least a minimal tree to search) and that a speed-up of 5-6 is the maximum attainable regardless of the number of processors used. Although his experiments were not performed in the context of a chess program, his results seem generally applicable.

In contrast, the mandatory work first approach[2] attempts to schedule the work so that nodes which must be searched are given preference over nodes whose need has not yet been established. In this approach, all nodes in the minimal alpha-beta tree are the initial set of nodes that must be examined. As processors return with the results of searching nodes, this information is assessed and may result in a set of new nodes being added to the list of work to do. Essentially, a master process maintains a list of outstanding work, and idle slave processes are assigned work from the top of this list. The major drawback with this scheme is that at any instant, many different parts of the tree are under evaluation. Large memory requirements are needed to store the outstanding work and the partially evaluated tree that ties all this information together.

A third approach is to assign individual processors to separate subtrees.[3] At the simplest level, one could imagine a master process that generates all the legal moves at the root node of the tree and then passes each move in turn to an available processor, giving it the necessary alpha-beta window and a search depth. This is probably the most intuitively obvious way of parallelizing alpha-beta, but it is not without its disadvantages. In particular, the major drawback with this tree-splitting scheme is that since the amount of work each processor is asked to do is comparatively large, it is often very inconvenient to share all the information that a processor generates (for example, transposition table entries).

3. Principal Variation Splitting Algorithm

The Principal Variation Splitting algorithm (PVSplit) used in the implementations described in this paper, is a variant of the tree splitting scheme. The algorithm must be used in conjunction with progressive deepening for best results. At the outset of iteration D+1, all the processors combine to traverse the best move sequence found at iteration D, since this sequence of moves (called the principal variation) has a very high probability of also being the best to depth D+1. Since we expect the score returned by this search to be the best for this iteration, we presume that all other moves are inferior until proven otherwise. Accordingly, we search the principal variation with a full window [alpha,beta], but search all subsequent moves with a minimal window [alpha,alpha+1]. Thus at the root of a sub-tree of depth D, the remaining moves are each given in turn to a processor for evaluation. In the event that a search of one of these moves "fails high" (exceeds our artificial upper bound of alpha+1), then this move must be re-searched since it is the newest principal variation for this iteration.

Figure 1 presents a pseudo-code representation of the PVSplit algorithm as actually implemented. The code reflects an underlying process structure of one "master" process to allocate work and multiple "slave" processes (usually one per CPU) to carry out the the work. Note that for the principal variation, PVS is applied recursively.

4. Performance Obstacles

One measure of success/failure for a parallel algorithm is the "speed-up" achieved; speed-up being defined as the ratio of the time taken by the non-parallel version to the parallel version. The ultimate objective in parallel programming is to achieve a speed-up linear with the number of processors. Unfortunately, there are some inherent problems with parallelism and the alpha-beta algorithm in particular that make this a very difficult objective.

Degradation in performance can be attributed to three costs: communication overhead, search overhead, and synchronization overhead. Communication overhead is the additional burden placed on a parallel program of having to spend time sending messages back and forth between processors. It is a function of the hardware and software protocols used to connect the computers, and of the amount of communication the program actually does. Given that these protocols are fixed, then if communication costs are significant, the programmer must revise his algorithm to minimize the frequency and size of messages sent between processors.

Search overhead may be inherent in the algorithm chosen. Sequential alpha-beta has "complete" information in the sense that at any point in the search, the program has access to all previously computed results. For example, not only the best alpha bound, but also the transposition table entries form a previous iteration. The parallel environment, where sub-trees are searched on different processors, gives rise to an interesting dilemma. If each processor is to have complete information in the sequential sense, then this requires all processors to share their information with each other. But this increases the amount of communication required, and hence the communication overhead. On the other hand, if each processor takes little or no advantage of results from other processors, it is almost certain that extra nodes will have to be searched. For example, one processor may already have searched a particular sub-tree and hence have a local transposition table entry for it. With incomplete information, a different processor that comes across the same sub-tree will be forced to re-search it. It is a trade-off; increased communication costs versus extra nodes searched. Search overhead is the cost attributable to the extra number of nodes searched in the parallel version of the program compared to the sequential version.

