KnightCap - a parallel chess program on the AP1000+

Andrew Tridgell
`Andrew.Tridgell@anu.edu.au`

CAP Project
Australian National University

Abstract

KnightCap is a parallel chess program developed for the AP1000+ running AP/Linux. It uses a parallel MTD(f) search that takes full advantage of the hardware capabilities of the AP1000+ to achieve good speedups. The program also demonstrates the advantages of AP/Linux over the traditional CellOS environment.

1 Introduction

The report describes an experiment in parallel game tree search on the Fujitsu AP1000+[2]. The experiment was conducted using a parallel chess program called KnightCap developed to run under AP/Linux[1] on the AP1000+.

The parallel game tree search algorithm uses a evaluation parallelism methodology rather than the more traditional tree partitioning methods which results in a very simple but effective parallel algorithm.

The resulting chess program has performed well in real chess games against a large number of human and computer opponents.

2 The chess game tree search problem

The most commonly used algorithms for chess computers are based on game tree search. In a game tree search the computer evaluates the game tree of future positions that can be reached from the current position and tries to find the move which provides the best obtainable position. Because the game tree is far too large to search completely it is instead searched only to a limited depth and leaf nodes are evaluated using an evaluation function which attempts to evaluate the winning chances without further searching.

```
procedure minimax(depth)
  if (depth == 0) then
    return evaluate()
  endif
  value = -INFINITY
  for each move m
    make_move(m)
    value = max(value, -minimax(depth-1))
    unmake_move(m)
  end
  return value
end
```

Figure 1: Pseudo-code for minimax

The simplest game tree search algorithm is the minimax algorithm. A pseudo-code implementation of minimax is shown in Figure 1

The minimax algorithm returns the so-called "minimax value" of the game tree. A simple modification of the algorithm allows the return of the move that achieves this value as well.

3 The alpha-beta algorithm

The minimax algorithm is not the most efficient way to search a game tree. The problem is that it searches a lot of unnecessary parts of the game tree, resulting in a lot of wasted time. In general it will search approximately b^d leaf nodes where d is the depth being searched and b is the "branching factor" of the tree. The branching factor is the average number of moves available

```
procedure alphabeta(depth, alpha, beta)
  if (depth == 0) then
    return evaluate()
  endif
  value = -INFINITY
  for each move m
    make_move(m)
    v = alphabeta(depth-1, -beta, -alpha)
    value = max(value, -v)
    unmake_move(m)
    alpha = max(alpha, value)
    if (alpha >= beta) then
       return alpha
    endif
  end
  return value
end
```

Figure 2: Pseudo-code for alpha-beta

in any given position.

A much better algorithm is the alpha-beta algorithm. This takes advantage of the fact that many parts of the tree do not need to be searched to reduce the number of leaf nodes.

A pseudo-code implementation of alpha-beta is shown in Figure 2

Alpha-beta reduces the expected number of leaf nodes to approximately $b^{d/2}$ as long as the moves at each point in the tree are well ordered. Sufficiently good ordering can be achieved using some simple heuristics.

When called with an initial value for alpha of $-\infty$ and an initial value for beta of ∞ the alpha-beta algorithm produces the same result as the minimax algorithm but much faster.

For our purposes a more interesting set of initial values is to set beta=alpha+1. This is called a "minimal window" search and runs very fast because of the large number of cutoffs that are obtained in the tree.

The minimal window search does not return the minimax value of the game tree, instead it answers the question "is the minimax value of the tree greater than alpha". This means that each minimal window alpha-beta search bisects the range of possible minimax values for the tree.

Figure 3 shows a simple algorithm for determining the minimax value of the game tree using a succession of minimal window alpha-beta searches.

```
procedure bisection(depth)
  lower = -INFINITY
  upper = INFINITY
  while (lower < upper) do
     t = choose_next(lower, upper)
     v = alphabeta(depth, t-1, t)
     if (v >= t) then
        lower = v
     else
        upper = v
     endif
  end
  return lower
end
```

Figure 3: Pseudo-code for a minimal window bisection search

The speed of this algorithm largely depends on the choices made by the choose_next() algorithm. A simple bisection search would be achieved using just (lower+upper)/2. More sophisticated choices can produce better performance. The choose_next() algorithm that is most often used is the MTD(f) algorithm[3].

4 Parallel game tree search

There are two fundamentally different approaches to parallel game tree search. The first (and most common) is to partition the game tree so that each processor evaluates a different section of the tree. The results are then combined to find the evaluation (and best move) for the whole tree.

The problem with this first approach is that typical game trees for games like chess are highly irregular, making the partitioning problem very difficult. The amount of time it will take to evaluate a branch in the tree is very difficult to predict accurately.

The second approach, the one adopted in KnightCap, is to divide the evaluation space across the processors so that each processor searches the same game tree but with a different evaluation window. This approach turns out to be quite effective for small to moderate numbers of processors.

5 Evaluation parallelism

The minimal window bisection search shown in Figure 3 lends itself naturally to a parallel implementation. The while loop is implemented in parallel across all the CPUs, with each CPU searching the same position but with a different parameter passed to the minimal window alpha-beta search.