Synchronization overhead is algorithm dependent and occurs any time a processor must wait idle until others complete their tasks, before being allowed to proceed with any new work. In our implementation, this primarily occurs near the end of each iteration of progressive deepening. Before proceeding to start the next iteration, the program must wait until all the moves of the current iteration have been analyzed. This may result in one or more processors remaining idle until the last move returns. Obviously synchronization delays are to be avoided if at all possible. Unfortunately, in our experience, algorithms that employ some degree of synchronization are usually simpler to understand, implement and, perhaps most importantly, debug.

In order to maximize program efficiency (i.e., increase speed-up) it is necessary to minimize the overheads. These overheads may be present due to the algorithm employed, the actual implementation, or external factors (e.g. operating system overhead). In any event, it is important to analyze the results and discover means for reducing any significant overheads. A balance must be struck, as often a decrease in a particular type of overhead is out weighed by a corresponding increase in another. For example, search overhead is usually inversely proportional to communication overhead.

5. Enhancements

Practice has shown that sequential alpha-beta can be supplemented by a variety of features to further increase performance. These include progressive deepening, transposition tables, and refutation tables.[4] However, in a parallel environment it is not obvious whether these features are still of benefit. Further, there may be enhancements to alpha-beta which work well in a parallel environment, but either do not work well or are not possible in the sequential case.

Transposition tables are of significant benefit to a sequential chess program. In a parallel environment, they give rise to a search vs communication overhead trade-off. One possible implementation of transposition tables would be to have a global table maintained by the master process. Before a slave process searches an interior node, a message is sent to the master to look this position up in the table and report back the results. Similarly, when a slave has finished its analysis of a sub-tree, it can send a message to the master to add this new result to the table. The table is global in the sense that each slave has the benefit of each other's work. This scheme helps minimize search overhead by sharing information between the slaves. On the other hand, it dramatically increases the volume of messages and hence, increases the communication overhead.

An alternate implementation of transposition tables would have each slave maintaining its own local table and not sharing this information with the other slaves. The scheme has the advantage of eliminating all the extra communication costs incurred by the previous scheme. However, a slave may end up searching a sub-tree that is not in its transposition table, but is possibly in another slave's, resulting in replicated work and increased search overhead. This expected increase in search overhead may be offset by the fact that each slave's table contains less information than a global table would and therefore table collisions can be minimized. Experimental results seem to indicate that local tables do indeed yield better time performance than global tables, even though more nodes are searched.[5]

Refutation tables provide an inexpensive means of conveying the important information contained in a transposition table, i.e. the principal variation and refutations for each alternative move. These tables are fairly small and inexpensive to maintain. In a parallel environment, they have proved to be a significant enhancement. It is possible that a particular move has been searched by one processor at some earlier time, and now must be re-searched by a different processor (such as would occur with progressive deepening). The refutation line can be stored by the master and sent to the slave when assigned that move to search. Then at least, the slave has some knowledge about the move by which it can guide the subsequent tree search. This is a simple way of allowing processors to share knowledge.

Progressive deepening is of benefit in a parallel implementation, but suffers from a serious defect not encountered in the sequential environment. Progressive deepening involves searching all moves to a depth D, sorting the moves based on the scores returned, and then searching to depth D+1. In a parallel version, where individual moves are assigned to separate processors, one must wait until all the moves of one iteration are complete before the sort can be done and the next iteration started. This idleness is reflected in synchronization overhead. Some solutions to this problem involve either having idle processors start work on the next iteration, or re-assigning these processors to help the busy ones complete their task. Both solutions require an extra degree of complexity and process/processor management.

Since full sharing of information between processors can be expensive (e.g. global transposition tables) one can look for ways of sharing only important information. One way is to communicate new values of alpha/beta as they change. Once a master receives information from a slave that updates the current alpha-beta window, that information can be transmitted to the other slaves. In one of our implementations (Phoenix) interrupts are used to ensure the speedy transmission of new search windows to other processors. Lack of this feature had been thought to be a serious disadvantage, but in our case its inclusion produced only a small positive benefit.

An issue that arises when dealing with multiple processors is that of scheduling. We would like to assign work (sub-trees) to processors in such a way as to minimize both processor idle time and search overhead. One way of accomplishing the latter is to maintain a record of which moves were searched on which processor. Then, at some future point (perhaps the next iteration), instead of arbitrarily allocating a move to a processor, try and assign the move to a processor that has already seen this move before. This scheme should enhance the utility of the local transposition table. A variation on this has been tried, with as yet, no appreciable gains due to over-loading of the transposition table.

6. Implementation

Our first experiments with a multiprocessor system were performed on four SUN workstation (Motorola 68000 based) processor boards, with software support provided by a VAX/UNIX system, as described in recent paper [ref]. While some of these results, specifically those pertaining to the Parabelle program, are summarized here, the more recent results for Phoenix have been obtained using a more powerful and more flexible system which relies on a 10 Mbit Ethernet (a bus-oriented local area network) to couple four SUN workstations and two VAX/UNIX processors. The increased flexibility comes from the simpler set-up and re-configuration of the processors, and the ease with which they may communicate with each other.In particular, it is possilbe to create a execution environment, in which to conduct our experiments. We call this environment a "Virtual Tree Machine" (VTM).

In a tree machine physical processors are interconnected in a tree-structured manner such that each node (processor) can only communicate directly with its parent and children nodes. In a **virtual** tree machine physical processors are replaced by **processes** under operating system control, and wired interconnections are replaced by virtual communication paths. Our VTM provides the full capabilities of the UNIX operating system to each node in the "processor tree". This includes file handling, debugging and performance monitoring. By using the Ethernet as the communication medium it is possible to create the "machine" automatically from a topological map of the desired system, before control is given to the application at hand. To do this, the user simply creates a file containing the description of the sibling relations in the desired processor tree. Also the UNIX system can be used to control the virtual nodes so that more than one node can reside on a physical machine. Since the machine exists in a timeshared environment, sharing resources with other users of the facilities, the cost of experimentation is greatly reduced. On the other hand, the performance may be affected whenever other users require service from a processor. Finally, since communication paths exist between all physical processors (over Ethernet) no fundamental limitations are placed on either processor tree depth or fanout.

The implementation of the VTM is based on the Interprocess Communication Primitives introduced in the latest version of Unix (4.2bsd) [ref]. These primitives provide an elegant abstraction from the raw network protocols ultimately used by communicating processes. This abstraction presents the user with a clean, reliable two way communication path based on the client/server model of connection establishment [.ref.]. That is, the communicating processes are divided into servers (passive component accepting a connection) and clients (active component initiating a connection).

In accordance with this model our VTM consists of a collection of **node-servers** one running on each physical computer in the intended configuration. These node-servers then create node-processes by recursively traversing an internal representation of the user supplied configuration. Once the whole tree as been created and communication paths established, the node processes pass control to the application processes, which can now communicate over reliable two-way ports.

6.1. Example

Assume that the desired configuration for a specific tree machine is as shown in Figure 2.

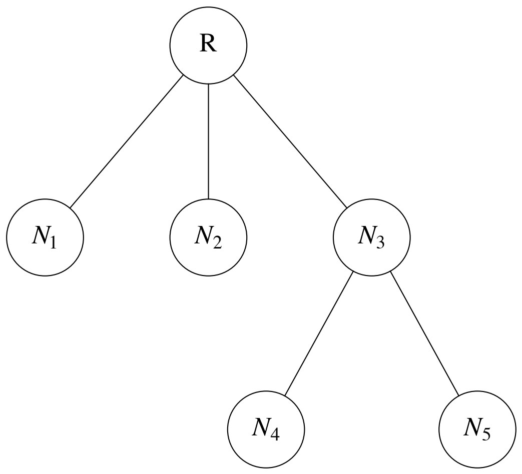
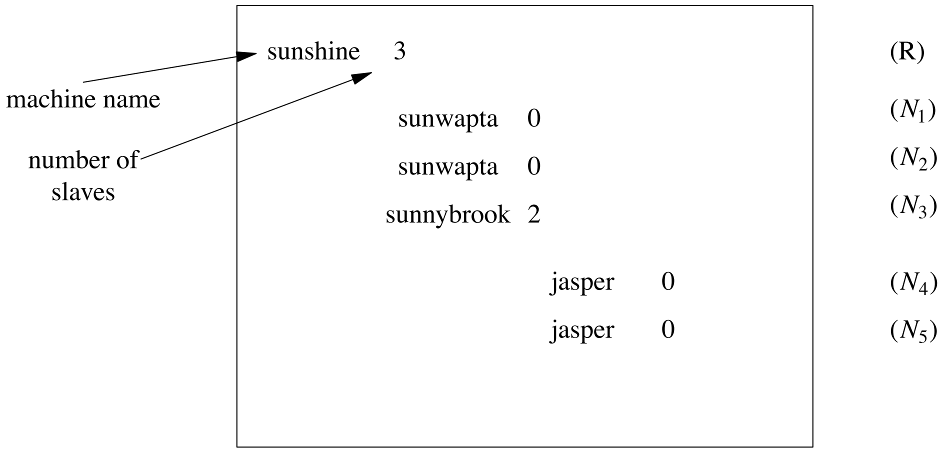