A record is kept of which CPU is searching with what parameters and when any CPU finishes a alpha-beta run then the result of that run is used to narrow the global lower/upper parameters. A new choice of alpha-beta parameter is then chosen for that CPU and it again starts searching. At the same time, any CPU that is already searching with alpha-beta parameters which are outside the new global lower/upper range is interrupted and restarted with new parameters inside the lower/upper range.

6 Distributed hash tables

At first glance it may seem that the CPUs will be duplicating a enormous amount of work, as they are all searching the same game tree, thus leading to very poor parallel speedups. This problem is avoided by using a global hash table along with local move ordering information.

The hash table in a chess program is a area of memory that stores the results of searches over parts of the game tree. Whenever a "node" in the game tree is completed the resulting evaluation is stored in a hash table along with the depth of the search at that point in the tree. Before searching each node a check is made to see if the node has already been searched with a depth at least as large as the current depth, and if so then the hashed result is used.

For a parallel game tree search the hash table is stored globally, distributed across all the CPUs in the machine. This means that all CPUs have access to the partial tree search results of all other CPUs. This global storage prevents CPUs from doing much of the redundant searching that would otherwise be performed.

n=1	n=10	n=20	n=40
3017	963	480	265

Table 1: Time to reach depth 8 for varying numbers of cells (in seconds)

n=1	n=10	n=20	n=40
3257	1029	579	401

Table 2: Time to reach depth 10 for varying numbers of cells with null moves and razoring enabled (in seconds)

In the case of the AP1000+ the put() and get() operations supported in the communications hardware provide the ideal mechanism for supporting global hash tables. The ordering of hash insertions is not critical so a CPU is able to initiate a hash table store using put() without waiting for the operation to complete. In a similar fashion it is possible to overlap the computation of the evaluation function with the get() operation which fetches a global hash table entry during a lookup.

Even with a global hash table the CPUs would tend to perform many redundant tree searches if they all used the same order of move generation at each node. The level of redundant searching is further reduced by computing the move ordering heuristics using locally stored move ordering data (the "history" data that helps with move ordering). With local move ordering and a global hash table the number of redundant searches is reduced to an acceptable level so that a quite efficient parallel search is produced.

7 Parallel speedup

The parallel chess algorithm was tested on a 64 cell AP1000+ running AP/Linux with 64 MB of RAM per cell.

The results shown in Figure 1 show the time taken to complete a depth 8 search using four different numbers of cells. The overall speedup with 40 cells is approximately 11, and a speedup of 6 is reached with just 20 cells. This compares well with simple tree based parallelism methods, particularly when the simplicity of this technique is considered.

Similar results are shown in Figure 2 but this time a depth 10 search is completed and two commonly used forward pruning techniques are applied during the search. In particular R=2 null-moves are used along with a simple move razoring technique. These techniques dramatically change the shape of the game tree and greatly reduce the amount of time taken to reach a specified search depth. They also appear to reduce the available parallelism in the game tree, as is evidenced by the lower parallel speedup.

It should be noted that all speedup results of this kind are highly position dependent, although the results shown are typical of KnightCaps performance. The asynchronous nature of the parallel algorithm means that the game trees searched with different numbers of CPUs can vary quite substantially, making exact comparisons impossible.

8 AP/Linux integration

One aim of this project was to provide a demonstrator for the AP/Linux operating system on the AP1000+. A competition-level chess program is a complex piece of code which requires many operating system facilities not available under the traditional CellOS environment provided with the AP1000+.

In particular, KnightCap uses the X windows system along with the Mesa graphics library to provide a 3-dimensional rendered chess board.

KnightCap also uses system-V shared memory to facilitate the communications between separate tasks handling the display and control of the program and the chess search engine. The display task also handles automatic interaction with the Internet Chess Server, allowing KnightCap to play continuously against any human challengers who wish to play it.[1]

9 Future work

Work on KnightCap is now concentrating on automatic learning algorithms to improve the search and evaluation functions through automatic play on the Internet Chess Servers.

[1] KnightCap has recently achieved a rating of 2532 on the Internet Chess Server after several thousand games

10 Conclusion

KnightCap has demonstrated a simple but effective algorithm for parallel game tree search which is ideally suited to the environment provided by the Fujitsu AP1000+ parallel computers running AP/Linux. A parallel efficiency of around 50% is achievable with a small to moderate number of processors.

11 Acknowledgements

The author acknowledges the support provided by the ANU-Fujitsu CAP Research Program.

References

[1] D. Sitsky A. Tridgell, P. Mackerras and D. Walsh. AP/Linux - a modern OS for the AP1000+. In *PCW'96*, October 1996.

[2] H. Ishihata, T. Horie, S. Inano, T. Shimizu, S. Kato, and M. Ikesaka. Third generation message passing computer AP1000. In *International Symposium on Supercomputing*, pages 45 – 55, November 1991.

[3] Aske Plaat. *Research Re: search & Research*. PhD thesis, Tinbergen Institute and Department of Computer Science, 1996.