Figure 2. A Desired Processor Tree

Now the nodes must be assigned to physical processors. This is done explicitly, for example:

The root (R) on *sunshine*
Nodes N1, N2 on *sunwapta*
Node N3 on *sunnybrook*
Nodes N4, N5 on *jasper*

A "configuration file" is then created, containing the names of the machines involved and how the nodes are distributed amongst them, Figure 3.



```
sunshine    3                                  (R)
        sunwapta    0                          (N1)
        sunwapta    0                          (N2)
        sunnybrook  2                          (N3)
                  jasper    0                  (N4)
                  jasper    0                  (N5)
```

Figure 3. Tree-machine Definition

When the root process is executed on *sunshine* the configuration file is read and an internal representation of the configuration tree created. This representation is passed to the node-servers on *sunwapta* (where two nodes are created) and to *sunnybrook* (where one node is created). From *sunnybrook* the tree is forwarded to *jasper* (where two nodes are created), Figure 4.

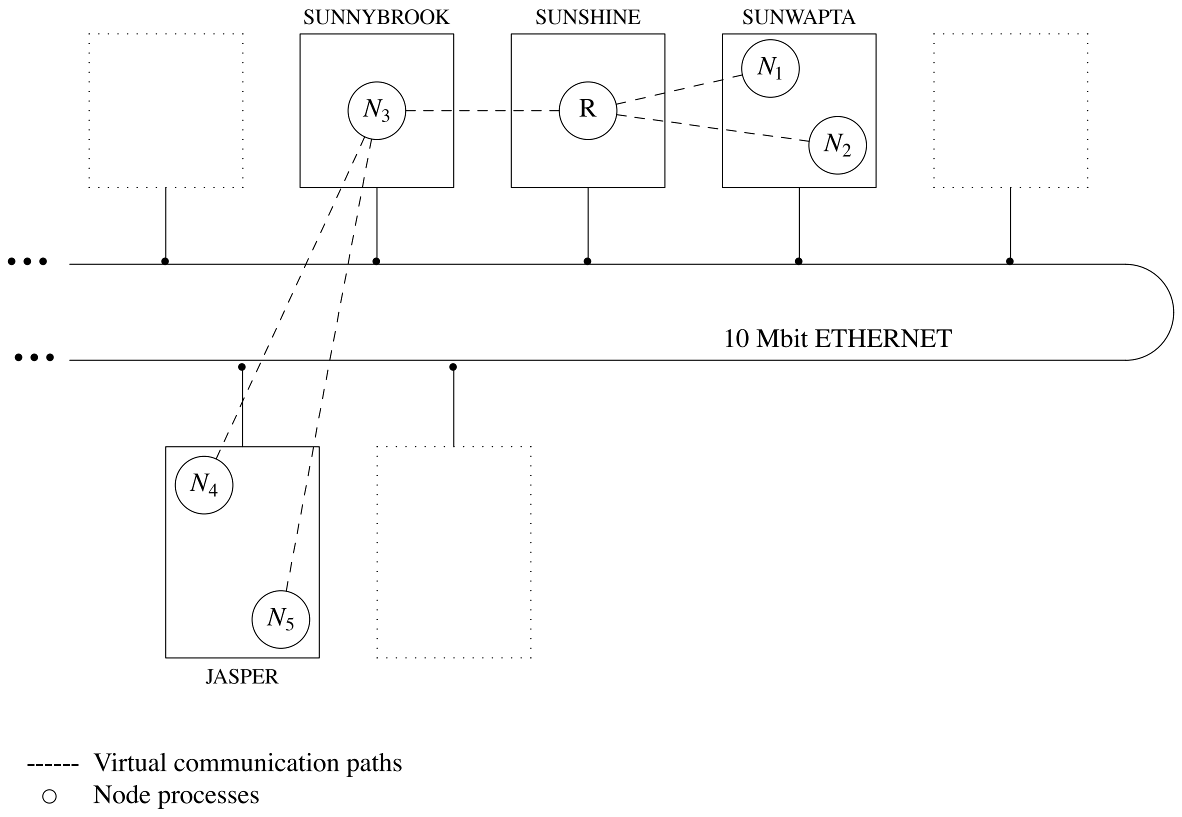


Figure 4. Mapping of Processes onto Hardware Configuration

The primary advantages of our new system are: lower cost, faster turnaround for experiments, easier debugging and performance monitoring, experimentation with different processor tree depth and fanout is possible, and node processes run under operating system control.

The major disadvantages come from the need to obtain accurate timing results. For example, the operating system overhead and the need to serve other users. These disadvantages are not too serious since the operating system provides worthwhile services (and so is acceptable) and in principle the operating system could be modified to deny access to other clients for the duration of an experiment (even though it may run for many hours or days). In our case we relied on overnight and weekend executions to reduce the impact of other user requests.

7. Results

Parallel implementations of two chess programs have been developed and are currently running on our system. The programs, Parabelle (based on Ken Thompson's 1974 program Tinkerbelle) and Phoenix (based on Jonathan Schaeffer's 1983 WCCC entry), vary greatly in their amount of chess knowledge and tree searching efficiency. Figure xxx shows the speed-ups achieved for both programs for various search depths. Since Phoenix is considerably faster than Parabelle, it is meaningful to compare Phoenix's 7 ply searches against Parabelle's 6 ply searches (both take very close to the same amount of real time). At these depths, both programs achieve speed-ups in the range of 3.2 - 3.4 for four processors. Unfortunately, at this time results with more than four processors are not available.

There are several interesting observations to be made from the results presented in Figures xxx and yyy. First, it is possible to achieve greater than a linear speed-up (for example, position xxx). This phenomenon can be caused by a variety of reasons, two of which are mentioned. Since slave processes return in indeterminate order, the ordering of moves at the root node can be different from the sequential version. This can have a positive as well as negative effect on the amount of searching required. Another means by which greater than linear speed-up can be achieved is if alpha/beta values are shared between processors. If one processor returns with a new alpha bound and that information is immediately communicated to the other slaves, it is possible that the new tighter search bounds will result in a significant decrease in work that the slave has to do in searching its current move.

Second, some positions exhibit very poor speed-ups (for example, position xxx). Again, there are many possible reasons for this. One possibility is poor alpha/beta information. While sub-trees are being searched with the current alpha/beta bounds, one slave could be in the process of discovering a new principle variation, thus producing a new, better alpha bound. Until this sub-tree is fully searched, the new alpha bound is not available and other slaves may be spending a great deal of effort refuting moves using the current alpha bound, when the new bound would make the refutation simpler.

Third, speed-ups increase with search depth. Both Parabelle and Phoenix exhibit better parallelism with deeper searches. With the deeper searches, much of the overhead incurred through use of parallelism is dissipated over a longer time-interval and hence has less impact.

A cursory glance at Figure xxx shows that the experimental results fall short of the ideal linear speedup (N processors giving an N-fold speedup). It is important to analyze the reasons for the slow-down in order to best understand limitations of the approach and to devise ways of improving this performance.

Time overhead (TO) is defined as the percentage loss in speed-up of using N processors compared to the ideal speedup. This can be expressed as

$$TO = N * \frac{Time\_using\_N\_CPUs}{Time\_using\_1\_CPU} - 1$$

This is an easily attained quantitative measure of the total loss due to overhead. The principle reasons for the loss in performance, as explained in Section 4, are due to communication overhead (CO), search overhead (SO), and synchronization overhead (SY). The overheads are related by

$$TO = CO + SO + SY$$

Communication overhead is the cost attributable to the sending of the messages. This cost is not easily discerned from the tables given. In order to estimate this cost, Phoenix counts the number of messages communicated and multiplies this by the CPU cost per message, as determined by a special timing program. Note in Figure xxx, that CO is a relatively insignificant cost and effectively becomes negligible as the depth of search increases.

The loss attributable to search overhead can be approximated by using the observation that the number of nodes searched is directly proportional to the time spent searching. Thus SO can be estimated by comparing the number of nodes searched:

$$SO = \frac{Nodes\_searched\_for\_N\_CPUs}{Nodes\_searched\_for\_1\_CPU} - 1$$

In Figure xxx, it can be seen that SO gradually increases with the number of processors used. The remaining cost, synchronization overhead, can be estimated using the known values of TO, CO and SO. Referring again to Figure xxx, it is clear that as the number of processors increases, synchronization cost becomes the primary reason for loss of performance. This is easily explained as the more processors there are, the more processors potentially remain idle when synchronizing.

A new version of Phoenix is being developed with the intent of minimizing overhead, particularly synchronization overhead. It is clear that as the number of processors used increases, so does the amount of overhead. Eventually we will reach a limiting plateau when the addition of an extra processor yields no (or even negative) benefit. Extrapolating the results in Figure xxx indicates that this might occur with Phoenix using xxx processors and achieving a speedup of xxx. By minimizing overhead, one can try and push this plateau as high as possible. It is desirable to find an algorithm where processors can be arbitrarily added to the system; each addition incrementally improving the speedup. The prospect of running 1000 processors or more in parallel is not very far in the future, and it is not clear that the algorithms we use for 2 - 10 processors will be sufficient to meet this challenge.

8. Conclusions

From the preceeding it is clear that there is still considerable work to be done. There is no experimental data available for parallel alpha-beta implementations beyond eight processors. The literature is full of simulations predicting tremendous speed-ups. Unfortunately, there exists a large gap between theory and practise, in part because the theoretical models simulation results do not properly account for the real costs that are the invevitable price an actual implementation must pay.

The writing of efficient parallel programs for computer chess is still in the experimental phase. Countless hundreds of machine hours have been devoted to chasing "good" ideas which one would expect to work well in a parallel environment but frequently return nothing for the effort expended. It is relatively easy to come up with an implementation that consistently gives a degree of parallelism; it is another matter to come up with one that yields a speed-up asymptotic with the number of processors employed. Until we understand parallel programming much better, and have better tools for scheduling processors, asymptotic behaviour will remain a very distant goal.

References

1. G.M. Baudet, “On the branching factor of the alpha-beta pruning algorithm,” *Artificial Intelligence,* 10, pp. 173-199 (1978).
2. S.G. Akl and M.M. Newborn, “The principal continuation and the killer heuristic,” *Proc. ACM National Conference,* pp. 466-473, Seattle (1977).
3. J.P. Fishburn, “Analysis of speedup in distributed algorithms,” TR 431, Computer Science, Univ. of Wisconsin, Madison (May 1981).

4. M.S. Campbell and T.A. Marsland, “A Comparison of Minimax Tree Search Algorithms,” *Artificial Intelligence,* 20, pp. 347-367 (1983).

5. T.A. Marsland and F. Popowich, “Parallel Game-Tree Search,” *IEEE Transaction on Pattern Analysis and Machine Intelligence,* 7, 4, pp. 442-452 (July 1985).

Parallel-Phoenix Summary (7-ply)

	total node count				time in seconds				average speedup			
POS	CPU 1	CPU 2	CPU 3	CPU 4	CPU 1	CPU 2	CPU 3	CPU 4	2	3	4	Move
1	76165	51534	62098	67271	322	120	103	90	2.68	3.12	3.57	d6d1!
2	522474	546931	601879	621155	1707	1015	830	861	1.68	2.50	1.98	e4e5
3	498499	550371	702146	629026	2005	1148	1058	774	1.74	1.89	2.59	f6f5!
4	518130	670296	668699	710602	2029	1337	961	832	1.51	2.11	2.43	e5e6!
5	5331731	5089591	5381744	5643336	20450	10065	7845	6002	2.30	2.60	3.40	a2a4!
6	134470	146935	191163	200357	609	291	254	209	2.90	2.39	2.91	g5g6!
7	3567485	4002634	3315003	3775075	13657	7919	4591	3926	1.72	2.97	3.47	h5f6!
8	62175	78789	65695	72853	251	175	113	93	1.43	2.22	2.69	f4f5!
9	2334959	2606696	2375343	2663327	9262	5389	3342	2906	1.71	2.77	3.18	d1e1
10	1596471	1780676	1937378	1833982	5154	2900	2167	1582	1.77	2.37	3.25	c6e5!
11	1943732	2715190	2598904	2664318	7238	5300	3729	2871	1.36	1.94	2.52	f2f4!
12	787593	883528	892458	874475	2704	1549	1046	882	1.74	2.58	3.60	d7f5!
13	2472462	2570648	2891894	2804866	10133	5323	3966	3044	1.90	2.55	3.32	a1c1
14	278119	313488	316771	326510	988	590	424	359	1.67	2.33	2.75	d1e1!
15	312984	340998	330165	325052	1055	584	398	308	1.80	2.65	3.42	g4g7!
16	612388	704207	682160	641886	2540	1524	1085	812	1.66	2.34	3.12	d2e4!
17	692962	692024	736435	707120	2729	1380	1116	857	1.97	2.44	3.18	h7h5!
18	3058867	2355166	2864961	1662059	12802	5101	4230	1966	2.50	3.20	6.51	f7f5
19	749960	813759	796431	877581	2569	1410	965	816	1.82	2.66	3.14	e8e4!
20	2957488	3180509	2796635	2321337	12086	6701	3976	2390	1.80	3.30	5.50	c3b5
21	1181570	1234291	1232048	1228862	4977	2305	1506	1146	2.15	3.30	4.34	f5h6!
22	4882915	4739679	7158402	5611695	17507	8637	9279	5451	2.20	1.88	3.21	e6e5
23	1436618	1745974	1727602	1730761	5907	3818	2573	2093	1.54	2.29	2.82	c8f5
24	1901209	2062619	2103969	2180858	6526	3641	3334	3072	1.79	1.95	2.12	f2f4!
TTL	37911426	39876533	42429983	40174364	145207	78222	58891	43342	1.85	2.46	3.35	17
	1579642	1661522	1767916	1673931	6050	3259	2453	1805	1.84	2.48	3.25	

```
Process Master( pos, maxdepth )
{
     /* Iterative deepening details, such as sorting after    */
     /* each iteration are omitted.                     */
     for( depth = 1; depth <= maxdepth; depth++ )
     {
          score = -PVS( pos, depth, -INFINITY, +INFINITY );
     }
     return( score );
}
Process Slave;
{
        for( ; ; )
        {
                ReceiveMaster( pos, depth, alpha, beta );
                value = -AlphaBeta( pos, depth-1, alpha, beta );
                SendMaster( pos, value );
        }
}

Function PVS( pos, depth, alpha, beta )
{
     if( depth == 0 )
          return( Evaluate( pos ) );
     /* Generate move list indexed from 0 - #moves */
     #moves = Generate( pos, moves[] );
     if( #moves == 0 )
          return( Evaluate( pos ) );

     /* Recursively search principle variation */
     /* Details of actually making/retracting moves omitted   */
     score = -PVS( pos.mv[0], depth-1, -beta, -alpha );

     /* Use all the CPUs to search rest of moves in parallel  */
     move = 1;                 /* Index into mv*/
     value = -INFINITY;             /* Value of node*/
     idle = 0;                 /* #CPUs idle   */
     /* Loop until all work has been sent and returned   */
     while( idle < numberofslaves )
     {
          /* First numberofslaves CPUs must be sent work */
          /* before we can receive from them        */
          if( move > numberofslaves )
               who = ReceiveSlave( position, value );
          score = MAX( score, value );
          if( score > alpha )
          {
               /* Fail-high.  Research with full window*/
               alpha = score;
               SendSlave( who, position, depth-1, -beta, -alpha );
          }
          else if( move > #moves )
               /* No more moves.  Mark CPU as idle */
               idle = idle + 1;
          else {
               /* Search next move with minimal window   */
               SendSlave( who, pos.mv[move], depth-1, -alpha-1, -alpha );
               move = move + 1;
          }
     }
     return( score );
}
```

Figure 1. Principal Variation Splitting.